木構造を扱うアルゴリズムの変換による合成†

中川 裕志††

プログラム変換という paradigm においては，記述が容易で明快な宣言的プログラムを変換によって必ずしもわかりやすくはないが効率の良いプログラムを得ることになる．従来のプログラム変換の研究においては，データ構造を扱うアルゴリズムに関する部分があまり進んでおらず，実用的なプログラム変換システムへ向けての一つの障害になっていた．本論文では，とくに木構造における挿入，削除のアルゴリズムを，プログラム変換によって合成しようとする試みについて述べる．対象とする言語は宣言的，手続き的の両方の記述が容易な Prolog を選ぶ．ここでの変換の特徴は，まず木構造を一度リストに展開し挿入，削除あるいはポインタの付け替えといった操作をリスト上で行う．これによって複雑な木構造上での操作を考えやすくすることができる．適当な操作の後に再び変換によって木構造を直接扱うアルゴリズムを得る．この変換は必ずしも等価変換でない部分も含むが，人間のアルゴリズム発案のための思考において使われている知識を取り込んでおり，将来の自動変換システムへの足掛かりになると考えられる．とくに本論文の方法においてその有効性が顕著だったのは，同じ結果を導出する再計算を避ける，という知識であり木構造を扱うアルゴリズムの根底をなしていることがわかった．

1. 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], Tarnlund[6], Kahn[7], 竹島[8], 中村[9], Reddy[10], (3)関数型プログラムに対して効率化を行うもの (Burstall[11], Darlington[12),13], Wand[14]) に分類されるであろう．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，前記の文献 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)の路線においてデータ構造，とくに木構造を扱ってみようとする試みに関して述べる．

2. 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．ただし情報を chunk 単位で計った場合の短期記憶の容量は，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ときには，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．たとえば次のような heuristic が浮かんでくる．

H1. 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．

H2. 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．

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，計算機のメモリは，基本的に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

† Prolog Program Transformation of Tree Manipulation Algorithm by HIROSHI NAKAGAWA (Department of Computer Engineering, Faculty of Engineering, Yokohama National University).

†† 横浜国立大学工学部情報工学科

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であり，変換結果としては，直接木構造を扱う効率の良いプログラムを得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．

3. 宣言的なプログラム

宣言的なプログラムでは，2章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 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．*x は[*x | *ry]の構造からわかるようにリストの CAR 部なので，アトムである．もちろん append を与えられた第3引数から第1，第2引数を生成するように用いた場合の解の生成順序に従って，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)
```

4. 変　　換

この章では，木への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，2章で述べた 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．まず 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(*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得られる．

```
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 全体に及んで行う．この名前変えを済ませると，append はもはや何ら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．fold 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．なおこの例からわかるように H1. H2. は等価性を保存してはいない．

5. 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，まず木 t(*l, *x, *r) から目的要素 *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], *y).           (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 *a 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 d1 を導入する．d1 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).
```

これは見つけた *a の削除に対応する．変換過程をすべ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))←
    d 1(*c, *e, *d).                                     (9.7)
d 1(t([ ], *x, *r), *r, *x)←.                            (9.8)
d 1(t(*l, *x, *r), t(*n, *x, *r), *z)←
    d 1(*l, *n, *z).                                     (9.9)
```

(9.1)は終了条件である．(9.2)は heuristic H 2 による NIL 削除による木の不変性を表している．(9.3), (9.4)は2分探索アルゴリズムを表している．(9.5), (9.6)は探索によって見つけた *a に子供が一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表している．(9.7)は探索によって見つけた *a に子供が二つ付いている場合に，新述語 d 1 を呼ぶものである．d 1(*t, *r, *x) は木 *t の左部分木の最左要素 *x を取り出し，残りの部分木 *r を返すものである．(9.8)は最左の要素が発見された場合であり，(9.9)は左部分木の左側をたどる操作を表している．したがって，(9.7)は，*a を根とする木の右部分木の最左要素を削除する *a の場所に置き換える操作を表している．このプログラムでは変換のステップ数は新述語 d 1 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．

6. 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説明する．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([ ], [ ], *h)←.
bltree(*z, t(*l, *x, *r, *n), *n)←
    append(*l 1, [*x | *r 1], *z),
    bltree(*l 1, *l, *m),
    bltree(*r 1, *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，木の性質を表している．ここで問題になるのは test に fail して backtrack したとき，次にいかなる木構造を生成すればよいかである．これは人間がアル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，概略次のような形のプログラムが得られる．

```
bltree(*z, f(*x))←append(*), append(**),
    bltree, bltree, test 1(*x).
bltree(*z, g(*x))←
    append(*1), append(**1),
    bltree, bltree, test 2(*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，H 2 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．

```
p(*a, *b, f(*z))←q, r(*a, *b, *z), test 1(*z).
p(*a, *b, g(*z))←q, r(*a, *b, *z), test 2(*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．

H3. (11.2)において，各 assertion ごとに異なる test 1, test 2… の部分をまとめた新述語 s を導入し，(11.2)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))←test 1(＊z).
s(＊z, g(＊z))←test 2(＊z).
    ⋮          ⋮                    (11.3)
```

これにより，r(＊a, ＊b, ＊x) の計算は1回しか行われなくなる．明らかにこの変換は，(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．

7. 例（AVL-バランス木への要素の挿入）

6章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もつ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(＊i 1, [＊x | ＊r 1], ＊z),
    bltree(＊l 1, ＊l, ＊m),
  bltree(＊r 1, ＊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．まず，(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 H 1. 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)
```

(13.4), (13.5)の ＊g は木 ＊f の高さであり，bltree 中のテスト用 goal である test ((12.2)で定義されている）によって計算されることになる．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, ＊h),
  t(＊i, ＊j, ＊k, ＊l), ＊l)←＊a＜＊f, ＊a＜＊c,
  tins(＊a, ＊b, ＊m), tins([ ], ＊d, ＊n),
  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 H 2 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1回 unfold し同様に H 2 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([ ], ＊g, ＊r),
  append(＊q, [＊f | ＊r], ＊s),
```

append(∗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場合分けを行う．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場合を分離する．

append(∗o, [∗c | ∗q], ∗s),
append(∗p, [∗f | ∗r], ∗q)
→append∗q, [∗f | ∗r], ∗s),
append(∗c | ∗p], ∗q)

この変換に伴って(13.7)の head に出力として表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 を消去すると次の二つの clause が得られる．

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 のペアが現れる．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，次のような変換を行う．

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, ∗y, ∗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1回だけ unify されるだけなので，計算量は少ない．最後に H3. 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．結果は次のとおりである．

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, ∗d, ∗f, ∗g, ∗v, ∗w,
∗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). (13.13)
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). (13.14)

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1回多く unfold し，さらに append の結合法則を2回適用する．すなわち ∗a≧∗f, ∗a＜∗c の場合 unfold によって元の木 t(t(∗b, ∗c, t(∗d, ∗e, ∗f)), ∗i, ∗j) …(ただし木の高さは省略して表現してある)…に対応して，次のような append の並びが現れる．

append(∗x, [∗i | ∗j], ∗s),
append(∗b, [∗c | ∗y], ∗x),
append(∗d, [∗e | ∗f], ∗y)

ここでまず2，3番目の append の対に結合法則を適用し，その結果の append の並びの1，2番目の

append の対に対して再び結合法則を適用すると，次の結果が得られる．

```
append(*y, [*e | *x], *s),
  append(*f, [*i | *j], *x),
append(*b, [*c | *d], *y)
```

これに対応する木の構造は次のようになる．

```
t(t(*b, *c, *d), *e, t(*f, *i, *j))
```

上記の結合法則および木構造の変換以外の部分は単一 LL-回転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．最終的な結果をすべて載せると長くなるので省略する．

8. 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述べた．とく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思われる．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予定である．

謝辞 本研究にあたっては電総研の佐藤泰介氏の“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，情報大の平賀譲氏の“人間の思考の基本の一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．電総研の中島秀之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 を与えてくれる．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．なお本研究は文部省科学研究費（特定研究一多次元知識情報処理）に予算のバックグランドを得ている．

参考文献

1) Partsch, H. et al.: Program Transformation Systems, *ACM Comput. Surv.*, Vol. 15, No. 3, pp. 199-236 (1983).
2) Hogger, C. J.: Derivation of Logic Programs, *J. ACM*, Vol. 28, No. 2, pp. 372-379 (1981).
3) Bible, W.: *Syntax-Directed, Semantic-Supported Program Synthesis, Artificial Intelligence 14*, North-Holland (1980).
4) Sato, T.: Transformational Logic Program Synthesis, Proc. of FGCS '84 (1984).
5) H. Tamaki and T. Sato: Unfold/Fold Transformation of Logic Programs, 2nd Logic Programming Conference (1984).
6) Tarnlund, S.-A. and 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, 1st 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 Conference '84 (1984).
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 Conference '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. and Darlington J.: A Transformation System for Developing Recursive Programs, *J. ACM*, Vol. 24, No. 1, pp. 44-67 (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, No. 1, pp. 1-46 (1981).
14) Wand, M.: Continuation-Based Program Transformation Strategies, *J. ACM*, Vol. 27, No. 1, pp. 164-180 (1980).

(昭和59年11月28日受付)
(昭和60年 2 月21日採録)