ソフトウェア基礎論24−3
(1988. 2. 12)

コンビネータとグラフ還元

Combinators and Graph Reductions

杉藤　芳雄
Yoshio SUGITO

電子技術総合研究所　ソフトウェア部　言語処理研究室
Language Processing Section, Computer Science Division, ELECTROTECHNICAL LABORATORY

あらまし　　組合せ論理の世界で演算子として登場するコンビネータは、その書き換え規則の連続的適用である還元という変換過程を通して、関数型言語プログラムの処理方式の一大勢力に寄与している。還元の実行形態としては共有構造を活用するグラフ還元が注目されている。本稿では、コンビネータおよび還元について概観したあと、組合せ論理の世界での還元系を用いて関数型言語プログラムをグラフ還元によりスタック等を用いずに“素朴に”実行する流儀を試みる。そして、この流儀を特に再帰プログラムに適用する場合を検討し、その方法と実験結果を示す。

Abstract　　Combinators, which behave as operators in the frame of Combinatory Logic, contribute to one of the most promising approaches for processing functional language programs by means of applying their rewriting rules successively to the programs. As for performing the transformation process, called a reduction, one of the most remarkable ones is a so-called graph reduction because of its merits on sharing substructures. In this report, after reviewing briefly combinators and reductions, we try to do a way executing "naively"(i.e. without using any stacking mechanism) a functional language program via graph reduction in the Combinatory Logic world with the aid of a reduction system written in a graph manipulation language. And we especially concentrate our attention on applying the way to recursive programs, and show its method and the results.

1. はじめに

コンビネータは、組合せ論理（Combinatory Logic）[1]の世界で“演算子”あるいは“関数”のように振舞うものである。コンビネータにそれが必要とする“引数”を施す効果は当該コンビネータに関する書き換え規則とみなすことができ、これらの書き換え規則を適用する系列は還元（reduction）と称する。

コンビネータの“引数”の取扱いは、ラムダ計算（λ-Calculus）における束縛（binding）規則のような繁雑なものを必要としない利点がある。それゆえ、コンビネータは、直観的には意味を理解しにくい記号列になりやすい欠点があるにもかかわらず、いわゆる副作用を引起こさない処理が前提の関数型言語[2]の処理方式の一つとして重視されている。

実際、関数型言語で記述されたプログラム（以下では関数型プログラムと記す）という原始コードをコンビネータ表現（即ち、組合せ論理の記号列）という目的コードに変換（“コンパイル”に相当）したものを、組合せ論理の世界での還元により評価（“実行”に相当）するという処理方式は、Turner[3]がコンビネータ表現のコード長の最適化に関する新しい書き換え規則を導入するという、いわばコロンブスの卵的な発見により関数型プログラムをほぼ実用的に処理できることを実証したことで脚光を浴びた。

この事実は、従来その理論的側面に殆どの関心を集めていた組合せ論理やコンビネータの世界に、実用的な計算機構という側面を設立するのに充分であった。

また、還元の実現形態は、記号列をどのようなデータ構造で表現するかに大きく依存する。記号列のデータ構造は、一般には木構造（あるいはリスト構造）を用いるが、部分構造の共有を許すグラフ構造を用いる場合には、いわゆるグラフ還元[4][5][6]が実現される。

本稿で取扱う内容は、表題の漠然とした大きさとは裏腹に、かなり些細な事柄へのこだわりに由来している。即ち、関数型プログラムの処理方式の一つとして知られるコンビネータ表現の還元方式に関して、グラフ還元を利用する場合の説明としてよく用いられる階乗プログラムのような再帰プログラム例では、再帰呼出しの関数名に向かうべき有向辺が根（root）（即ち、関数定義の総体）を指しているグラフでの最初の数ステップの図示だけで打切られ、肝心の再帰呼出しの時点での取扱いについて殆ど言及されないまま済ませてあること（例えば[7][8]）へのささやかな疑問である。

実際に後続の還元を図示しつつ追跡してみると、再帰呼出しに到達した時点で当惑させられることになろう。つまり、新たに呼出されるべき無傷のプログラム本体（ボディ）は存在せず、あるのは今までのグラフ還元で書き換えられた傷だらけの本体だけという惨状である。

もちろん、スタックを併用していく方式を考えれば済む話であると超然としている立場もあろうが、あくまでもコンビネータ表現のグラフ上で、スタック機構を一切用いずに、還元を“愚直”あるいは“素朴”に遂行していくことを追求するとどうなるか、とくに再帰プログラムの場合には、というのが本稿の立場である。

以降では、コンビネータおよびグラフ還元について概観したあと、関数型プログラムのコンビネータ表現に対して組合せ論理の還元系を用いて“素朴に”グラフ還元を実行する流儀を試みる。そして、この流儀を特に再帰プログラムにあてはめる場合を検討し、その方法と実験結果を示す。

2．コンビネータ

組合せ論理におけるコンビネータという演算子は、ラムダ計算におけるλ変換と同様の作用を、何とかして束縛変数を用いずに実現させることを目論んで導入されたものである。従って、束縛変数にまつわる代入などの煩瑣な規則に悩まされることはない反面、その記号列の意味が感性に訴えるものから遠いことは事実である。

まず組合せ項（combinatory term）は、次のように再帰的に定義されるものである。

(1) 各アトムは単一の組合せ項。（コンビネータは特定の単一アトムから成る組合せ項である。）
(2) ＡおよびＢがそれぞれ組合せ項ならば（ＡＢ）は単一の組合せ項。

左詰めの括弧対から成る組合せ項の括弧は省略する習慣があるので、（（（ＡＢ）Ｃ）Ｄ）はＡＢＣＤと略記される。

以下にコンビネータの代表的な例を挙げてみよう。ここで大文字の英字は“演算子”あるいは“関数”に相当するコンビネータという組合せ項であり、小文字の英字は“引数”に相当する組合せ項である。矢印の右辺はコンビネータを作用させた結果である。

```
S  x  y  z  ==>  x  z  (y  z)
K  x  y     ==>  x
I  x        ==>  x
B  x  y  z  ==>  x  (y  z)
C  x  y  z  ==>  x  z  y
W  x  y     ==>  x  y  y
Y  x        ==>  x  (Y  x)
```

上記のような各効果は当該コンビネータに関する書き換え規則とみなすことができる。組合せ論理（やラムダ計算）では、合法的な“形”（すなわち項）を書き換え規則により合法性を保ちつつ変換していく操作をできる限り続けていくことで最終的に得られる“形”（標準形 Normal Form）をもとの“形”に対する値（すなわち評価結果）とみなしているが、この変換過程のことを特に還元と称している。

コンビネータによる還元の例を以下に示す。ここで下線部は次に書き換え規則が適用される位置であり括弧対記号 () は冗長な括弧表現を除去する操作を意味している。

S (BBS) (KK) x y z	$\overset{S}{==>}$ (BBS) x ((KK) x) y z
$\overset{()}{==>}$ BBSx (KKx) y z	$\overset{K}{==>}$ BBSx (K) y z
$\overset{()}{==>}$ BBSxKyz	$\overset{B}{==>}$ B (Sx) Kyz
	$\overset{B}{==>}$ (Sx) (Ky) z
$\overset{()}{==>}$ Sx (Ky) z	$\overset{S}{==>}$ xz ((Ky) z)
$\overset{()}{==>}$ xz (Kyz)	$\overset{K}{==>}$ xz (y)
$\overset{()}{==>}$ xzy	

即ち、S（BBS）（KK）という複合コンビネータは第２、第３の“引数”を交換するので、コンビネータＣと同等の作用をすることになる。

組合せ論理は、ラムダ計算と同様に関数そのものを対象としている以上、いわゆる関数型言語との相性が良いことは言をまたない。それゆえ、関数型プログラムを組合せ論理で許される“形”のコード（コンビネータ表現）に変換（この過程を[3]では“コンパイル”と称する）したあと、このコードに然るべき還元を施すことにより値を求めることで関数型プログラムの実行とみなすような、関数型言語の処理系が考えられることになる。

ここで注意すべきは、単なるコンビネータに関する還元だけではプログラムに含まれる加減乗除算や比較／判定などの演算を“実行”できないことであり、還元形式だけでこれらの演算をも実行可能とするには当該演算の評価系に関する書き換え規則を追加しておけば良い。例えばごくありふれた関数から成る関数型言語での関数評価系に関する書き換え規則は次のようなものである（[9]）。

Plus	e_1	e_2		==> execute(e_1 + e_2)
Minus	e_1	e_2		==> execute(e_1 − e_2)
Times	e_1	e_2		==> execute(e_1 * e_2)
Divide	e_1	e_2		==> execute(e_1 ／ e_2)
Eq	e_1	e_2		==> If $e_1 = e_2$ Then “T” Else “F”
Neq	e_1	e_2		==> If $e_1 \neq e_2$ Then “T” Else “F”
Lt	e_1	e_2		==> If $e_1 < e_2$ Then “T” Else “F”
Gt	e_1	e_2		==> If $e_1 > e_2$ Then “T” Else “F”
Qcond	“T”	e_2	e_3	==> e_2
Qcond	“F”	e_2	e_3	==> e_3

関数型プログラムをコンビネータ・コードに“コンパイル”するための手順は、[3]に示されているように、以下の４段階である。ここでｘはアトムから成る単一の組合せ項、E_iは組合せ項である。

(1) 与えられた関数型プログラムを、２項を括弧対とする括弧表現にする。（Curry化）

例：　Def pred x = x - 1　　==>　Def pred x = ((Minus x) 1)

(2) 定義式の左辺の引数を“移項”してアブストラクション操作に備える。

例：　Def pred x = ((Minus x) 1)　==>　Def pred = [x]((Minus x) 1)

(3) 定義式の右辺に、以下のアブストラクション操作を可能な限り適用してコンビネータ・コードに変換する。

[x] (E_1 E_2)　==>　S ([x] E_1) ([x] E_2)

[x] x　==>　I

[x] y　==>　K y　（ｙは定数またはｘ以外の変数）

例：　Def pred = [x]((Minus x) 1) ==> S([x](Minus x))([x] 1) ==>
S(S([x] Minus)([x] x))([x] 1) ==> S(S(K Minus) I)(K 1)

(4) コンビネータ・コードを短縮する以下の最適化操作を行う。

① S (K E_1) (K E_2) ==> K (E_1 E_2)
② S (K E_1) I ==> E_1
③ S (K E_1) E_2 ==> B E_1 E_2 (①②が不可能のとき)
④ S E_1 (K E_2) ==> C E_1 E_2 (①②③が不可能のとき)

例： S(S(K Minus) I)(K 1) ==> S(Minus)(K 1) ==> C (Minus) 1 ==> C Minus 1

ここで Hughes[4] の例題 (λy. +y ((λz. *zz) (*yy))) 4
即ち、 f (y) ≡ y+g (y*y) かつ g (z) ≡ z*z のときの f (4)
を求めることを、関数型プログラムのコンビネータ・コードへの“コンパイル”および“実行”の過程で示すことにしよう。

まず、関数型プログラムの“コンパイル”を行ない、コンビネータ・コードαを得る。

```
      (λy. +y((λz. *zz) (*yy)))
   == ([y]((Plus y) ([z]((Times z) z)) ((Times y) y))))
   ==> ([y]((Plus y) (S Times I)       ((Times y) y))))
   ==> (S Plus (B (S Times I) (S Times I))) == α
```

次に、αに引数4を作用させて、コンビネータおよび関数に関する書き換え規則の適用という還元を行なうことで、関数型プログラムを“実行”する。

```
(α 4) == ( (S Plus (B (S Times I) (S Times I))) 4)
S, (), ()
======>        (Plus 4 ((B (S Times I) (S Times I)) 4))
B, (), ()
======>        (Plus 4 ((S Times I) ((S Times I) 4)))
S, (), ()
======>        (Plus 4 ((S Times I) (Times 4 (I 4))))
(), I
======>        (Plus 4 ((S Times I) (Times 4 4)))
Times, (), S
=============> (Plus 4 (Times 16 (I 16)))
(), I
======>        (Plus 4 (Times 16 16))
Times
======>        (Plus 4 256)
Plus
======>        260
```

3. グラフ還元

還元は、記号列を書き換え規則の適用で変換していく過程における書き換え規則の系列である。記号列表現に関するデータ構造、記号列への書き換え規則の適用の位置あるいは順序、等の多様性により、還元戦略がいくつか考えられる。

記号列表現のデータ構造は、部分構造の共有（sharing）を許さないで部分構造の写し（copy）を用いる立場と部分構造の共有を許すようなグラフ形式を採用する立場に大きく分類される。

命令（という項）が関数または引数の定義（という項）を利用する際には当該定義の写しを別個にとりその写しに関して評価していく方式であるストリング還元（String Reduction）は、前者の立場である。

一方、命令が定義を利用する際には当該定義への参照（reference）に関して評価していく方式であるグラフ還元（Graph Reduction）は,後者の立場である。

これら2種類の還元の用語および定義は[6]に依存している。それらの当否は別にして敢えてここに載せたのは、多くの文献が未定義のままグラフ還元を題材としていること（例えば[4]）への警鐘からである。また、ラムダ計算に関する大著[10]には、還元の章が存在しても“グラフ還元”は一切登場せず、わずかに“還元グラフ”という似て非なるものが現れるだけである。

以上の状況を踏まえて、本稿では、還元を施すべき対象の記号列のデータ構造表現に共有構造を許しているものを漠然と“グラフ還元”と呼ぶことにする。

還元方式の内でとりわけグラフ還元が注目されているのは、共有部分が一旦評価されると２度目以降のアクセスでは再評価が不要になること、一般に記憶容量が軽減されること、等が大きな理由であろう。

記号列への適用可能な書き換え規則候補（その左辺の形をリデックス（redex）と称する）が同一時点で複数個存在する場合の候補選択に関する問題は、関数項とその引数項から成る組における各項の評価順という問題に密接に関係している。

評価順は、最初に引数項を評価する値呼び［作用順還元（applicative order reduction）あるいは最内側計算規則］と最初に関数項を評価する名前呼び［正規順還元（normal order reduction）あるいは最外側計算規則］とに類別される。名前呼びは停止する解が存在する場合には常に解を見出すことが保証されているものの、引数項を毎回新たに評価し直すという効率の悪さがある。一方、値呼びは一旦評価した部分に関しては２回目以降では評価済みにするという効率の良さがあるものの、停止する解と停止しない状態とが併存する場合に停止する保証がない欠点がある。

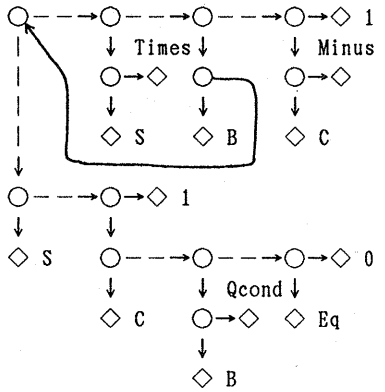
そこで、グラフ還元を名前呼びで実施（[3]では正規グラフ還元(normal graph reduction)と称する）すれば、効率は悪くならずに正しい解が得られるという両者の利点を活かす方式が実現することになる。

４．再帰プログラムの場合

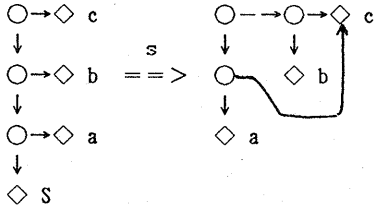
関数型プログラムが再帰形の場合のコンビネータ・コードの取扱いに関して、例えば次のような階乗プログラムを考えることにすれば、よくある解説は以下の様である。

```
Def fac n = if n=0 then 1 else n * fac(n-1)
          = (Qcond (Eq 0 n) 1 (Times n (fac (Minus n 1))))
          = (((Qcond ((Eq 0) n)) 1) ((Times n) (fac ((Minus n) 1)))
Def fac   = [n](((Qcond ((Eq 0) n)) 1) ((Times n) (fac ((Minus n) 1)))
          = (S (C (B Qcond (Eq 0)) 1) (S Times (B fac (C Minus 1))))
```

ここで右辺のコンビネータ・コードを木構造として表現するが、その際に右辺内の fac という再帰呼出しは木構造の頂点である根（即ち、fac の定義部全体）に有向辺を向けることで済ませられるので下図のようになる。このように共有構造をもたせると、木構造はグラフに変身することになる。ここで○は非終端点、◇は終端点（アトム点）をそれぞれ表している。



このようなグラフ上で還元を遂行するから“グラフ還元”であると言うのは自由だが、例えば次の様な書き換え操作でＳコンビネータを作用させるのと同様の要領で進めて行くと述べるだけで肝心の再帰呼出



しの時点での操作法を説明していないのは甚だ片手落ちである。確かに再帰呼出しに到達するまではコンビネータ（や関数）に関する書き換え規則を適用できてグラフを変換できる。ところが再帰呼出しに到達した時点で当惑させられる事態になる。根に向かう辺の先には新たに呼出されるべき無傷のプログラム本体は存在せず、あるのは今までのグラフ還元で書き換えられた傷だらけの本体だけという状況である。

もちろん、[3]ではスタックを併用してコンビネータや関数に関する還元を実際にグラフに施していく方式が述べられているが、再帰呼出しに関する共有構造を有するグラフ（即ち、再帰プログラム）の場合の処理の詳細が不明であることに超然としている立場も有り得よう。

しかし、スタックのような機構を用いずにあくまでもコンビネータ・コードに関するグラフ上だけで、還元を素朴に遂行していくことを追求する立場も有り得る。

後者の立場が、非再帰プログラムの場合には殆ど何も問題点がないことは既出の Hughes の例題からも容易に判断できるが、再帰プログラムの場合にはその取扱いに注意する価値があるので次節で検討する。

5．素朴な処理方式

前節では再帰プログラム fac のコンビネータ・コードへの“コンパイル”結果が次の様になることやそれに基づくグラフが示された。

```
Def fac  = (S (C (B Qcond (Eq 0)) 1)(S Times (B fac (C Minus 1))))
```

いま、このコンビネータ・コードを仮にβと呼ぶことにする。β内の再帰呼出し fac をアブストラクション操作して新たに得られるコンビネータ・コード（それをγとする）には fac が登場しないようにする。このγにYコンビネータを前置させれば再帰的コンビネータ・コード（再帰プログラム本体）が得られる。具体的には、次の様になる。

```
[fac]β = [fac]((S ((C ((B Qcond) (Eq 0))) 1)) ((S Times) ((B fac) ((C Minus) 1))))
       = (B (S (C (B Qcond (Eq 0)) 1) (B (S Times) (C B (C Minus 1)))))
       = γ
```

かくして再帰プログラムのコンビネータ・コードは (Y γ) となるので、このあと引数（階乗を求めたい整数値）を本プログラムに与えるには前記コードに後置させればよい。例えば fac(3) を求めるには、(Y γ 3) とすればよい。

実際に、このコンビネータ・コードを“素朴に”還元することで評価してみよう。即ち、スタック等は用いずにコンビネータ・コードのグラフの上だけで、コンビネータや関数に関する書き換え規則を適用していくことを試みる。その際、当研究室で開発したグラフ処理言語[11]により作成されたコンビネータ還元系および関数評価系並びに前記両者を統轄する処理系（以下では単に“還元系”と呼ぶ）を用いてコンビネータ・コードの還元を遂行する。（尚、関数型プログラムからコンビネータ・コードへの“コンパイル”も、同じく上記グラフ処理言語により作成された処理系を用いて実施される。）

この還元系で採用された戦略は以下に記すようなものである。

(1) コンビネータ還元系 を用いて指定回数以内で可能なだけ書き換え規則を（準）名前呼び方式で適用する。（但し、Yコンビネータに関する書き換え規則は、他のコンビネータに関する書き換え規則や冗長括弧の除去等の操作がすべて不可能な場合に初めて、その都度１回だけ試みられる。）
(2) 関数評価系を用いて可能な限り評価を行ない、最終的に単一のアトム点になれば、そのアトムの値が求めるものであり、実行を終了する。それ以外では (1)に戻る。
(3) 上記 (1) および (2) の書き換えや評価がいずれも全く施されない状態になれば、実行を打ち切る。

以上の戦略は、与えられたコンビネータ・コードが、正しく記述された（即ち、停止性が保証された）関数型プログラムを正しく“コンパイル”したものであることを前提にしているので、正しくないコンビネータ・コードが与えられると暴走する場合がある。

また、準名前呼びとは、リデックスの選択が必ずしも記号列全体での最左端ではなくて、深さ不定の括弧内での最左端として施される方式のことであり、これは既出グラフ処理言語のパタン・マッチング機能を凝らずに利用したことに由来している。（勿論、名前呼び方式を採用する方が望ましい。）

そして、Yコンビネータ適用に関する制約が設けられたのは、それが殆ど常に適用可能なリデックスなので恣意的に用いると暴走する危険性があるためである。実は(1)で“指定回数”を用意したのは、コンビネータ・コードを軽減するために(2)の段階を間欠的に利用する機会を与える意味合いがある他に、暴

走（らしい）状態を検出した場合に少しでも中断しやすくするためでもある。

以下に、階乗を求める関数型プログラムのコンビネータ・コードに引数３を与えて“素朴に”還元する過程を、操作系列を適宜中断してはその時点でのコンビネータ・コードを表示するという形式でまとめてある。コンビネータ・コードの記述を短縮するため、関数名は先頭１文字だけで表してある。（Cond を Qcond としておいたのは、コンビネータＣとの混同を避けるためである。）また、＊と＃で挟まれた部分は共有されている部分であることを意味する。操作系列での下線部は関数名であり、() は既出のように冗長括弧の除去操作、Copy は共有部分の存在が進行の妨げになる状態の際に実施される共有部分の複製操作である。（Copy や () の操作は採用するデータ構造により使用頻度や時機が変化するので、あくまでも参考にすぎない。）

```
(Y (B (S (C (B Q (E 0)) 1)) (B (S T) (C B (C M 1)))) 3)

    Y,
(  (*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)
(Y (*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)) 3)

    (),B,Copy,(),S,(),C,(),B,(),Eq,Qcond("F"),
((*(B (S T) (C B (C M 1)))# (Y (B(S(C(B Q(E 0))1)) *(B(S T)(C B(C M 1)))#))) 3)

    Copy,(),B,(),(),S,(),C,(),B,(),C,Minus,(),Y,
(T 3 ((*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)
   (Y (*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)) 2))

    (),B,Copy,(),S,(),C,(),B,(),Eq,Qcond("F"),
(T 3((*(B (S T)(C B(C M 1)))# (Y (B(S(C(B Q(E 0))1)) *(B(S T)(C B(C M1)))#)))2))

    Copy,(),B,(),C,(),(),S,(),B,(),C,Minus,(),Y,
(T 3 (T 2 ((*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)
        (Y (*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)) 1)))

    (),B,Copy,(),S,(),C,(),B,(),Eq,Qcond("F"),
(T 3(T 2((*(B(S T)(C B(C M 1)))# (Y(B(S(C(B Q(E 0))1)) *(B(S T)(C B(C M 1)))#1)))

    Copy,(),B,(),C,(),(),S,(),B,(),C,Minus,(),Y,
(T 3 (T 2 (T 1 ((*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)
            (Y (*B# *(S (C (B Q (E 0)) 1))# *(B (S T) (C B (C M 1)))#)) 0))))

    (),B,Copy,(),B,(),C,(),Copy,(),S,(),S,(),C,(),B,(),Eq,Qcond("T"),
(T 3 (T 2 (T 1 1)))

    Times(1*1),Times(2*1),Times(3*2).
6
```

以上により、３の階乗である６が確かに求められた。上記から、Ｙコンビネータが過不足なく再帰呼出しの時点でのみ登場し、自分の右側にある“無傷”のプログラム本体のコンビネータ・コードを複写して自分の左側に送ることが理解されよう。Ｙコンビネータが無傷のプログラム本体を複写して提供するという役割は、プログラム本体が偶々自分の右側に存在していればＹコンビネータの書き換え規則から当然に予想される現象であるが、他方Ｙコンビネータが再帰呼出しの時点でのみ過不足なく出現するという事実はコンビネータ・コードを一見しただけで容易に予想できるものではない。（このことはコンビネータを用いる場合の宿命的な欠点である。）ここでは、正しい関数型プログラムを正しくコンビネータ・コードにコンパイルして、許容される方法で実行（還元）したからだと、甚だ教条主義的に主張することによ

り、上記の還元戦略が不当なものではないことを指摘するに留めておく。

上記の還元戦略が妥当なものらしいことを補強する意味で、参考までにフィボナッチ関数という再帰プログラムを引数が２として還元してみよう。

簡単のためにフィボナッチ関数 fib の引数 n は非負整数であることを仮定する。階乗プログラムの場合と同様な手順でコンビネータ・コードにコンパイルしたものに、Ｙコンビネータを前置し、引数２を後置してから還元していく。

```
Def fib(n)≡ If 2)n Then n Else fib(n-2)+fib(n-1)
Def fib n = Qcond (Gt 2 n) n (Plus fib(Minus n 2) fib(Minus n 1))
          = (((Qcond((Gt 2) n)) n) ((Plus(fib((Minus n) 2))) (fib((Minus n) 1)))) = β
([fib](n]β))=(B (S (S (B Q(G 2)) I))(S(B S(B(B P)(C B(C M 2))))(C B(C M 1)))) = γ
fib(2) = (Y γ 2)
       = (Y  (B(S(S(B Q(G 2)) I)) (S(B S(B(B P)(C B(C M 2)))) (C B(C M 1))))  2)

    Y,(),B,Copy,(),S,(),S,I,(),(),B,(),Gt,
(Q "F" 2((*(S (B S (B (B P) (C B (C M 2)))) (C B (C M 1)))#       (Y (B(S(S(B Q(G 2)) I))
          *(S (B S (B (B P) (C B (C M 2)))) (C B (C M 1)))#))) 2))

    Qcond("F"),
((*(S (B S (B (B P) (C B (C M 2)))) (C B (C M 1)))#                (Y (B(S(S(B Q(G 2)) I))
  *(S (B S (B (B P) (C B (C M 2)))) (C B (C M 1)))#))) 2)

    Copy,(),S,(),B,(),S,(),B,(),(),B,(),C,(),B,(),(),C,(),Minus,C,(),B,(),C,(),
    Minus,Y,Copy,(),B,Copy,(),S,(),B,(),B,(),C,(),(),C,Copy,(),S,(),S,I,(),(),B,
    (),S,(),B,(),C,(),Minus,(),B,(),B,(),C,Minus,(),(),Gt,
(P        (*Y# *(B(S(S(B Q(G 2))I))(S(B S(B(B P)(C B(C M 2))))(C B(C M 1))))# 0)
(Q "T" 1(P(*Y# *(B(S(S(B Q(G 2))I))(S(B S(B(B P)(C B(C M 2))))(C B(C M 1))))#-1)
          (*Y# *(B(S(S(B Q(G 2))I))(S(B S(B(B P)(C B(C M 2))))(C B(C M 1))))# 0))))

    Qcond("T"),
(P        (Y    (B(S(S(B Q(G 2))I))(S(B S(B(B P)(C B(C M 2))))(C B(C M 1)))) 0) 1)

    Y,(),B,Copy,(),S,(),B,(),(),S,I,(),(),B,(),Gt,
(P (Q "T" 0 ((*(S (B S (B (B P) (C B (C M 2)))) (C B (C M 1)))#  (Y (B (S (S (B Q(G 2)) I))
             *(S (B S (B (B P) (C B (C M 2)))) (C B (C M 1)))#)))       (S (B Q(G 2)) I)
             ((S (B S (B (B P) (C B (C M 2)))) (C B (C M 1)))0))) 1)

    Qcond("T"),
(P 0 1)

    Plus(0+1).
 1
```

以上により、かなりのステップ数を要するものの fib(2) = fib(0) + fib(1) = 0 + 1 = 1 がやはり上記の還元戦略により確かに求められた。

6．おわりに

本稿で述べた事柄は既知の事実の集積であるかもしれないが、しかしながら序節で記した疑問に関して陽に説明された文献を寡聞にして知らないので、敢えて率直に問題提起を行ない、そしてグラフ還元を“素朴に”遂行する（らしい）方法を非形式的かつ初等的に紹介した。勿論、ここで紹介した方法が唯一最善のものであると主張するのではなく、とにかくスタック等を用いずともコンビネータ・コードのグラ

フ表現上での書き換え規則の適用だけで原理的には再帰プログラムをも還元できることを示したに過ぎない。何等かの参考になれば幸いである。

末筆ながら、本研究の機会を提供される棟上昭男ソフトウェア部長、および研究上の好ましい雰囲気を滲出して下さる当研究室をはじめとする関連各位に感謝したい。

参考文献

[1]例えば個人的好みでは J.R.Hindley,B.Lercher,J.P.Seldin: "Introduction to Combinatory Logic", London Mathematical Society Student Texts 1, Cambridge University Press, 1972
[2]P.Henderson: "Functional Programming--Application and Implementation",Prentice-Hall International,1980
[3]D.A.Turner: "New Implementation Techniques for Applicative Languages", Software-Practice and Experience, Vol.9, pp.31-49, 1979
[4]J.Hughes: "Graph Reduction with Super-Combinators", Oxford University Computing Laboratory Technical Monograph PRG-28, June 1982
[5]杉藤：“関数型言語とグラフ還元について”、情報処理学会ソフトウェア工学研究会資料87-SW-52-4 (1987.2.13)
[6]P.C.Treleaven, D.r.Brownbridge, R.P.Hopkins: "Data-Driven and Demand-Driven Computer Architecture", ACM Computing Surveys, Vol.14, No.1, PP.93-143
[7]疋田輝雄：“コンビネータによる言語処理系”、コンピュータソフトウェア、Vol.2, No.3,pp.27-35, 1985
[8]横内寛文：“ラムダ計算と高階プログラミング”、bit, Vol.19, No.14, pp.74-82, 1987
[9]杉藤：“組合せ子簡約系のグラフ処理言語による実現と再帰的プログラムの実行について”、情報処理学会ソフトウェア工学研究会資料 46-5(1986.2.6)
[10]H.P.Barendregt: "The Lambda Calculus--Its Syntax and Semantics", North Holland,1981
[11]Y.Sugito,Y.Mano: "Some applications using a graph manipulation language GML",Proc. of IEEE 1984 Workshop on Visual Languages, PP.53-58