LA_003

後継関数を持つリスト型非線形再帰プログラムに対する再帰除去法

Recursion Removal Method for List-processing Non-linear Recursive Programs with Descent Functions

市川 祐輔* 上田 和紀*
Yusuke ICHIKAWA Kazunori UEDA

1 はじめに

再帰プログラムは反復プログラムに比べて書きやすく読みやすい場合が多いが，計算機で実行する際には手続呼出しとスタック操作が必要である．そのため，インライン展開ができない，局所参照性が悪いなどのプログラム最適化上の問題を引き起こす．そこで，再帰プログラムを，計算量を増加させずにスタックを使用しない反復プログラムに変換する再帰除去法が1970年代より研究されてきた．

我々は，単一後継関数を持つ再帰プログラム (Recursive Program with One Descent Functions (RPOD)) に対する再帰除去法である**累積関数法**[6] を提案した．累積関数法とは，RPODを，重複計算及び再帰を除去し，線形オーダー反復プログラム，対数関数オーダー反復プログラム，閉式に変換する手法である．ここでRPODとは，$f(x) = \mathbf{if}\ p(x)\ \mathbf{then}\ b(x)\ \mathbf{else}\ a(c(x), f(d(x)), f(d^2(x)), \ldots, f(d^n(x)))$ の形式で記述される再帰プログラムである．数論的再帰プログラムに対する累積関数法の適用例としては，フィボナッチ関数，Double-Hailstorm関数などがあり [6]，さらに数式処理システム上の再帰除去システムとして Mathematica 上の実装についても報告した [5]．

本稿では，この累積関数法を拡張し，リスト型再帰プログラムに対する再帰除去を示す．リスト型再帰プログラムの例題として，プログラム変換の例として使用されている [7] ハノイの塔のプログラムを取り上げる．

本稿の貢献は，2点ある．一つ目は従来の累積関数法 [6] を拡張することにより適用対象を広げたことである．それにより複数の後継関数 (再帰呼び出し内で引数に適用される関数) を持つ関数に対しても，累積関数法が適用可能となった．この拡張の過程において，後継関数の制限を定義域の観点から定義した点 (定義2の条件 (2)) は，従来の手法である後継関数同士の関係で定義すること [2] と対比すると，筆者の知る限り新規である．二つ目は，従来の累積関数法が数論的再帰プログラムのみの例を示していたことに対して，リスト型再帰プログラムに対しても累積関数法の適用を示したことである．

2 累積関数法

累積関数法は，非線形再帰プログラム (自分自身への再帰呼び出しを二つ以上含む再帰プログラム) を対象とした再帰除去法である．本稿の累積関数法の対象プログラムは，以下のように定義される．

定義 1 RPMD 再帰プログラム $f(x)$ が，以下の形式で記述されれば，RPMD (recursive program with multiple descent functions) と呼ぶ．

$$f(x,y) = \mathbf{if}\ p(x)\ \mathbf{then}\ b(x,y)\ \mathbf{else}\ a(c(x,y), f(d(x), r_1(y)), f(d(x), r_2(y)), \ldots, f(d(x), r_k(y)))$$

ここで，x 及び y はベクトルでもよい．そして a を補助関数，d 及び r_i を後継関数と呼ぶ．従来の累積関数法の対象であるRPOD[6] と対比すると，RPMDにおいては上記の関数 r_i が加えられ，複数の後継関数を持つ再帰プログラムに対しても適用できるよう拡張されている．

定義 2 RPMD に対する**累積関数** f を以下の (1) 及び (2) の性質をもつRPMDとする．そのとき，以下の性質 (3) 及び (4) をもつ関数 $h(v,u,Y)$ は f に関する累積関数という．

(1) $p(d^N(x))$ を満たす最小の自然数 N が存在する．
(2) $\exists Y.\forall r \in \{r_1, r_2, \ldots, r_k\}.\forall y \in Y.r(y) \in Y$．ここで Y は変数の組の有限集合である．
(3) $\neg p(u)$ が成り立つとき，以下の式が任意の v 及び y に対して成り立つ

$$a(v, f(u, r_1(y)), f(u, r_2(y)), \ldots, f(u, r_k(y))) = a(h(v,u,Y), f(d(u), r_1(y)), f(d(u), r_2(y)), \ldots, f(d(u), r_k(y)))$$

(4) h は f への再帰呼び出しまたは自由変数を含まない

ここで，集合 Y は後継関数 r_i によって生成される変数のすべての組の集合を意味する (後述するように，本稿のハノイの塔のプログラムでは，ペグを示す3つの変数により生成される6つの組 `r1(a,b,c)=(a,b,c)`，`r2(a,b,c)=(a,c,b)` などから集合 Y が構成される．その意図するところは，再帰呼び出しによって現れる可能性のある変数の組を集合 Y で包含させて表現することにある)．条件 (2) は，例えば二種類の後継関数 d_1，d_2 がある場合に，$d_1^{k_1}(d_2^{k_2}(x)) = d_2^{k_3}(d(d_1^{k_4}(x)))$ (k は自然数) のように複数の後継関数がある場合の関係 [2] を，定義域に着目して定義したものである．したがって，複数の後継関数 r_i が，定義域 Y について閉じている状態を意味する (集合 Y が有限となる)．なお，条件 (2) を満たすためには，与えられたいくつかの後継関数から集合 Y を構成し，それに対応したその他の後継関数 r_i を発見する必要がある (ハノイの塔のプログラムにおいても，後述のように r_2 及び r_3 は与えられた後継関

* 早稲田大学理工学術院
Faculty of Science and Engineering, Waseda University

数であるが，その他の r_1 及び r_4〜r_6 を発見する必要がある)．この発見アルゴリズムの開発は今後の課題であるが，筆者は再帰呼び出しの関係のグラフから導出できると考える．

定理 1 累積関数定理

RPMDf が累積関数を持つならば，以下の反復プログラムに変換できる．

$$
\begin{array}{l}
floop(x,y) = \textbf{if } p(x) \textbf{ then } \quad \textbf{return } b(x,y) \quad \textbf{else} \\
\quad \textbf{begin } v := c(x,y); u := d(x); \\
\quad\quad Y := \{r_1(y), r_2(y) \ldots, r_k(y)\}; \\
\quad\quad \textbf{while not } p(u) \textbf{ do} \\
\quad\quad\quad \textbf{begin } v := h(v,u,Y);\ u := d(u); \textbf{ end}; \\
\quad\quad \textbf{return } \quad a(v, b(u, r_1(y)), b(u, r_2(y)) \ldots, b(u, r_k(y)) \\
\quad \textbf{end}.
\end{array}
$$

証明 *1* 任意の入力 x,y に対して，$f(x,y)$ の計算が停止する場合，$p(x)$ が成り立てば $f(x,y) = floop(x,y) = b(x,y)$ である．もし $\neg p(x)$ が成り立てば，$N = \min\{i | 1 \leq i \wedge p(d^i(x))\}$ を満たす自然数 N が存在し，以下の式が成り立つ．

$$
\begin{array}{l}
f(x,y) \\
= a(c(x,y), f(d(x), r_1(y)), \ldots, f(d(x), r_k(y))) \\
= a(h(c(x,y), d(x), Y), f(d^2(x), r_1(y)), \ldots, f(d^2(x), r_k(y))) \\
= a(h(h(c(x,y), d(x), Y), d^2(x), Y), \\
\qquad\qquad f(d^2(x), r_1(y)), \ldots, f(d^2(x), r_k(y))) \\
= \ldots \\
= a(h(\ldots, h(h(c(x,y), d(x), Y), d^2(x), Y), \ldots), \\
\quad f(d^N(x), r_1(y)), f(d^N(x), r_2(y)), \ldots, f(d^N(x), r_k(y))) \\
= a(h(\ldots, h(h(c(x,y), d(x), Y), d^2(x), Y), \ldots), \\
\quad b(d^N(x), r_1(y)), b(d^N(x), r_2(y)), \ldots, b(d^N(x), r_k(y)))
\end{array}
$$

以上の式は，以下の計算列で再現される．

$$
\begin{array}{l}
v := c(x,y);\ u := d(x);\ Y := \{r_1(y), r_2(y), \ldots, r_k(y)\}; \\
\left.\begin{array}{l} v := h(v,u,Y); \quad u := d(x); \\ \quad \ldots \\ v := h(v,u,Y); \quad u := d(x); \end{array}\right\} N-1 \\
\textbf{return } \quad a(v, b(u, u_1(y)), b(u, u_2(y)), \ldots, b(u, u_k(y)).
\end{array}
$$

この計算列は反復プログラムと同様の挙動を示す．なお，これらの計算列では値渡しを仮定する．任意の入力 x,y に対して，$f(x,y)$ の計算が停止しない場合，反復プログラム $floop(x,y)$ は停止しない．(証明終)

3 ハノイの塔のプログラムに対する累積関数法の適用例

拡張された本稿の累積関数法を，ハノイの塔のプログラムに適用して説明する．ハノイの塔のプログラムとは，3つのペグのうちの一つにある n 枚の大小のある円盤を，大きな円盤を小さな円盤の上に乗せない条件の下で，他の一つのペグへ移す順番を出力するプログラムである．一般的に，ハノイの塔のプログラムは以下のように定義される．

```
myh(n,a,b,c)= if n==1 then '(a,c) else
        cons(myh(n-1,a,c,b),
           cons('(a,c),myh(n-1,b,a,c)))
```

このままの形式では累積関数法を適用できないため，以下のプログラムに変換する．

```
myh(n,a,b,c)= if n==1 then '(a, c) else
  progn( v:=cons(null,cons(cons(a,c),null));
         rplaca(v,myh(n-1,a,c,b));
         rplacd(cdr(v),myh(n-1,b,a,c));
         v)
```

なお，この変換については，今後の研究が必要である．ただし，その方針は，補助関数 cons を用いる関数に対して rplaca, rplacd を用いるパターンを自動的に作成する手法が考えられる．ちなみにこの変換後のプログラムも非線形の計算量のため，計算時間は改善されていない (実験 1 参照)．

累積関数法は，3つのステップにより行われる．すなわち，(1) 補助関数及び後継関数の特定，(2) 累積関数の導出，(3) 累積関数定理の適用，である．

3.1 補助関数及び後継関数の特定

まず，補助関数を定義する．

```
a(v,vv,u1,u2,u3,u4,u5,u6) def=
 progn(
  rplaca(vv[1],u2);  rplacd(cdr(vv[1]),u3);
  rplaca(vv[2],u1);  rplacd(cdr(vv[2]),u5);
  rplaca(vv[3],u4);  rplacd(cdr(vv[3]),u1);
  rplaca(vv[4],u3);  rplacd(cdr(vv[4]),u6);
  rplaca(vv[5],u6);  rplacd(cdr(vv[5]),u2);
  rplaca(vv[6],u5);  rplacd(cdr(vv[6]),u4);
  v)
```

この補助関数の定義にあわせると，元のプログラムは，以下のように記述できる．なお，vv[i] は，リスト vv の i 番目の要素とする．ただし，紙面の都合により，上記式の rplaca(vv[3])〜rplaca(vv[6]) の行 (及び式変換により対応する部分) は以降では省略する．

```
myh(n,a,b,c)= if n==1 then '(a,c) else
 (progn
  s=[[null,[a,c]],[null,[a,b]],[null,[b,c]],
     [null,[b,a]],[null,[c,b]],[null,[c,a]]];
  rplaca(s[1],     myh(n-1,a,c,b));
  rplacd(cdr(s[1]),myh(n-1,b,a,c));
  rplaca(s[2],     myh(n-1,a,b,c));
  rplacd(cdr(s[2]),myh(n-1,a,c,b));
  ...
  car(s))
```

次に，後継関数は以下のように定義される：

```
d(n)=n-1,
r1(a,b,c)=(a,b,c), r2(a,b,c)=(a,c,b),
r3(a,b,c)=(b,a,c), r4(a,b,c)=(b,c,a),
r5(a,b,c)=(c,a,b), r6(a,b,c)=(c,b,a).
```

ここで，累積関数の定義の条件 [2] における集合 Y は，$Y = \{(a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)\}$ となる．そうすると，ハノイの塔のプログラムは，以下のように補助関数で記述可能である．

```
progn(
 s:= [[null,[a,c]],[null,[a,b]],[null,[b,c]],
```

```
      [null,[b,a]],[null,[c,b]],[null,[c,a]]];
  rplaca(s[1],     myh(n-1,a,c,b));
  rplacd(cdr(s[1]),myh(n-1,b,a,c));
  rplaca(s[2],     myh(n-1,a,b,c));
  rplacd(cdr(s[2]),myh(n-1,a,c,b));
  ...
 car(s))
= λs.a(car(s),s,
 myh(d(n),r1(a,b,c)),myh(d(n),r2(a,b,c)),
 myh(d(n),r3(a,b,c)),myh(d(n),r4(a,b,c)),
 myh(d(n),r5(a,b,c)),myh(d(n),r6(a,b,c)))
 [[null,[a,c]],[null,[a,b]],[null,[b,c]],
  [null,[b,a]],[null,[c,b]],[null,[c,a]]]
```

3.2 累積関数の導出

次に累積関数を導出する．これは，myh(d(n),r1(a,b,c))～myh(d(n),r6(a,b,c)) の unfold[8] によって行う．(具体的手法は，以下のように補助関数の第一，第二引数を変数 v，vv とおき，累積関数の定義にあわせて，unfold 後にこれらに対応する部分 (累積関数) を導く)

```
 myh(n,a,b,c)=
a(v,vv,
myh(d(n),r1(a,b,c)),myh(d(n),r2(a,b,c)),
myh(d(n),r3(a,b,c)),myh(d(n),r4(a,b,c)),
myh(d(n),r5(a,b,c)),myh(d(n),r6(a,b,c)))
=progn(
  rplaca(vv[1],     myh(d(n),a,c,b));
  rplacd(cdr(vv[1]),myh(d(n),b,a,c));
  rplaca(vv[2],     myh(d(n),a,b,c));
  rplacd(cdr(vv[2]),myh(d(n),a,c,b));
 ...
 v)
```

unfold 後に，まとめると以下のようになる．

```
progn(
s:=[[null,[a,c]],[null,[a,b]],[null,[b,c]],
    [null,[b,a]],[null,[c,b]],[null,[c,a]]];
  rplaca(vv[1],s[2]); rplacd(cdr(vv[1]),s[3]);
  rplaca(vv[2],s[1]); rplacd(cdr(vv[2]),s[5]);
  ...
  rplaca(s[1],     myh(d(d(n)),a,c,b));
  rplacd(cdr(s[1]),myh(d(d(n)),b,a,c));
  rplaca(s[2],     myh(d(d(n)),a,b,c));
  rplacd(cdr(s[2]),myh(d(d(n)),a,c,b));
  ...
 v)
```

そして補助関数で書き直すと，以下の式が導き出される．ただし，この変換は全体として等価な関係ではなく，リストの要素に着目して等価な関係である．LISP 関数の equal の意味で等価な関係である．この等価性については後述する．

```
=a(v,progn(
  s:=[[null,[a,c]],[null,[a,b]],[null,[b,c]],
      [null,[bj,a]],[null,[c,b]],[null,[c,a]]];
      rplaca(vv[1],s[2]);rplacd(cdr(vv[1]),s[3]);
      rplaca(vv[2],s[1]);rplacd(cdr(vv[2]),s[5]);
  ...
 s)
 ,myh(d(d(n)),r1(a,b,c)),
  myh(d(d(n)),r2(a,b,c)),
  ...)
```

したがって，以下の累積関数が導出される．

```
h_{1}(v,vv,Y):=v;
h_{2}(v,vv,Y):=
   progn(
     s:=[[null,[a,c]],[null,[a,b]],[null,[b,c]],
         [null,[b,a]],[null,[c,b]],[null,[c,a]]];
     rplaca(vv[1],s[2]); rplacd(cdr(vv[1]),s[3]);
     rplaca(vv[2],s[1]); rplacd(cdr(vv[2]),s[5]);
     ...
     s)
```

3.3 累積関数定理の適用

累積関数定理から，反復プログラムが生成される．

```
myh(n,a,b,c)=if n==1 then [a,c] else
 progn(
  s:=[[null,[a,c]],[null,[a,b]],[null,[b,c]],
      [null,[b,a]],[null,[c,b]],[null,[c,a]]];
  v:=car(s);  u:=n-1;
  while(u!=1){ v:=v; vv:=h(v,vv,Y); u:=u-1; };
   rplaca(vv[1],[a,b]); rplacd(cdr(vv[1]),[b,c]);
   rplaca(vv[2],[a,c]); rplacd(cdr(vv[2]),[c,b]);
   ...
   return(v);)
```

ここでプログラムの等価性について述べる．一般的にプログラム変換の前後のプログラムは等価である (等価でなければならない) が，本稿の変換過程では破壊的関数を用いて，以下の意味で一部プログラムの等価性を破壊した変換を行っている．すなわち，本論文で考えるプログラム変換はグラフ構造としての S 式を必ずしも保存しないが，木構造としての S 式を保存する (したがって変換後の反復プログラムに対して，rplaca などの副作用を持つ関数を適用しない限り，変換前のプログラムと同一の結果を生成するため，問題はない)．その結果，重複リストを生成する反復プログラムに比べて，必要なリスト量も少なくかつ計算スピードも速い (次節参照) 反復プログラムを得られた．

表 1　累積関数法適用前後の実行時間比較 [1000 回の合計, 単位秒]

	反復	再帰	修正後再帰
$n = 10$	0.109	0.609	4.172
$n = 20$	0.093	> 60	> 60
$n = 30$	0.141	> 60	> 60

4　比較実験及び関連研究

本節では，2 つの比較実験を示す．実験環境は，Windows XP, Pentium M 1.6GHz, 1GB RAM, clisp 2.8 である．一つ

表2　変換後の反復プログラムの比較 [100回の合計, 単位秒]

	累積関数法	従来法 (従属リスト)	従来法 (独立リスト)
$n = 100$	0.094	0.015	0.063
$n = 1000$	0.531	0.328	0.875
$n = 10000$	14.891	11.844	21.672

目の実験では変換前後の実行時間を示した (図1参照). 反復は累積関数法適用後の反復プログラム，再帰は元のハノイの塔のプログラムであり，修正後再帰は累積関数法を適用するために rplaca などを用いて修正された再帰プログラムである．再帰を除去し，かつ重複計算を除去しているため，累積関数法により劇的な高速化が達成できたことが確認された．二つ目は，変換後の反復プログラム同士の実行時間の比較である (表2参照)．これは，ハノイの塔のプログラムに対するプログラム変換の研究がある [7] ため，その論文記載の反復プログラムと本稿の累積関数法による反復プログラムの比較実験である．ただし，論文 [7] においては，プログラム変換の等価性の議論はなく，独立したリストの生成の有無を容易に変更できるため，重複計算に対応した複数のリストを生成する反復プログラム (従来法，独立リスト) の場合と，重複計算に対応した複数のリストを生成しない反復プログラム (従来法，従属リスト) の場合の二種について実験をした．実験結果 (表2) によると，累積関数法による反復プログラムは，プログラム (独立リスト) に比較すると高速に実行されるが，プログラム (従属リスト) に比較すると遅く実行された．前者の理由は，プログラム (独立リスト) は，各要素が独立のセルで表現されているため，メモリ使用量が多く計算時間が増加したからであり，後者の理由は，累積関数法による反復プログラムは rplaca などの操作に余分な時間が必要とされているからだと考えられる．

従来からの再帰除去法 [9] と比較すると，累積関数法の特徴は，変換後の反復プログラムがトップダウンに計算されることである．その実益は，後継関数に逆関数が存在しない関数に対しても，後継関数の計算量を増加させずに計算できることである．すなわち，従来の再帰除去法 [9] では，Double-Hailstorm 関数など後継関数に逆関数が存在しない場合に，再帰の深さをカウントし，入力に対してその再帰の深さ分後継関数を適用して計算する必要がある．この過程で，後継関数の計算量は，二乗の計算量になってしまう (あるいは，それらを記憶するために，再帰の深さ分のスペースが必要である)．他方，累積関数法はトップダウンに計算するため，後継関数の計算量を増加させず (あるいは，スペースを増加させず)，従来の再帰除去法より高速な (あるいはメモリ必要量の少ない) 反復プログラムに変換できる．これらの比較については，論文 [6] において述べた．ただし，ハノイの塔のプログラムでは，後継関数に逆関数が存在する (後継関数 $n-1$ に対して逆関数 $n+1$ が存在する) ため，累積関数法のこの点に関する利点は発揮されない．

また，本稿で対象とする非線形再帰プログラムに対しては，タプリング手法 [1, 3] が適用できるが，その後に再帰除去法を適用する段階においては，上記の再帰除去法の問題が発生する．タプリング手法 [1, 3] との比較については，論文 [4, 6] において述べた．

5　まとめ

本稿では，累積関数法の対象を拡張し，それに合わせて新たに累積関数を定義し，そして累積関数定理及びその証明を示した．本拡張の特徴は，複数の後継関数に対応できるよう，後継関数の定義域の視点から定義した点にある．そして，本稿の累積関数法について，ハノイの塔のプログラムを例として説明した．また，ハノイの塔のプログラムに関する従来の変換技術との比較実験（を行い議論した．今後の課題としては，補助関数及び前処理の自動化アルゴリズムの開発があげられる．

参考文献

[1] W. N. Chin, "Towards an automated tupling strategy", *In Proc. Conference on Partial Evaluation and Program Manipulation*, pp. 119–132 ACM Press, Copenhagen, June 1993.

[2] N. H. Cohen, "Eliminating Redundant Recursive Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 265–299, 1983.

[3] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano, Tupling Calculation Eliminates Multiple Data Traversals, *2nd ACM SIGPLAN International Conference on Functional Programming, Amsterdam, ACM Press.* pp. 164–175, June, 1997.

[4] 市川祐輔，小西善二郎，二村良彦：再帰除去におけるタプリング法の限界と累積関数法, 6F-2, ソフトウェア科学会大会論文集, 2002年9月.

[5] 市川祐輔，二村良彦，上田和紀：数式処理システム Mathematica 上における再帰除去システム, 第4回情報科学技術フォーラム (FIT2005), A-022, pp. 53–54, 2005.

[6] Y. Ichikawa, Z. Konishi, and Y. Futamura, "Recursion Removal from Recursive Programs with Only One Descent Function" *IEICE Trans. Inf. & Syst.*, vol. E88-D No.2, pp. 187–196, 2005.

[7] A. Pettorossi, "Towers of Hanoi Problems: Deriving Iterative Solutions by Program Transformations", *BIT,* Vol. 25, No. 2, pp. 327–334, 1985

[8] A. Pettorossi and M. Proietti, "Rules and Strategies for Transforming Functional and Logic Programs", *ACM Comput. Surv.*, Vol. 28, No. 2, pp. 360–414, 1996.

[9] M. S. Paterson and C. E. Hewitt, "Comparative Schematology", In Conference on Concurrent Systems and Parallel Computation Project MAC (Woods HOLE, MA), pp. 119–127, 1970.