プログラミング 18－21
(1998. 3. 24)

線形再帰プログラムからの再帰除去法の実現とその問題点

坂本巨樹† 川本史生† 小西善二郎† 二村良彦††
†早稲田大学大学院 理工学研究科
††早稲田大学 理工学部

再帰プログラムは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き呼び出しとスタック操作が必要である．それゆえ，与えられた再帰プログラムをスタックを使用せずしかも計算量も増加させずに，反復型プログラムに変換する方法が古くから研究されている．我々は，線形再帰プログラム(再帰呼び出しを実質的に 1 個所でしか行なわないプログラム)を計算量やスペース使用量を増やさずに反復型プログラムに変換する方法(再帰除去法)について先に報告した．その後，我々はその報告に基づき，線形再帰プログラムを能率のよい反復型プログラムに自動変換する再帰除去システムを Lisp を用いて実現した．本稿では，その実現法，適用例及び問題点について報告する．

Implementation of Recursion Removal System for Linear Recursive Programs and its Problems

Naoki Sakamoto† Fumio Kawamoto† Zenjiro Konishi† Yoshihiko Futamura††
†Graduate School of Science and Engineering, Waseda University
††School of Science and Engineering ,Waseda University

Recursive Programs are often easy to write and read, but in executing on computers, they require procedure calls and stack operations. Therefore, methods to transform given recursive programs into iterative ones without using stack and increasing amount of computation time have been studied. We have already proposed methods to transform a linear recursive program, which essentially include only one recursive procedure call in it. Later on, we implemented the system of transforming linear recursive programs into efficient iterative programs automatically, based on our proposed methods, in Lisp. This paper describes the way of implementation, applicable examples and the problems with the implementation.

1 はじめに

我々は，線形再帰プログラム(再帰呼び出しを実質的に1個所でしか行なわないプログラム)を計算量やスペース使用量を増やさずに反復型プログラムに変換する方法(再帰除去法)について先に報告した[4].その後我々は[4]に基づき，線形再帰プログラムを能率のよい反復型プログラムに自動変換する再帰除去システムを Lisp (Allegro Common Lisp) を用いて実現した．本稿では，その実現法及び適用例について報告する．本稿で扱う再帰プログラムは f(x)=if p(x) then b(x) else a(c(x),f(d(x)))(左線形再帰)や g(x)=if p(x) then b(x) else a(g(d(x)),c(x))(右線形再帰)の形式をしたものである(a,b,c,d,x 等に関する制限の詳細については文献[4]を参照されたい).この形を標準形と呼ぶことにする． また，本稿で扱うプログラムは線形再帰プログラムの中でも累積関数を有するものや，補助関数 a が擬似結合的[4]であるものに限る．そのような制

限をつけた上でも，実用上有用な多くの例題が自動変換できることを示す．

2 再帰除去の自動化

末尾再帰プログラムからの再帰除去は単純なので幾つかの言語処理系でも組み込まれている．しかし，末尾再帰よりも複雑な線形再帰プログラムについては，補助関数 a が結合性を有する場合[3]以外には，再帰除去自動化の報告を我々は聞いていない．ある程度複雑な再帰プログラムから再帰除去をする系統的方法は多数報告されている[1,2]．しかし，それらの方法は完全自動化できるほどには機械化されていない．我々は，補助関数が擬似結合的である場合および累積関数を有する場合[4]に，線形再帰プログラムから自動的に再帰除去を行なう実験を行なった．完全自動化を行なうためには，より強力なプログラム変換システム（例えば一般部分計算[6]）や数式処理システム[7]が必要である．将来は，一般部分計算器等[5,6,7]と相互に呼び合う形で使用することを想定している．しかし，本稿では，[4]で述べた再帰除去規則を単独で使用した場合の成功例について報告する．

3 補助関数 a が擬似結合的である場合

リスト処理でよく現れる関数 cons は結合的ではないが，擬似結合的である[4]．従って，[4]の再帰除去規則を利用した再帰除去プログラム recursion-removal を適用すれば，次のような形式で反復型プログラムが出力される．

```
> (recursion-removal
          '(if (p x)
            (b x)
            (cons (c x) (f (d x)))))
>(IF (P X)
     (B X)
   (LET* ((V (LIST (C X))) (W V) (U X))
     (DO ()
         (((LAMBDA (A) (P A)) (D U)))
       (SETF U (D U))
       (SETF W ((LAMBDA (A)
          (PROGN (RPLACD W A) A))
                (LIST (C U)))))
     ((LAMBDA (A) (RPLACD W (B A))) (D U))
     V))
```

3.1 自動変換可能な関数の例

```
(1) pair-merge(x)
/*リスト x の隣り合う要素をマージする:
例えば pair-merge([3 1 4 1 5 9 2])=[[1 3][1 4][5 9][2]]
    pair-merge([[1 3][1 4][5 9][2]])=[[1 1 3 4][2 5 9]]
(これを繰り返すことでマージソートが可能)*/
pair-merge(x) = if x=[] then []
 else if atom(car(x)) then
 { if cdr(x)=[] then [[car(x)]] else
 [mer-a(car(x),cadr(x)).pair-merge(cddr(x))] }
 else{ if cdr(x)=[] then x else
 [merge(car(x),cadr(x)).pair-merge(cddr(x))] }
 (2) distribute(x,y)
/*y を x の各要素に cons したリストを作る:
例えば distribute([b c d],a)=[[a.b][a.c][a.d]]
(例えば，集合の全ての部分集合を求めるときに現れる*/
distribute(x,y) = if x=[] then []
          else [ [y.car(x)].distribute(cdr(x).y)]
```

上記の pair-merge(x)のような 1 引数の場合だけでなく distribute(x,y)のように，2 引数（多引数）であっても 2 番目以降の引数が定数のような働きをする関数（c(x,y)で，y の中身を見ない．d(x,y),p(x,y)についても同様）ならば自動変換可能である．

つまり，この他にも append(x,k)や random-list(x,n)（n 以下の長さ x の乱数列を作る）なども自動変換可能となる．この他にも，補助関数 a が cons である場合は多数あるので，適用範囲は広い．（変換されたプログラムの性能評価は，[7]参照）

4 関数 f(x)が累積関数 h を有する場合

例えば補助関数 a が結合的であるならば，f(x)は累積関数 h(u,v)=a(c(u),v)を有する[4]．従って，[4]の再帰除去規則を利用した再帰除去プログラム recursion-removal を適用すれば，次のような形式で反復型プログラムが出力される．

```
> (recursion-removal
     '(if (p x)
          (b x)
          (aa (c x) (f (d x)))) 'aa)
>(IF (P X)
     (B X)
   (LET ((V (C X)) (U X))
     (DO ()
```

```
(((LAMBDA (A) (P A)) (D U)))
  (SETF U (D U))
  (SETF V (AA V (C U))))
((LAMBDA (A) (AA V (B A))) (D U)))
```

これにより例えば，a(x,y)=append(x,y)のとき，再帰除去可能である．何故ならば，append を補助関数とする線形再帰プログラムは左線形になることが多いので，特に副作用の心配が無い場合には，a(x,y)=aa(y,x)となる関数を新しく宣言してやることで右線形になる．適用できる例としては，reverse(x)(リスト x を反転させる)がある．また，再帰還元(再帰を減らすことで計算量を少なくする)を行なえるプログラム flatten(x), twist(x)などにも適用できる[5].

5 多引数関数への実現法

これまでは，実質的に 1 引数の場合を中心に再帰除去法の実現について述べてきた．ここでは，多引数関数の場合の変換方法に就いて述べる．例としては

```
merge(x,y)=if x=[] then y
          else if y=[] then x
 else if car(x)<car(y) then [car(x).merge(cdr(x),y)]
      else [car(y).merge(x,cdr(y))]
```

例えば merge([1 2 3],[2 3 4])=[1 2 2 3 3 4]
を使用する．f(x)=if p(x) then b(x) else a(c(x),f(d(x)))の形で定義され，1 引数の場合では自動的に再帰除去可能であったから merge を以下のように定義する．

```
(defun merge (x y)
    merge1((cons x y)))
(defun merge1 (x)
  (if (or (null (car x)) (null (cdr x)))
      (if (null (car x))
          (cdr x)
          (car x))
      (cons (if (< (car (car x)) (car (cdr x)))
                (car (car x))
                (car (cdr x)))
            (merg  (if (< (car (car x)) (car (cdr x)))
                       (cons (cdr (car x)) (cdr x))
                       (cons (car x) (cdr (cdr x))))
                   ))))
```

つまり，merge の 2 引数を(cons x y)として，見かけ上 1 引数の関数 merge1(x)を呼ぶ．merge1 は f(x)=if p(x) then b(x) else a(c(x),f(d(x)))の形で定義され，1 引数であるので，merge1 を recursion-removal にかけてやればよい．しかし，merge1 は再帰除去できるが余計な cons を導入したために最初のプログラムより速くはならない．そこで，merge1 を再帰除去した後で，最適化を考える．つまり，merge1(x)を再帰除去したプログラムから反復版プログラム mergeloop(x,y)を得ることを考える．まず変数の置き換えを行なう．

car(x)→ x ,cdr(x)→ y, car(u)→ux, cdr(u)→uy,
(u x) → (ux x) (uy y)
これは，関数 change による．
そのあとにプログラムを見ると

```
(((LAMBDA (A)
(OR (NULL (CAR A)) (NULL (CDR A)))
   (IF (< (CAR ux) (CAR uy))
          (CONS (CDR ux) uy)
          (CONS ux (CDR uy)))))
```

のように cons を行なった後で，car や cdr を行なうというムダが生じている．これを取り除くために，今まで lambda 式で変換していたところを lambda 式を使わずに変換すると

```
((OR       (NULL (CAR (IF (< (CAR ux) (CAR uy))
                          (CONS (CDR ux) uy)
                          (CONS ux (CDR uy)))))
           (NULL (CDR (IF (< (CAR ux) (CAR uy))
                          (CONS (CDR ux) uy)
                          (CONS ux (CDR uy)))))))
```

ここから，cons の後に car,cdr を行なっているところを関数 remove-cons により取り除くと

```
((OR (NULL (IF (< (CAR UX) (CAR UY)) (CDR UX) UX))
     (NULL (IF (< (CAR UX) (CAR UY)) UY (CDR UY)))))
```

となりこの部分の最適化は完了．
また

```
(SETF U        (IF (< (CAR ux) (CAR uy))
                   (CONS (CDR ux) uy)
                   (CONS ux (CDR uy))))
```

の部分は trans-setf により

```
(IF (< (CAR UX) (CAR UY))
        (SETF UX (CDR UX))
        (SETF UY (CDR UY)))
```

となる．
これらの手順をまとめると
(1) 与えられたプログラムを標準形のプログラムに cons を使って 1 引数で書き直す．

(2) 上記(1)のプログラムの本質的な部分を recursion-removal により，再帰除去を行なう．ただし，後で最適化を行なう部分は lambda 式を用いないこととする．

(3) recursion-removal により得られた 1 引数のプログラムから 2 引数の反復型プログラムを得る．

(3.1) 関数 change により変数の置き換えをする．

(3.2) 関数 remove-cons により cons の後の car,cdr を部分計算する(2ヶ所)．

(3.3) 関数 trans-setf により setf の部分の cons の後の car,cdr を部分計算する(1ヶ所)．

(4) 上記(3)の処理の終わったプログラムを

```
(defun floop (x y)
    /// 3 のプログラム///  )
```

とすることで完了．

上記(1)～(4)中で(1)はまだ自動化されていないがそれ以外の部分は自動化が実現されている．(2),(3)の部分を処理する関数を auto2rm とする．これにより，例えば vsum(x,y)(ベクトルの和を求める)や prod(x,y)(ベクトルの内積をもとめる)などの再帰除去が自動的に可能である．

6 プログラム開発での使用例

6.1 pair-merge sort

ここでは，マージソートプログラムの作成を例として，プログラム開発における我々の再帰除去システムの利用法について説明する．

```
(defun pair-merge (x)
  (if (or (null x) (null (cdr x)))
      (if (null x)
          nil
          (if (atom (car x))
              (list (list (car x)))
              x))
      (cons (if (atom (car x))
                (merge-a (car x) (cadr x))
                (merge (car x) (cadr x)))
            (pair-merge (cddr x)))))
/*リストをペアマージ化する(線形再帰)*/

(defun merge (x y)
  (cond ((null x)   y)
        ((null y)   x)
        ((<  (car x)  (car y))
                (cons (car x) (merge (cdr x) y)))
        (t    (cons (car y) (merge   x   (cdr y))))))
/*リスト x と y をマージする(線形再帰)*/
(defun merge-a (x y)
  (cond ((> x y) (list y x))
        (t   (list x y))))
/*アトム x,y をマージしたリストを作る(再帰なし)*/
(defun pair-merge-sort (x)
  (cond ((and (atom (car x)) (null (cdr x))) x)
        ((null (cdr x))   (car x))
        (t   (pair-merge-sort (pair-merge x)))))
/* リスト x をペアマージソートする(末尾再帰)*/
```

pair-merge は 1 引数の線形再帰であり，標準形で書かれているので，これを recursion-removal にかける．(3 節を参照)

```
>(recursion-removal
     /// pair-merge の中身 ///)
>(IF (OR (NULL X) (NULL (CDR X)))
     (IF (NULL X) NIL (IF (ATOM (CAR X)) (LIST (LIST
(CAR X))) X))
  (LET* ((V
          (LIST (IF (ATOM (CAR X))
                    (MER-A (CAR X) (CADR X))
                    (MERGELOOP (CAR X) (CADR X)))))
          (W V)
          (U X))
    (DO ()
        (((LAMBDA (A)
            (OR (NULL A)
                (NULL   (CDR A)))) (CDDR U)))
      (SETF U (CDDR U))
      (SETF W
      ((LAMBDA (A) (PROGN (RPLACD W A) A))
        (LIST (IF (ATOM (CAR U))
            (MER-A (CAR U) (CADR U))
            (MERGELOOP (CAR U) (CADR U)))))))
      ((LAMBDA (A)
         (RPLACD W
                 (IF (NULL A)
                     NIL
                   (IF (ATOM (CAR A))
                     (LIST (LIST (CAR A))) A))))
```

```
                (CDDR U))
        V))
(defun pair-mergeloop (x)
    ///出力結果///
    ) とすることで, pair-merge の再帰除去が完了.
```

merge は 2 引数であり, 再帰の使われ方が実質的に 2 番目の引数が定数として扱われていないので, 5 節の方法により再帰除去を行なう. まず, merge(x,y)を 5 節のプログラムに書き換える. merge1 の中身を auto2rm にかける.

```
>(auto2rm
      /// merge1 の中身/// )
>(IF (OR (NULL X) (NULL Y))
     (IF (NULL X) Y X)
   (LET*  ((V
   (LIST (IF (< (CAR X) (CAR Y)) (CAR X) (CAR Y))))
            (W V)
            (UX X)
            (UY Y))
     (DO ()
((OR (NULL (IF (< (CAR UX) (CAR UY)) (CDR UX) UX))
     (NULL (IF (< (CAR UX) (CAR UY)) UY (CDR UY)))))
       (IF (< (CAR UX) (CAR UY))
            (SETF UX (CDR UX))
            (SETF UY (CDR UY)))
       (SETF W
            ((LAMBDA (A) (PROGN (RPLACD W A) A))
(LIST (IF (< (CAR UX) (CAR UY)) (CAR UX) (CAR UY))))))
     (RPLACD W
      (IF (NULL (IF (< (CAR UX) (CAR UY)) (CDR UX) UX))
        (IF (< (CAR UX) (CAR UY)) UY (CDR UY))
        (IF (< (CAR UX) (CAR UY)) (CDR UX) UX)))
     V))
(defun mergeloop (x y)
    ///出力結果 ///  ) とすることで, merge の再帰除去が完了. 最後に
(defun pair-merge-sortloop (x)
   (cond ((and (atom (car x)) (null (cdr x))) x)
            ((null (cdr x))  (car x))
    (t   (pair-merge-sortloop (pair-mergeloop x)))))
```

pair-merge-sort 自体を書き直すことでプログラム全体を再帰から反復に変換できる.

6.2 冪乗の再帰プログラム exp(x,y)からのロシア人奴隷法の導出

x の y 乗のような冪乗計算の方法としてロシア人奴隷法が知られている. ここでは, 再帰除去によりロシア人奴隷法を導出する方法を示す.

```
(defun exp (x y)
   (if (= y 0)
         1
         (if (oddp y)
            (* x (exp (sqr x) (floor (/ y 2))))
            (exp   (sqr x) (/ y 2)))))
(defun sqr (x)
   (* x x))
```

/* exp(x,y)は x の y 乗を求める再帰プログラム*/

exp(x,y)は 2 引数の線形再帰であるが, その中の再帰の呼ばれ方が exp(sqr(x),y/2)となっているため, 実質的に 1 引数の扱いをすることができず, 5 節の方法による.

まず, exp を次のように定義し直す.

```
(defun exp (x y)
   (exp1 (cons x y)))
(defun exp1 (x)
   (if (= (cdr x) 0)
      1
      (* (if (oddp (cdr x))
            (car x)
            1)
         (exp1 (cons   (sqr (car x))
                  (floor (/ (cdr x) 2)))))))
```

そして, exp1 を auto2rm にかける.

```
>(auto2rm
   ///exp の中身///    '*)
>(IF (= Y 0) 1
   (LET ((V (IF (ODDP Y) X 1)) (UX X) (UY Y))
      (DO ()
        ((= (FLOOR (/ UY 2)) 0))
        (SETF UX (SQR UX))
        (SETF UY
          (FLOOR (/ UY 2)))
        (SETF V (* V (IF (ODDP UY) UX 1))))
      (* V 1))))
(defun exploop (x y)
  /// 出力の結果 ///) で再帰除去が完了.
```

この結果は，同じ判定を 2 回行なっており，さらに最適化をする必要がある．つまり，下線部の setf の方をループの判定条件の中に入れて，さらに(* v 1)を v にしてやればよい．これで，べき乗を求める再帰のプログラムから，ロシア人奴隷法が導けている．

7 再帰除去法の効果

表1 pair-merge-sort 及び exp の実行時間

		処理時間 (s)
pair-merge -sort	再帰版	132.140(49.190)
	反復版	62.812(17.590)
	差	69.328(31.600)
	改善率	0.475(0.357)
exp	再帰版	49.406(5.410)
	反復版	20.595(1.510)
	差	28.811(3.90)
	改善率	0.416(0.279)

但し，差＝(再帰版)－(反復版)とし，改善率＝(反復版)/(再帰版)とする．括弧内の数値はごみ集め時間及び改善率を表している．(使用機種 Sun Ultra2，処理系 Allegro Common Lisp)

pair-merge-sort では，長さ 60000 自由度 100000 の乱数列で 100 回繰り返した．exp では，exp(2,100000)を 10000 回繰り返した．いずれも 2 倍以上の性能向上が達成できた．さらに，メモリー使用量も改善が見られた．

ちなみに pair-merge-sort では，長さをさらに大きくすると再帰版は stack-overflow を起こして実行不能となる．exp では，第 1 引数を大きくすると，オーダーは変わらないのに変数が overflow を起こすので，第 1 引数を小さくした．

8 おわりに

線形再帰プログラムからの再帰除去について補助関数 a が限られた性質を有する場合について，自動変換が可能であることを紹介した．そのような制限の下でも多くの例が適用可能であることを示した．今後の課題としては，次のものがあげられる．

(1)一般の多引数関数についても紹介した方法が適用できるようにする．

(2)適用できる範囲を広げるため，今まで発見されている例については，自動再帰除去可能にする．

(3)累積関数や擬似結合性の実現関数を求めるアルゴリズムを発見する．

(1)については 2 引数の場合に cons を使って 1 引数に書き直した上で，再帰除去を行なうという実現方法を紹介したが，一般的に n 引数の場合の実現を考えると cons で 1 引数に書き直すよりも list で書き直したほうがより一般的にできる．しかし，方法は今回紹介した方法とほぼ同じであるから実現可能である．

また(3)の課題が解決されれば，再帰除去を自動化することは，かなり容易になる．しかし，累積関数や擬似結合性の実現関数を機械的に求めることはかなり難しいと思われる．そのためには，一般部分計算[6]や数式処理[8]を利用する必要がある．当面は，今回の報告のような個々の例を(1),(2)に関して増やしていくことによって，適用可能な再帰除去の範囲を広げ，かつ再帰除去に関する知見を深めていくことが求められる．それと同時に(3)の課題を進行していくことが理想的であると考えている．

さらに，今回はプログラマーがプログラム開発中にツールとして用いる方法を紹介したが，将来的にはプログラムを与えられ，その中から再帰除去可能なものを自動的に判定し，反復プログラムに変換されたプログラムを返すようなものとしたい．

参考文献

[1] Burstall, R.M. and Darlington, J. :
A Transformation System for Developing Recursive Programs, JACM, Vol.24, No.1, 1977, pp.44-67.

[2]Cohen, N.H. :Eliminating Redundant Recursive Calls, ACM TOPLAS., Vol.5, No.3, 1983, pp.265-299.

[3]Darlington, J. and Burstall, R.M. : A System which Automatically Improves Programs, Acta Informatica, Vol.6, No.1, 1976, pp.41-60.

[4]二村,大谷 :線形再帰プログラムからの再帰除去とその実際的効果, コンピュータソフトウェア, Vol. 15, 1998.

[5]二村, 大谷, 筧, 坂本, 小西:木再帰プログラムからの再帰還元, 情報処理学会プログラミング研究会
, PRO 18-20, 1998 年 3 月

[6] Futamura, Y., K. Nogi and A. Takano : Essence of generalized partial computation, Theoretical Computer Sciences 90, 1991, pp.61-79.

[7] 筧, 坂本, 二村:3E-08 再帰除去のゴミ回避効果, 情報処理学会第56回全国大会論文集3E-08, 1998年3月.

[8] 川本, 辰巳, 二村:3E-05 母関数を用いたプログラム変換, 情報処理学会第 56 回全国大会論文集3E-05, 1998 年 3 月.