プログラミング-言語・基礎・実践- 21－1
(1995. 3. 9)

線形再帰プログラムからの再帰除去法

二村良彦 大谷啓記

早稲田大学理工学部情報学科

〒169 東京都新宿区大久保3-4-1

e-mail: futamura@cfi.waseda.ac.jp, ootani@futamura.info.waseda.ac.jp

あらまし 線形再帰プログラムをスタックを使用しない反復型プログラムに変換するための新しい規則について報告する.補助関数が結合的である場合など,特殊な線形再帰プログラムの変換法は1970年中頃から良く知られていた.本稿で提案する方法はそれ等よりも更に適用範囲が広く,しかも副作用を持つプログラムにも適用可能なものである.本稿ではまず再帰プログラムについて定式化する.次に累積関数を定義し,累積関数を用いた再帰除去法及びその適用例を示す.最後に疑似結合性を定義し,疑似結合性を利用した再帰除去法及びその適用例を示す.

キーワード **再帰プログラム,再帰除去,プログラム変換**

Recursion Removal Techniques for Linear Recursive Programs

Yoshihiko Futamura *Hirofusa Otani*

Department of Information and Computer Science
School of Science and Engneering, Waseda University

Okubo 3-4-1, Shinjuku-ku, Tokyo, 169, Japan

e-mail: futamura@cfi.waseda.ac.jp, ootani@futamura.info.waseda.ac.jp

Abstract This paper reports new rules for transforming linear recursive programs into iterative ones. Recursion removal techniques for linear recursive programs, especially with associative auxiliary functions have been studied since mid 1970s. We propose new rules which are applicable to programs with side effects. We will present some practically interesting examples. This manuscript consists of 3 parts. (1) Notations for describing resursive programs, (2) Definitions and examples of the cumulative function rule and (3) Definitions and examples of the pseudo-associative function rule.

key words **recursive program, recursion removal, program transformation**

1 はじめに

再帰プログラムは書き易く読み易い場合が多い. しかし, 現代の多くの計算機で実行する際には手続き呼出しとスタック操作のためのオーバーヘッドが必要である. それ故, 与えられた再帰プログラムを, スタックを用いない反復プログラムに変換するいわゆる再帰除去法の研究が1970年中頃から行われてきた [3] [4][7].

同じ計算が, 再帰もスタックも使わない反復型プログラムに書ければ, 実行スピードは1桁近く速くなり, 使用メモリーは'再帰の深さ'分の1 (即ち再帰の深さが1000回ならば1/1000) になる. もちろんAckermann関数等の非原始帰納関数を反復型プログラムに変換することは原理的に不可能である [12].

一方, プログラム中に再帰呼出を本質的に1回しか含まないいわゆる再帰プログラムは線形スタックを用いた反復型プログラムに変換できることが知られている. そこで, 与えられた線形再帰プログラムがスタックを使わずに常数メモリで反復型プログラムに変換できるか否かの判別法とその変換法は長い間研究されてきた [1][2][5][6][13].

しかし過去の文献を調査したところ, 従来の線形再帰プログラムからの再帰除去法では, 適用条件が強すぎたり, 副作用が考慮されていないため, 現実のプログラムに対して適用不能であった.

本稿では能率の悪いプログラムを半自動的に改良する方法として, 普通のプログラマが書く多くの線形再帰プログラムを, スタックを使わない反復型プログラムに変換する (再帰除去する) 系統的方法について述べる [10][11]. 又その方法の強力さを示す為に, 典型的な現実的プログラムからの再帰除去例を示す.

なお以下においてアルゴリズムの記述にはISO PAD[DIS8631] 表現を用いる [8].

2 再帰の形式と再帰除去法

以下では再帰プログラムをスタックを使わない反復型プログラムに変換することを「再帰を取るまたは再帰除去する」と呼ぶ. また式の評価規則は最左最内規則及びcall-by-value semanticsを仮定する. 再帰プログラムは一般に図1の形式をしている.

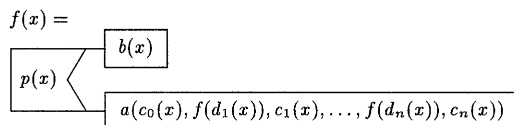

図1: 再帰プログラムの一般形

ここで, a, b, c_i 及び d_i を各々, 補助関数, 基底関数, 制御関数及び後継関数と呼ぶ. そしてそれらは f への再帰呼出し及びその他の自由変数を含まないものとする. 但し, x は変数ベクトル $\boldsymbol{x} = \langle x_1, \ldots, x_m \rangle$ でも良い. 同じ計算をする再帰プログラムについても補助関数 a と制御関数 c_i の決め方は一意ではない. 能率の良いプログラムを書くためには次の経験則がある [9].

部分計算の法則: $f(d_i(x))$ の値に依存しない値をできる限り抜き出して制御関数 $c_i(x)$ を作る.

$n=1$ のとき再帰プログラムは線形であると呼ぶ. 線形再帰プログラムには左線形, 右線形及び中線形があるが, 例えば右線形プログラムは図2の形式を有する.

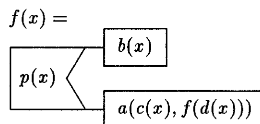

図2: 右線形再帰プログラム

そして左線形プログラムは図3の通りである.

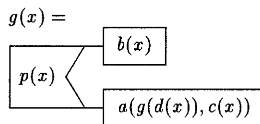

図3: 左線形再帰プログラム

副作用のないプログラムだけを扱う場合にはそれ等を区別する必要はなく, どの型でも例えば右線形にすることができる (証明は付録1参照). プログラムに副作用がある場合, 中線形は扱いにくいので, 本稿では左右の線形のみを扱う. 線形の内でも特に図4の f_1 と f_2 の形式のものを各々末尾再帰及び単純再帰と呼ぶ.

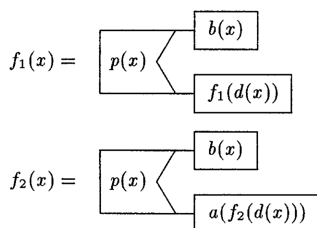

図4：末尾再帰プログラム及び単純再帰プログラム

この2つについては簡単な再帰除去規則が知られている[7]. また一般的な線形プログラムについても, 補助関数aが結合法則を満たす（即ち結合的）ならば, 容易に再帰除去が可能である[7]. その他の時は特殊な場合を除いて再帰除去規則は知られていない. 再帰呼出しを本質的に2箇所で行う（即ち$n \geq 2$）場合の再帰除去法は動的計画法の応用が知られているが, 一般性のあるのものは少ない[5].

3 累積関数を用いた再帰除去法

ここでは補助関数aが結合的である場合も含んだ, より適用範囲の広い再帰除去法について述べる.

3.1 累積関数の定義

定義1：$f(x)$は前述図2の右線形再帰プログラムとする. この時下記の性質を持つ関数$h(v,u)$をaに関するfの累積関数(cumulative function)と呼ぶ:
「$not(p(u))$ならば, 任意の式vに対して$a(v,f(u)) = a(h(v,u),f(d(u)))$. 但し, hはfへの再帰呼出し及びその他の自由変数を含んではならない.」

ここで, vが例えば2次元の変数ベクトルである場合には, $h(\langle v_1,v_2\rangle,u)$の代わりに$h(v_1,v_2,u)$と表記しても良いことにする. また$h$が例えば2次元の関数ベクトル$\langle h_1,h_2\rangle$の場合には$a(v_1,v_2,f(u)) = a(h_1(v_1,v_2,u),h_2(v_1,v_2,u),f(d(u)))$である. 3次元以上の場合や$u$についても同様にして扱う.

3.2 累積関数の例

例1: 補助関数aが結合的である場合

補助関数aが結合的即ち$a(x,a(y,z)) = a(a(x,y),z)$を満たす場合を考える. この時, $f(u) = a(c(x),f(d(u)))$より$a(v,f(u)) = a(v,a(c(u),f(d(u))))$である. 従って, $h(v,u) = a(v,c(u))$とすれば, hはfの累積関数である.

例2: 補助関数aが逆べき乗関数である場合

補助関数が逆べき乗関数即ち$a(x,y) = y^x$である場合を考える. $f(u) = a(c(x),f(d(u)))$より$a(v,f(u)) = a(v,a(c(u),f(d(u))))$である. ここで, 逆べき乗関数に関する指数法則$a(x,a(y,z)) = a(y*x,z)$から, $a(v,f(u)) = a(c(u)*v,f(d(u)))$となり, $h(v,u) = c(u)*v$を得る. 従って, hはfの累積関数である.

例3: プログラムが図5である場合

プログラムが図5である場合を考える. 但し, c_1とc_2の決め方は部分計算の法則による.

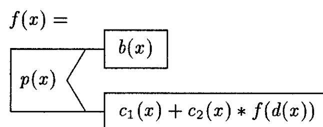

図5：2次元の累積関数ベクトルを持つプログラム$f(x)$

$a(v_1,v_2,u) = v_1 + v_2 * u$であるから, $a(v_1,v_2,f(u)) = a(h_1(v_1,v_2,u),h_2(v_1,v_2,u),f(d(u)))$を解いて, $h_1(v_1,v_2,u) = v_1+v_2*c_1(u)$及び$h_2(v_1,v_2,u) = v_2*c_2(u)$を得る. 従って, $\langle h_1,h_2\rangle$はfの累積関数である.

例4: プログラムが図6である場合

$f(x)$は$\lceil \frac{x}{2} \rceil$を引き算だけを用いて計算する関数, 即ち図6である場合を考える.

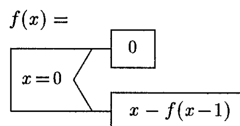

図6：$\lceil \frac{x}{2} \rceil$を計算するプログラム$f(x)$

$c_1(x) = x$, $c_2(x) = -1$及び$d(x) = -1$とすれば, 上述の例3より$h_1(v_1,v_2,u) = v_1+v_2*u$及び$h_2(v_1,v_2,u) = v_2*(-1)$を得る. 一方$f(x)$の数学的性質を使うこ

とにより,次のように1次元の累積関数が得られる: 今度は$a(x,y)=x-y$, $c(x)=x$及び$d(x)=x-1$とする. 今までと同様に$a(v,f(u))=a(h(v,u),f(d(u)))$を解いて, $h(v,u)=v-\lceil\frac{u}{2}\rceil+\lceil\frac{u-1}{2}\rceil = \text{if } odd(u) \text{ then } v-1 \text{ else } v$を得る. 従って, $\langle h_1,h_2\rangle$及びhはfの累積関数である.

3.3 累積関数を用いた再帰除去

累積関数の性質を利用することにより右線形再帰プログラムから再帰を除去することができる.

定理1: 右線形再帰プログラム$f(x)$が累積関数を持てば, それは図7の反復型プログラム$floop(x)$に変換することができる.

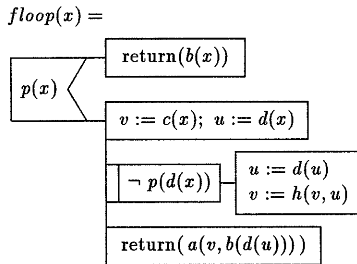

図7: 累積関数を利用して得た反復型プログラム$floop(x)$

c及びdが副作用を持つ場合にもfloopは正しく動作する.

証明) $d^n(x) = \overbrace{d(d(\ldots d}^{n}(x)\ldots))$ $(n>0)$と定義する. $\neg p(d(x))$ならばある$n=\min\{i\,|\,0\le i \land p(d^i(x))=true\}$が存在する. よって

$$\begin{aligned}
& a(c(x),f(d(x))) \\
=\ & a(c(x),a(c(d(x)),f(d^2(x))) \\
=\ & a(h(c(x),d(x)),f(d^2(x)))) \quad (\text{定義1より}) \\
=\ & a(h(c(x),d(x)),a(h(c(d(x)),d^2(x))),f(d^3(x)))) \\
=\ & a(h(h(c(x),d(x)),d^2(x)),f(d^3(x))) \\
=\ & \ldots \\
=\ & a(h(h(c(x),d(x)),d^2(x)),f(d^3(x))) \\
=\ & a(h(h(\ldots h(c(x),d(x))\ldots),d^{n-2}(x)),d^{n-1}(x)), \\
& \quad f(d^n(x))) \\
=\ & a(h(h(\ldots h(c(x),d(x))\ldots),d^{n-2}(x)),d^{n-1}(x)), \\
& \quad f(b(x)))
\end{aligned}$$

従ってその値は次の計算列により計算できる.

$v := c(x);\ u := x;$

$u := d(u);\ v := h(v,u);$

$\ldots$

$u := d(u);\ v := h(v,u);$

$v := a(v,b(d(u)));$

また, この計算列は$c(x)$及び$d(x)$に関する評価順序を保っている. 故に$c(x)$または$d(x)$に副作用を含んでいても計算結果を保つ.
(**証明終**)

具体的には, 例4の$f(x)$ (図6) から累積関数を利用して再帰除去し, 図8の反復型プログラム$floop(x)$が得られる.

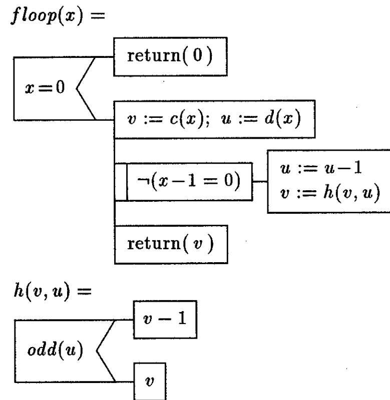

図8: 図6のプログラムの反復型プログラム$floop(x)$

以上, 右線形再帰プログラムに関する累積関数及びそれを利用した再帰除去法について示した. 同様に左線形再帰プログラムに関しては累積関数を以下のように定義すればよい.

定義2: $g(x)$は前述図3の左線形再帰プログラムとする. この時下記の性質を持つ関数$h(u,v)$をaに関するfの累積関数と呼ぶ:
「$not(p(u))$ならば, 任意の式vに対して$a(f(u),v)=a(g(d(u)),h(u,v))$. 但し, hはgへの再帰呼出し及びその他の自由変数を含んではならない.」

gを反復型にした$gloop$も$floop$と同様に定義できる. しかしこの場合にcまたはdに副作用

がある場合には$gloop$は正しく動作しない可能性がある. 何故ならば$c(x)$と$d^i(x)$の実行順序が入れ代わるからである.

証明) 定理1と同様にして確かめられる.
（証明終）

4 疑似結合性を利用した再帰除去法

ここではリスト処理プログラムの再帰除去に有効な方法について述べる. 以下ではLISPの関数を使うが, 読み書きのしやすさの関係からリストを角括弧を用いて表す. 例えば NIL= [], $(A\ B\ C) = [A\ B\ C]$, list(car(x))=[car(x)], cons(A,B)=[A.B] と表記する.

4.1 擬似結合性の定義

定義3：aを右線形再帰プログラム$f(x)$の補助関数とする. aに対して下記の性質を持つ関数a'が存在する時, aを疑似結合的(pseudo-associative)と呼ぶ:
「$a(x,y)$ =prog($a'(x,y),x$) かつ
$a(x,a(y,z))$=prog($a'(a'(x,y),z),x$).
但し, prog(u,v) はuとvを実行し, vの値を返す関数である.」

4.2 擬似結合性の例

例5: 補助関数がcons(x,y)である場合

補助関数がconsである場合を考える. この時, cons(x,y) =rplacd([x], y) より, $f(x)$を図8のように書き換えることができる. ここで $a(x,y) =$ rplacd(x,y) 及び $a'(x) =$ prog(rplacd(x,y), y) とすると, $a(x,a(y,z))$ = prog($a'(a'(x,y),z),x$) を満たす（詳細は付録2を参照）. 従ってaは疑似結合的である.

例6: プログラムが$append(x,k)$である場合

例5の具体例として, プログラムがリストを連結する関数$append$である場合を考える. $f(x) = append(x,k)$ とおく（但し, kは任意の常リスト）. 即ち$f(x)$は図9のように定義できる. この時, 例5と同様に cons(x,y) =rplacd([x], y) として, $f(x)$を図10のように書き換えることができる.

$f(x) =$

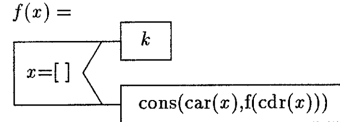

図9：$append(x)$ と等価なプログラム $f(x)$

$f(x) =$

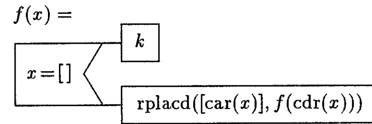

図10：再定義された$append(x)$

ここで $a(x,y) =$ rplacd(x,y), $c(x) =$ [car(x)], $d(x) =$ cdr(x) 及び $a'(x) =$ prog(rplacd(x,y), y) とすれば, 例5よりaは疑似結合的である.

例7: プログラムが$halve(x)$である場合

補助関数の引数としてベクトルを持つプログラムを考える. 図11の$halve(x)$は与えられたリストxの奇数番目の要素からなるリストと偶数番目の要素からなるリストをconsしたS式を返すプログラムである. 例えば, $halve$([1 2 3 4 5])=[[1 3 5] 2 4] となる.

$halve(x) =$

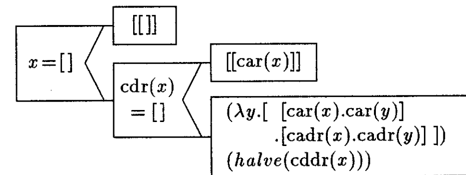

図11：リストを2分するプログラム $havle(x)$

ここで, $c_1(x)$ = [[[]].[[]]], $c_2(x) =$ car(x),
$c_3(x)$ =cadr(x), $d(x) =$ cddr(x),
（c_1, c_2, c_3の決め方は部分計算の法則による）
$a(v_1, v_2, v_3, u)$ = rd(
ra(v_1,rd(ra(car(v_1), v_2),car(u))),
rd(ra(cdr(v_1), v_3),cdr(u)))
とし,
$a'(v_1, v_2, v_3, u)$ = prog(rd'(rd'(
ra(v_1,rd(ra(car(v_1), v_2),car(u))),
ra(cdr(v_1), v_3)),cdr(u)), u)
とおく. 但し,

rd$(x,y)=$rplacd(x,y), ra$(x,y)=$rplaca(x,y),

rd'$(x,y)=$prog(rd$(x,y),y)$

とする. この時,

$a(v_1,v_2,v_3,u)=\mathrm{prog}(a'(v_1,v_2,v_3,u),v_1)$,

$a(w_1,w_2,w_3,a(v_1,v_2,v_3,u))=\mathrm{prog}($
$a'(a'(w_1,w_2,w_3,v_1),v_2,v_3,u),w_1\ )$

を満たすので, aは疑似結合的である.

4.3 擬似結合性を用いた再帰除去

擬似結合性の性質を利用することにより右線形再帰プログラムから再帰を除去することができる.

定理2: 右線形再帰プログラム$f(x)$を考える. 補助関数aが疑似結合的ならば, $f(x)$を図12の反復型プログラム$floop(x)$に変換することができる.

$floop(x)=$

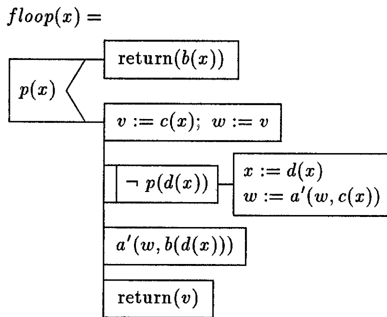

図12: 擬似結合性を利用して得た反復型プログラム$floop(x)$

c及びdが副作用を持つ場合にもfloopは正しく動作する.

証明) $\neg p(d(x))$ ならば ある $n=\min\{i\,|\,0\le i\wedge p(d^i(x))=true\}$が存在する. よって

$$\begin{aligned}
& a(c(x),f(d(x))) \\
=\ & a(c(x),a(c(d(x)),f(d^2(x)))) \\
=\ & \ldots \\
=\ & a(c(x),a(\ldots,a(c(d^{n-2}(x)),a(c(d^{n-1}(x)), \\
& \qquad b(d^n(x))))\ldots) \\
=\ & a(c(x),a(c(d(x)),a(\ldots \\
& \quad (\lambda v.prog(a'(a'(v,c(d^{n-1}(x))),b(d^n(x))),v)) \\
& \quad (c(d^{n-2}(x))\ldots)
\end{aligned}$$

(定義3より)

$$\begin{aligned}
=\ & \ldots \\
=\ & (\lambda v.prog(a'(a'(\ldots \\
& \quad (a'(v,c(d(x))),\ldots),c(d^{n-1}(x))),b(d^n(x))),v)) \\
& (c(x))
\end{aligned}$$

従ってその値は次の計算列により計算できる.

$v:=c(x);\ w:=v;$

$x:=d(x);\ w:=a'(w,c(x));$

$\ldots$

$x:=d(x);\ w:=a'(w,c(x));$

$w:=a'(w,b(d(x)));$

$return(v);$

また, この計算列は$c(x)$及び$d(x)$に関する評価順序を保っている. 故に$c(x)$または$d(x)$に副作用を含んでいても計算結果を保つ.
(証明終)

具体的には, 例7の$halve(x)$から疑似結合性を利用して再帰除去し, 図13の反復型プログラム$halveloop(x)$が得られる. 但し, $p(x)=(x=[]$ or cdr$(x)=[])$ かつ $b(x)=$ if $x=[]$ then [[]] else [[car(x)]] とする.

$halveloop(x)=$

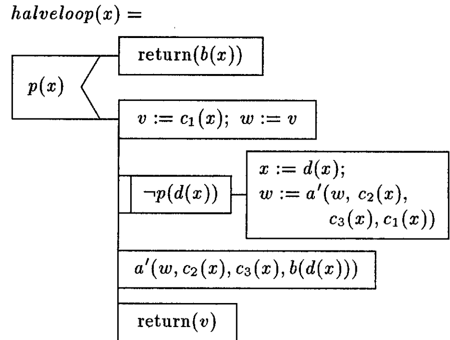

図13: プログラム$halve(x)$の反復型プログラム$halveloop(x)$

以上, 右線形再帰プログラムに関する擬似結合性及びそれを利用した再帰除去法について示した. 同様に左線形再帰プログラムに関しては擬似結合性を以下のように定義すればよい.

定義4：aを左線形再帰プログラム$g(x)$の補助関数とする．aに対して下記の性質を持つ関数a'が存在する時，aを疑似結合的と呼ぶ:

「$a(x,y) = \mathrm{prog}(a'(x,y), x)$ かつ
$a(a(x,y),z)) = \mathrm{prog}(a'(x, a'(y,z)), z)$．」

gを反復型にした$gloop$も$floop$と同様に定義できる．しかしこの場合にcまたはdに副作用がある場合には$gloop$は正しく動作しない可能性がある．何故ならば$c(x)$と$d^i(x)$の実行順序が入れ代わるからである．

証明) 定理2と同様にして確かめられる．
（証明終）

5 おわりに

数学的な計算及びリスト処理をする線形再帰プログラムから再帰を除去する新しい方法について報告した．また，方法の強力さを示す為に，典型的な現実的プログラムからの再帰除去例を示した．

今後の課題としては以下の3点が挙がる．

- 現実の線形再帰プログラムへの適用範囲の調査．
- tree recursion への拡張．
- LISPコンパイラへの組み込み．

また，累積関数を系統的に求める方法については，現在検討中であり，別途報告する予定である．

参考文献

[1] Arsac, J. and Kodratoff, Y. : Some Techniques for Recursion Removal from Recursive Functions, ACM TOPLAS., Vol.4, no.2, 1982, pp.295-322.

[2] Arsac, J. : Foundations of Programming, Academic Press, 1985.

[3] Boyer, R.S. and Moore, J.S. : Proving theorems about LISP functions, JACM, Vol.22, No.1, 1975, pp.129-144.

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

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

[6] Colussi, L : Recursion As an Effective Step in Program Development, ACM TOPLAS., Vol.6, No.1, 1984, pp.55-67.

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

[8] 二村良彦：プログラム技法-PADによる構造化プログラミング，オーム社，1984.

[9] Futamura, Y. : Partial Evaluation of Computation Process, Computers, Systems, and Controls 2, No.5, 1971.

[10] 二村，大谷：オペレータの疑似結合性を利用した再帰除去法，日本ソフトウエア科学会第11回大会，D4-1, 1994.

[11] 二村，他：プログラム変換方式，特願平6-263439, 1994年10月.

[12] Manna, Z. : Mathematical Theory of Computation, McGRAW-HILL, 1974.

[13] Paull, M.C. : Algorithm Design, John Wiley & Sons, 1988.

付録1: 左線形再帰プログラムと右線形再帰プログラム

第2節で制御関数$c(x)$及び後継関数$d(x)$に副作用が無い場合には左線形再帰プログラム（図3）と右線形再帰プログラム（図2）を区別する必要はない，と記した．これを示す．

証明) $c(x)$及び$d(x)$に副作用が無い場合，call-by-value semantics におけるλ変数を用いれば，図3は図14のように表すことができる．

図13は補助関数を$\lambda u,v.a(v,u)$とすることで右線形再帰プログラムとみなすことがで

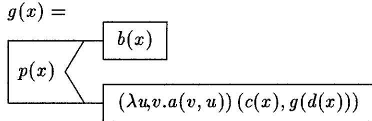

図 14 : 左線形再帰プログラムの右線形再帰化

きる. 逆も同様である. 従って, 左線形再帰プログラムと右線形再帰プログラムは区別する必要はない.
(証明終)

付録 2: cons の擬似結合性

第 4.2節の例 5 で示した, cons の擬似結合性について詳説する. cons はメモリ割り当ての手順を考えると $\mathrm{rplacd}([x], y)$ (=$\mathrm{rplacd}(\mathrm{list}(x), y)$) と等価である. よって, $\mathrm{cons}(x, y) = \mathrm{rplacd}([x], y)$ と置ける. ここで,

$a(x, y) = \mathrm{rplacd}([x], y)$,

$a'(x, y) = \mathrm{prog}(\mathrm{rplacd}(x, y), y)$

とする. この時,

$a(x, y) = \mathrm{prog}(a'(x, y), x)$ かつ

$a(x, a(y, z)) = \mathrm{prog}(a'(a'(x, y), z), x)$

を満たす. なぜなら,

$$
\begin{aligned}
& prog(a'(x, y), x) \\
= \ & \mathrm{prog}(\mathrm{prog}(\mathrm{rplacd}(x, y), y), x) \\
= \ & \mathrm{rplacd}(x, y) \\
= \ & a(x, y) \ .
\end{aligned}
$$

また, ポインタの張り替え順序に注意すると

$a(x, a(y, z))$

$= \mathrm{rplacd}(x, \mathrm{rplacd}(y, z))$

(y, x の順に cdr 部のポインタを張り替える)

= x → y → z

$= \mathrm{prog}(\mathrm{prog}(\mathrm{rplacd}(\mathrm{prog}(\mathrm{rplacd}(x, y), y), z), z), x)$

(x, y の順に cdr 部のポインタを張り替える)

$= \mathrm{prog}(a'(a'(x, y), z), x)$.

以上より, a は疑似結合的である.

付録 3: halve の補助関数に関する擬似結合性

第 4.2節の例 6 で示した, halve の補助関数 a に関する擬似結合性について詳説する. a 及び a' は以下の通りであった.

$$
\begin{aligned}
a(v_1, v_2, v_3, u) = \mathrm{rd}(& \\
& \mathrm{ra}(v_1, \mathrm{rd}(\mathrm{ra}(\mathrm{car}(v_1), v_2), \mathrm{car}(u))), \\
& \mathrm{rd}(\mathrm{ra}(\mathrm{cdr}(v_1), v_3), \mathrm{cdr}(u))) \\
a'(v_1, v_2, v_3, u) = \mathrm{prog}(\mathrm{rd'}(\mathrm{rd'}(& \\
& \mathrm{ra}(v_1, \mathrm{rd}(\mathrm{ra}(\mathrm{car}(v_1), v_2), \mathrm{car}(u))), \\
& \mathrm{ra}(\mathrm{cdr}(v_1), v_3)), \mathrm{cdr}(u)), u)
\end{aligned}
$$

但し, $\mathrm{rd}(x, y) = \mathrm{rplacd}(x, y)$,
$\mathrm{ra}(x, y) = \mathrm{rplaca}(x, y)$, $\mathrm{rd'}(x, y) = \mathrm{prog}(\mathrm{rd}(x, y), y)$
とする.

a' により a が擬似結合性を満たすならば以下の式が成り立つ.

$$
\begin{aligned}
& a(v_1, v_2, v_3, u) = \mathrm{prog}(a'(v_1, v_2, v_3, u), v_1), \\
& a(w_1, w_2, w_3, a(v_1, v_2, v_3, u)) = \mathrm{prog}(\\
& \qquad a'(a'(w_1, w_2, w_3, v_1), v_2, v_3, u), w_1).
\end{aligned}
$$

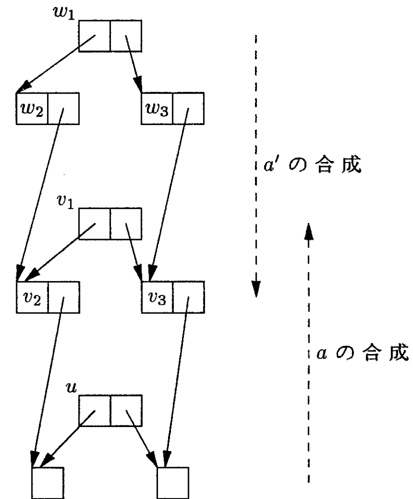

図 15 : halve に関する a 及び a' におけるメモリ割り当て手順

a のパラメタの評価順序を考えると, 図 15 のようなメモリ割り当てを図の下側から上方向に向かって行っている. 一方 a' は図の上側から下方向に向かってメモリを割り当てている. しかし, a 及び a' の評価後に割り当てられるメモリ内要素はどちらも同じである. 従って両者の返すリストの状態は一致する.
以上より, halve の補助関数 a は疑似結合的である.