プログラミング 18－20
(1998. 3. 24)

木再帰プログラムからの再帰還元

二村良彦† 大谷啓記†† 筧 一彦††† 坂本巨樹††† 小西善二郎†††
†早稲田大学理工学部情報学科 ††ＮＴＴコミュニケーションウェア(株)
†††早稲田大学理工学研究科

プログラムが再帰呼出しを実質的に2個所以上含むような非線形再帰の場合には，再帰除去は理論的に不可能な場合が多い．しかし，再帰を完全に除去できない場合でも，プログラムに含まれる再帰呼出しの箇所を実質的に減らすことが可能な場合がある．本稿では，再帰呼出しを実質的に2個所含む非線型再帰プログラム(木再帰プログラム)から，一方の再帰を実質的に除去する系統的方法およびその適用例と効果(成功例と失敗例)について報告する． 再帰プログラムから再帰呼出しの個数を実質的に減らすことを，我々は再帰還元(Recursion Reduction)と呼ぶ．

Recursion Reduction from Tree Recursive Programs

Yoshihiko Futamura†, Hirofusa Ohtani††, Kazuhiko Kakehi†††, Naoki Sakamoto††† and Zenjiro Konisi†††
†School of Science and Engineering, Waseda University, ††NTT Communicationware Corp.
†††Graduate School of Science and Engineering, Waseda University

Recursion reduction is a process to reduce the number of places of recursive calls in a non-linear recursive program, which includes more then one recursive call in it. While, recursion removal or recursion elimination is a process to eliminate all recursive calls from a program. Some times, recursion reduction is possible even when recursion removal is impossible. We have investigated the applicability and effectiveness of recursion reduction from a tree recursive program, which includes just two recursive calls in it. This paper describes two successful examples and one awkward one.

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が，計算機で実行する際には手続き呼出しとスタック操作が必要である．それ故与えられた再帰プログラムを,スタックを用いずしかも計算量を増加させないで反復プログラムに系統的方法で変換する，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[2,3,4]. 特に再帰呼出しを実質的に1個所しか含まない線形再帰プログラムからの再帰除去の研究は古い．線形再帰プログラムについても，再帰除去が理論的に不可能な場合は当然ある．しかし，プログラムが再帰呼出しを実質的に2個所以上含むような非線形再帰の場合には，再帰除去は理論的に不可能な場合が更に多く，その分野における再帰除去の研究はあまり行なわれていない．しかし実際には，再帰を完全に除去できない場合でも，プログラムに含まれる再帰呼出しの箇所を実質的に減らすことが可能な場合がある．本稿では，再帰プログラムから再帰呼出しを完全に除去するのではなくて，その個数を実質的に減らすことを，再帰還元(Recursion Reduction)と呼ぶ．

我々は再帰還元の効果を，再帰呼出しを実質的に2個所含む非線型再帰プログラム(木再帰プログラム)について調べた．そのために，木再帰プログラムに含まれる再帰呼出しの一方を無視することにより，線形再帰プログラムとみなし，それに線形再帰プログラムからの再帰除去規則[4]を適用した．これにより，一方の再帰呼出しを末尾再帰呼出しに出来る場合がある．末尾再帰は，実質的には再帰呼出しではない，即ちスタックが不要である，のでこの変換により再帰還元を行なうことが出来る．しかし，再帰還元は常に有効とは限らない．例えば，分割統治法の効果によりO(n log n)で計算できていた木再帰プログラムから再帰還元することにより，その性能を$O(n^2)$に劣化させる場合もありうる．本稿では，この再帰還元の適用範囲および効果について報告する．

2 準備

ここでは以下に述べるクラスに属する木再帰プログラムから再帰呼出しを実質的に一個所減らすために，文献[4]で述べられた線形再帰プログラムからの再帰除去に関する定理1の左線形再帰版を利用する．ただし本稿においては，式の評価規則は最左最内規則即ちcall-by-value semanticsを仮定する。またアルゴリズムの表現にはLispのような関数型言語を用いる．データー構造はLispのS式のようなものを用いるが，Lispの丸カッコの代りに角カッコを用いる．例えばLispのリスト(A B C)を[A B C],nilを[ ]のように表わす．

定理1:左線形再帰プログラム

g(x)= **if** p(x) **then** b(x) **else** a(g(d(x)),c(x))

が累積関数hを持つならば，g(x)は

gtail(x)= **if** p(x) **then** b(x) **else** gtail1(d(x),c(x))

と強等価である．ただし，

gtail1(u,v)= **if** p(u) **then** a(b(u),v) **else**

gtail1(d(u),h(u,v)).

ここでa,b,cおよびdをそれぞれ補助関数、基底関数、制御関数、および後継関数と呼ぶ。そしてそれらはfへの再帰呼出しおよびその他の自由変数を含まないものとする。ただしxは変数ベクトル$<x_1, ...,x_m>$でも良い。xが変数ベクトルの場合$f(<x_1, ...,x_m>)$の代りに$f(x_1, ...,x_m)$と書くこともある．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の決めかたは多様である。これらに関する詳細な議論は文献[4]を参照されたい．なお，hがgの累積関数であるとはa(g(u),v)= a(g(d(u)),h(u,v))が成立することである．

定理1の証明の概要:gが累積関数hを持つならば，a(g(u),v)=a(g(d(u)),h(u,v)). 従って，

g(x)=a(g(d(x)),c(x))=a(g(d^2(x)),h(d(x),c(x)))

=a(g(d^3(x)),h(d^2(x),h(d(x),c(x))))=...

=a(g(d^n(x)),h(d^{n-1}(x),h(...h(d^2(x),h(d(x),c(x))))))

=a(b(d^n(x)),h(d^{n-1}(x),h(...h(d^2(x),h(d(x),c(x))))))).

従って，g は末尾再帰関数 gtail と等価である．証明の詳細は文献[4]を参照されたい．□

上記の定理1は線形再帰プログラムからの再帰除去を行なう際に利用されたものである．ここではそれを再帰還元のために利用するのであるから，制御関数cがgへの再帰呼出しを含んでも良いことにする．このように関数cに関する条件を変えても定理1の変換は等価であることに注意されたい．

ここで，D_{gp}により関数gの定義域に含まれかつ述語pが成立しない要素の集合を表わせば，

a(g(d(x)),c(x)) =gtail1(d(x),c(x)) for $x \in D_{gp}$.

従ってu=d(x), v=c(x)とおけば，次の系1が得られる．ただし，$d(D_{gp})=\{d(u) | u \in d(D_{gp})\}$かつ$c(D_{gp})= \{c(u) | u \in c(D_{gp})\}$とする．

系1: a(g(u),v)=gtail1(u,v) for $u \in d(D_{gp})$ and $v \in c(D_{gp})$.

3 木再帰プログラムからの再帰還元

木再帰プログラムの一般形は大雑把には次の通りである:

g(x)=**if** p(x) **then** b(x) **else** a(g(d_1(x)),g(d_2(x)),c(x)). ただし，上記の線形再帰関数と同じく，a,b,cおよびdをそれぞれ補助関数、基底関数、制御関数、および後継関数と呼ぶ。そしてそれらはfへの再帰呼出しおよ

びその他の自由変数を含まないものとする。本稿の以下の部分では，木再帰プログラムのうちで次の2つのサブクラスについて考える．

3.1 補助関数が結合的な単純木再帰プログラム

次のような形式をした木再帰プログラムを単純木再帰と呼ぶ：

g(x)= **if** p(x) **then** b(x) **else** a(g(d_1(x)),g(d_2(x))).

c(x)が無いことがこの関数の特徴である．ここでは補助関数aが結合的な単純木再帰プログラムについて考える．

d(x)=d_1(x), c(x)=g(d_2(x))とおき，定理1を適用すると下記のgtail(x)が得られる．ただし，aが結合的なので定理1中のh(u,v)はa(c(u),v)である：

gtail(x)=**if** p(x) **then** b(x) **else** gtail1(d_1(x), g(d_2(x))).

ただし，gtail1(u,v)=

if p(u) **then** a(b(u),v) **else** gtail1(d(u),a(g(d_2(u)),v)).

系1より，上記gtail1中のuとvについてu∈d_1(Dg,p) and v∈g(d_2(Dg,p)) ならば，a(g(u),v)= gtail1(u,v)となりgが消去できる．従って次の定理2が成立する．

定理2：与えられた単純木再帰プログラムgにおいてその補助関数a が結合的であり，かつ上記gtail1中のuとvについて次の3条件が成立するものとする：

(1)u∈d_1($D_{g,p}$),

(2)v∈g(d_2($D_{g,p}$)),

(3)¬p(u) and d_2(u)∈d_1($D_{g,p}$).

この時gは下記のgtailに変換できる：

gtail(x)=**if** p(x) **then** b(x) **else** gtail1(d_1(x), gtail(d_2(x))).

ただしgtail1(u,v) = **if** p(u) **then** a(b(u),v) **else**

gtail1(d1(u), gtail1(d2(u),v)).

ここで，gtail1の中に現れる2つの再帰呼出しの一方は末尾再帰となり，実質的には再帰呼出しになっていないことに注意されたい．

例1：関数flattenへの適用

flatten(x)は，与えられたリストxを行きがけ順（in-order）に探索し，出現したアトムを出現順にリストする関数である．例えば，flatten([[A B] [[C D] [E F]] G]) =[A B C D E F G]：

flatten(x)=**if** atom(x) **then** (**if** null(x) nil **else** [x])

else append(flatten(car(x)),flatten(cdr(x))).

ここで，a(x,y)=append(x,y), b(x)=**if** null(x) **then** nil **else** [x], d_1(x)=car(x), d_2(x)=cdr(x), p(x)=atom(x), g(x)= flatten(x)とおけば，定理2の条件(1)～(3)が成立する．従って，関数flattenには定理2が適用可能であり，次のプログラムが得られる：

flattentail(x)=**if** atom(x) **then**

(**if** null(x) **then** nil **else** [x])

else flattentail1(car(x), flattentail1(cdr(x))).

flattentail1(u,v)= **if** atom(u)

then append(**if** null(u) **then** nil **else** [u], v)

else flattentail1(car(u), flattentail1(cdr(u),v).

xに含まれるconsセル(節と呼ぶ)の個数をnとすると，この変換によりプログラムの時間計算量は平均約O(n log n)から最悪O(n)に減少する．実際には，flatten関数の正確な計算量は，我々の調査の範囲では見つかっていない．しかし，かなり近い値としてO(n log n)が挙げられることを，後述のtwist関数の計算量と共に付録Aで議論する．flattentailの計算量が最悪O(n)であることは，それがリストを帰りがけ順になぞっているだけであることから明らかである．consの実行回数に関してはアトムの個数の約2倍にまで減少する．flattentail1中のappendは下記の簡約化により除去することができ，それによりconsの実行回数は更に半減し，丁度アトムの個数と同じになる：

flattentail1(u,v)= **if** atom(u)

then (**if** null(x) **then** v **else** cons(x, v))

else flattentail1(car(u), flattentail1(cdr(u),v).

例2：関数mergesortへの応用

下記のmergesortプログラムに対しても，上記のflatten同様，定理2が適用可能である（mergeが結合的であるから）．

mergesort(x)= **if** cdr(x)=nil **then** x **else**

(λ y. merge(mergesort(car(y)),mergesort(cdr(y))))

(halve(x)).

ここで, a(x,y)=merge(x,y), b(x)=x, p(x)=(cdr(x)=nil), d_1(x)=car(halve(x)),d_2(x)=cdr(halve(x))とおけば, 定理2より下記のmergesorttail(x)が得られる:

mergesorttail(x)= **if** cdr(x)=nil **then** x
else mergesorttail1(car(halve(x)),
mergesorttail(cdr(halve(x)))
=(λy. mergesorttail1(car(y),mergesorttail(cdr(y)))
(halve(x)).
mergesorttail1(u,v)= **if** cdr(u)=nil **then** merge(u,v)
else mergesorttail1(car(halve(u)),
mergesorttail1(cdr(halve(u)),v))
= **if** cdr(u)=nil **then** merge(u,v) **else** (λy.
mergesorttail1(car(y),mergesorttail1(cdr(y),v))
(halve(u)).

ただし, mergeとhalveは文献[4,6]の方法により再帰除去可能な下記の線形再帰プログラムである.

merge(x,y)=**if** x=nil **then** y **else** (**if** y=nil **then** x)
else(**if** car(x)<car(y) **then** cons(car(x),merge(cdr(x),y)))
else cons(car(y), merge(x, cdr(y)).

halve(x)= **if** x=nil **then** [nil] **else if** cdr(x)=nil **then** [[car(x)]] **else** (λy.[[car(x).car(y)].[cadr(x).cdr(y)]])
(halve(cddr(x))).

しかし, mergesorttail1が停止する時にmergeが実行され, そこでO(n)時間要する. 従って, 全体の実行時間は, 平均O(n log n)からO(n^2)に悪化する可能性がある. 何故ならば, この変換によりmergesortが挿入ソートに変わってしまったのである. このように, 分割統治法の効果によりO(n log n)で計算できていた木再帰プログラムから再帰除去することにより, その性能を劣化させる場合もありうる.

3.2 補助関数が結合的な半単純木再帰プログラム

次のような形式をした木再帰プログラムを半単純木再帰と呼ぶ:

g(x)= **if** p(x) **then** b(x) **else** a(g(d_1(x)),e(g(d_2(x)),x)).

c(x)=xであることがこの関数の特徴である. ここでは補助関数aが結合的な半単純木再帰プログラムについて考える. d(x)=d_1(x), c(x)=e(g(d_2(x)),x)とおき, 定理1を適用すると下記のgtail(x)が得られる:

gtail(x)= **if** p(x) **then** b(x) **else** gtail1(d_1(x), e(g(d_2(x)),x)).

ただし,

gtail1(u,v) =**if** p(u) **then** a(b(u),v) **else** gtail1(d(u),
a(e(g(d_2(u)),u),v)).

定理3: 与えられた半単純木再帰プログラムgにおいてその補助関数aが結合的でありかつ, 上記gtail1中のuとvについて次の4条件が成立するものとする:

(1)a(e(g(d_2(u)),u),v)=e'(a(g(d_2(u)),w),v)なるe'とwが存在する,

(2)u∈d_1(D_{gp}),

(3)v∈g(d_2(D_{gp})),

(4)¬p(u) and d_2(u)∈d_1(D_{gp}).

この時gは次のgtailに変換可能である:

gtail(x)= **if** p(x) **then** b(x) **else** gtail1(d_1(x),
e(gtail(d_2(x)),x)).

ただし, h(u,v)=a(c(u),v)より

gtail1(u,v)= **if** p(u) **then** a(b(u),v) **else** gtail1(d_1(u),
e'(gtail1(d_2(u),w),v)).

定理3の証明の概略: 系1より, 上記gtail1中のuとvについて条件2,3,4が成立するならば, 系1より a(g(d_2(u)),w)=gtail1(d_2(u),w)となり, (1)を利用してa(e(g(d_2(u)),u),v)中のgが消去できる. □

例3: 関数twistへの適用例

twist(x)は, 与えられたリストxの要素を全てのレベルに渡って逆転する関数である. 例えば, twist([[A B] [[C D] [E F]] G])=[G [[F E] [D C]] [B A]].

twist(x)= **if** atom(x) **then** x **else**
append (twist(cdr(x)), [twist(car(x))])
=**if** atom(x) **then** x **else**
append (twist(cdr(x)), e(twist(car(x)),x)).

ただし, e(x,y)=list(x)=[x]かつappendは, 第2引数がnilの時に第1引数をコピーしないように, 次のように定義する:

append(x,y)= **if** null(y) **then** x **else** (**if** null(x) **then**
y **else** cons(car(x), append(cdr(x),y)))).

ここでe'(x,y)=cons(x,y), b(x)=x, p(x)=atom(x), a(x,y)=append(x,y), d_1(x)=car(x), d_2(x)=cdr(x)と置けば, append(e(x,y),z)=append([x],z)=cons(x,z)=e'(x,z). 従って, append(e(g(d_2(u)),u),v)=e'(g(d_2(u)),v)=

e'(append(g(d_2(u)),nil),v)=e'(gtail1(d_2(u),nil),v).従って, w=nilであり定理3より, twistは下記のように再帰還元可能である.

twisttail(x)= **if** atom(x) **then** x

else twisttail1 (cdr(x),[twisttail(car(x))]).

ただし,

twisttail1(u,v)=**if** atom(u) **then** append(b(u),v)
else twisttail1 (cdr(u),
cons(twisttail1(car(u),nil),v))
= **if** atom(u) **then** (**if** v=nil **then** u **else** v)
else twisttail1 (cdr(u),
cons(twisttail1(car(u),nil),v)).

(appendの展開により得られる. uがアトムの時, vがnilでなければuがnilであることに注意).

appendの除去が行なわれたため時間計算量及びconsの実行回数が共に平均O(n log n)から最悪O(n)に減少している. ただしnは, xに含まれるconsセル(節)の個数である. 実際には, twist関数の正確な計算量は, 我々の調査の範囲では見つかっていない. しかし, かなり近い値としてO(n log n)が挙げられることを, 付録Aで議論する. twisttailの計算量が最悪O(n)であることは, それがリストを行きがけ順になぞっているだけであることから明らかである. 例1と同じく, 変換によりプログラムの性能が飛躍的に向上している. 実際に, テストデータとして全ての葉がアトムAであるような深さ17の完全2分木を用いた計測によれば, flattenの場合で約11～17倍, そしてtwistについては約3倍の性能向上が観測された(この時のnは2^{18}でありかつ出現するアトムAの個数は2^{17}である). 性能向上率が常に18(即ちlog n)倍になっていないのは利用した処理系の性能に依存する(補助関数のインライン化等の)問題があるからである[4]. また, 例2の方が性能向上率が悪いのは, 利用しているappend関数が例2の方が, 高性能であり, それを除去した際の効果が比較的少ないと言う理由による.

ちなみに, twist関数の非再帰版を作成する問題はLispの古い教科書[5]に現れる難易度の高い演習問題である. 文献[5]の解答と本稿で得られたtwisttailとは殆ど同じものであることを付録Bで議論する.

4 おわりに

補助関数が結合的である木再帰プログラムに線形再帰除去法を適用することにより, 下記の2点が実現できた:(1) 2つの再帰呼出しのうちの一方の末尾再帰化, (2) 補助関数の結合性の利用によるプログラム計算量の減少.

また, 上記(2)が常に可能とは限らないことも示した(例2). 本稿で提案した方法が有効な場合の判別および再帰除去法の自動化が今後の課題である.

参考文献

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D.: *Data Structures and Algorithms*, Addison-Wesley, 1983.

[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] Levin, M. and Berkeley, E.: LISP 2 PRIMER, TECH MEMO 2710/101/00, System Development Corporation, 2500 Colorado Avenue, Santa Monica, CA, 15, July, 1966.

[6] 坂本, 川本, 小西, 二村:線形再帰プログラムからの再帰除去実現とその問題点, 情報処理学会プログラミング研究会, PRO 18-21, 1998 年 3 月

[7] Sprugnori, R.:The Generation of Binary Trees as a

Numerical Problem, J.ACM, Vol.39, N0.2, 1992, 317-327.

付録A　twist関数の計算量概算

ここではn個の節を含むリストxに対するtwist(x)の計算量がおおむね平均O(n log n)であることを示す．これはクイックソートの計算量の証明[1]よりも更に大雑把なものである．正確な計算量の評価法は，我々の調査の範囲では見つかっていない．

先ず，節の数について説明する．例えば，x=[A B C]ならばn=3，　x=nil ならばn=0そしてx=[A [B C] D]ならばn=5である．xのcdrに含まれる節の数をkとする．この時，xのcarに含まれる節数はn-k-1である．そしてkについて次の仮定を設ける．「kは0以上n-1以下の値を等確率でとる」．この仮定は，明らかに正しくない[7]．しかし，大雑把な計算量を求める際に用いても，結果にはそれ程大きな影響は与えないものと考える．

twist(x)によって実行されるconsの平均回数をc(n)とすれば，c(n)の上界は下記の再帰不等式で表わすことが出来る：

(1) c(0)=0

(2) $c(n) \le \frac{1}{n}\sum_{k=0}^{n-1}(c(k)+c(n-k-1)+1+k)$

上の式(2)における最後の項kは，appendの計算量の上限を表わすことに注意されたい．この不等方程式をクイックソートの場合と同様にして解けばc(n)=O(n log n)が得られる．この解を得るための仮定が正しくないので，この解はあくまでも計算量の目安であると考えて戴きたい．

付録B　twisttail関数の非再帰版rev

本稿では，系統的方法によりtwist関数から再帰還元を行ない実質的に再帰呼出しを1個所でしか行なわない関数twisttailを導出した．一方，文献[5]においては，twistと同じ関数revを再帰呼出しを含まない方式で記述してある．これは，「再帰を含まない形でrevを記述せよ」という演習問題の模範解答である．

文献[5]におけるrevに含まれる変数名と合わせるために，twisttailの変数名及び，twisttailの呼出しの方法を下記のように変更する．定理3はaを一般的な結合的関数とした場合のものであるが，これをappendのように単位元(nil)を持つ補助関数に特殊化すれば，下記のものへの変換規則が得られる．

twisttail(x)=twisttail1(x,nil).

twisttail1(x,y)= **if** atom(x) **then** (**if** y=nil **then** x **else** y) **else** twisttail1 (cdr(x), cons(twisttail1(car(x),nil),y)).

一方，revの非再帰版は次のようなプログラムである(文献[5]163頁)．

```
FUNCTION REV(X); BEGIN SYMBOL Y,U,V;
A:  IF NULL X THEN
        (IF NULL U THEN RETURN Y ELSE GO B)
    U←X.U;
    V←Y.V;
    Y←NIL;
    X←CAR X;
IF NOT ATOM X THEN GO A ELSE Y←X;
B:  Y←Y.CAR V;
    X←CDAR U;
    U←CDR U;
    V←CDR V;
    GO A
END;
```

プログラム中，例えばY.CAR Vはcons(Y,car(V))の意味である．XとYは各々twisttail1のxとyに対応する．UとVは変数XとYの値を待避回復するためのスタックである．また，VはREVの中途結果を待避回復するためにも使われる．このプログラムは，良く調べてみると時間および空間計算量共にtwisttailと同じである．文献[1]等に述べられている簡単な再帰除去規則により，twisttailを上記のような反復プログラムに翻訳することは演習問題としても，易しい部類に属する．プログラムの読みやすさについては，revは難解である．twisttailが理解できているから，やっと理解できる程度と思われる．プログラム変換のメリットは，わかり易いプログラムから出発して，それだけを見ると難解な高性能プログラムに辿りつけることであることが理解できよう．