ソフトウェア基礎論 28－7
（1989. 3. 10）

グラフ還元とデータ構造

Graph Reduction and Data Structure

杉藤　芳雄

Yoshio SUGITO

電子技術総合研究所　情報アーキテクチャ部　言語システム研究室

Computer Language Section, Computer Science Division, ELECTROTECHNICAL LABORATORY

あらまし　関数型言語の処理方式として有力視されるものに、関数型言語で記述したプログラムを組合せ論理の世界に写像して目的コードとし、それを組合せ論理のコンビネータ等に関する書き換え規則に基づく還元により評価実行するものがある。その際、還元の実行形態としては、記号列の変換をグラフ表現上での書き換えとみなす“グラフ還元”と称されるものが、いくつかの利点ゆえに特に注目される。グラフ還元がグラフを操作対象とする以上、グラフのデータ構造の差異がグラフ還元に及ぼす影響は無視できないものがあることが予想されるが、本報告ではその予想が事実であることを図解入りで示す。また、再帰プログラムの場合におけるグラフ還元とデータ構造の関係も検討する。

Abstract　As one of the most promising approach for processing functional language programs, there exists the one which transforms a functional language program into its corresponding Combinatory Logic expressions, and then which evaluates the expressions by means of a reduction based on rewriting rules of combinators in Combinatory Logic. In that case, as for the methods of executing reduction, "graph reduction", where transformation of strings is regarded as transformation of graphs, is especially worthy of notice because of its several merits. As graph reduction deales with graph itself, we can suppose that the difference among data structures of graph should have strong influence on the behavior of graph reduction. In this report, with the aid of illustrations we show that the supposition is true. And we also investigate the relations between graph reduction and data structure in case of recursive programs.

1．はじめに

組合せ論理（Combinatory Logic）[1]で“演算子”や“関数”の役目を果たすのはコンビネータである。必要な“引数”をコンビネータに後置した事前状態と、それらの“引数”にコンビネータを作用させた事後状態との関係は当該コンビネータの書き換え規則とみなすことができる。与えられた記号列に施される適用可能な書き換え規則の系列は還元（reduction）と称する。

コンビネータが“引数”を処理する方法は、ラムダ計算（λ-Calculus）での束縛（binding）規則のような煩瑣なものは一切必要とせずに、単に“引数”の配置を変更するだけの機械的操作にすぎないという見通しの良さを与える利点がある。コンビネータは“引数”の内容を全く考慮せずに配置を変更するだけである。

コンビネータによる操作をこれ以上続行できない状況に陥った場合、コンビネータ以外の演算子（例えば四則演算や比較演算あるいは制御構文）が存在するならば、それらを適用できる状況にはあるのが通常である。そして、それらの演算子を作用させて局面が打開され、そのまま最終的な評価値に到達するか、あるいは再びコンビネータによる操作が必要になるかのいずれかである。

コンビネータを含む記号列の還元は、意味を直観的には理解しにくい記号列になる傾向があるにもかかわらず、いわゆる副作用をもた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、関数型言語で記述したプログラム（以降、関数型プログラムと称する）という原始コードを組合せ論理の記号列（即ち、コンビネータを含む表現）という目的コードに写像（“コンパイル”に相当）したものに対して、コンビネータの書き換え規則を中心とする還元により評価（“実行”に相当）するという処理方式は、新しいコンビネータの書き換え規則をコンパイル過程に導入して目的コード長を短くするというTurner[3]の最適化技法の工夫により、ほぼ実用的に関数型プログラムを処理できることが実証されたので注目を浴びている。

例えば階乗計算の再帰的な関数型プログラムを組合せ論理表現に基づく目的コードとして評価実行する場合、プログラム入力として与えられた単一の入力引数（例えばｎの階乗を求めるときの自然数ｎ）から、階乗計算に必要な積、差、大小判定、再帰続行の可否を判定するif文、等の各関数を実行するのに必要な各入力引数を然るべき位置に然るべき時機に調達するのがコンビネータの役目であり、用意された入力引数から実行結果としての出力引数を創出するのが各関数の役目である。これらの出力引数を新たな入力引数としてコンビネータが活動を再開するのが常態である。

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。

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。即ち、記号列を何らかのデータ構造として表現しておけば、コンビネータを作用させる効果は部分データ構造の然るべき配置換えに相当している。このことは、データ構造の選択次第では配置換えの巧拙に大きく影響することをも意味する。

本稿で取扱う内容は、記号列の変換をグラフ表現上での書き換えとみなす“グラフ還元”と称されるものが、いくつかの利点ゆえに還元の実行形態として特に注目されることに因み、それのデータ構造との関連を調べることである。グラフ還元がグラフを操作対象とする以上、グラフのデータ構造の差異がグラフ還元に及ぼす影響は無視できないものがあることが充分に予想される。そこで以降では、実際に代表的な２種類のデータ構造を用意してグラフ還元を施すことで還元過程の差異を図解入りで比較検討し、更には再帰プログラムの場合についても考察する。

２．組合せ論理の還元

組合せ論理における演算子の役割を果たすコンビネータは、ラムダ計算におけるλ変換と同様の作用を、束縛変数を一切用いずに実現することを意図して導入されたものである。従って、束縛変数にまつわる煩瑣な規則に悩まされることはないという優れた点がある反面、その記号列の意味が直観的に理解しにくいという高価な代償を払っている。

先ず、組合せ論理の還元、即ち、コンビネータに関する書き換え規則を簡単に紹介する。

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アトムから成る単一の組合せ項。）

(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。 ■

左詰めの括弧対で囲まれた組合せ項の括弧対は省略する習慣があるので、（（（ＡＢ）Ｃ）（ＤＥ））　は　ＡＢＣ（ＤＥ）　と略記される。

以上の準備で、次に代表的なコンビネータの書き換え規則例を挙げてみよう。ここで大文字の英字はコンビネータという組合せ項であり、小文字の英字はコンビネータという演算子の“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。尚、各書き換え規則の左辺の形をリデックス (redex) と称する。

```
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
Y x       ==>  x (Y x)
```

組合せ論理では、合法的な“形”（すなわち組合せ項）から出発して上記のような書き換え規則により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。

組合せ論理における還元の例を以下に示す。ここで括弧対記号 () は冗長な括弧表現を除去する操作（即ち、上述の略記可能括弧対を除去する操作）を意味する。

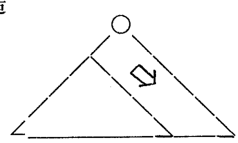
$$S(BBS)(KK)xyz \overset{S}{==>}$$
$$(BBS)x((KK)x)yz \overset{()}{==>} \overset{()}{==>}$$
$$BBSx(KKx)yz \overset{K}{==>}$$
$$BBSx(K)yz \overset{()}{==>} BBSxKyz \overset{B}{==>}$$
$$B(Sx)Kyz \overset{B}{==>} (Sx)(Ky)z \overset{()}{==>}$$
$$Sx(Ky)z \overset{S}{==>} xz((Ky)z) \overset{()}{==>}$$
$$xz(Kyz)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

即ち、Ｓ（ＢＢＳ）（ＫＫ）という複合コンビネータは還元により第２、第３の“引数”であるｙとｚを交換する標準形が得られるので、コンビネータＣと同等の作用をすることになる。

尚、上例の３行目の記号列ではＢやＫの書き換え規則のリデックスが存在しているが、このように書き換え規則を適用できる箇所が同時期に複数存在する場合も有り得る。還元は書き換え規則の適用系列として定義されるので、同じ式から出発する複数種類の還元が存在する事態の方が、むしろ普通である。この場合、ラムダ計算と同様に合流性が保証されていれば、複数種類の還元のいずれも同一の評価値（標準形）をもたらすことになる。

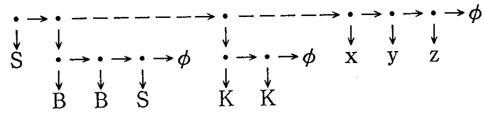
３．記号列のデータ構造と還元

３．１　自然なデータ構造

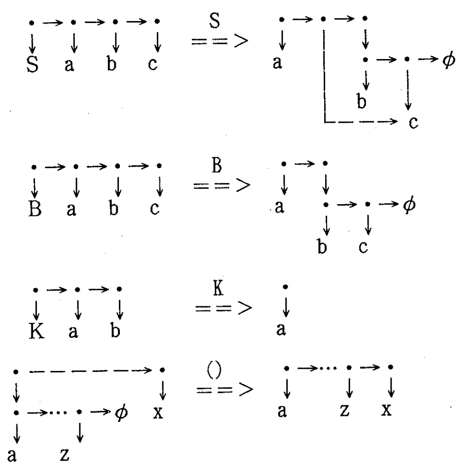


前章で示した還元の過程を忠実に反映させることを目論んで用意した記号列のデータ構造は、右図のようなものである。即ち、２進木 (binary tree) の根から右下端に向けて式（記号列）を左→右に配置していく方式であり、下降型とよぶことにする。下降型では、根から最右端沿いの経路にある点からの左分岐点がアトムのときは最上位項アトムであり、非アトムのときは括弧で囲まれた項の開始である。下降型は、左分岐の深さと項の括弧対の深さとが対応しているので式を読取りやすい利点があるものの、括弧対の中に存在する項の終りを意味するアトム（φ項）を常に必要とする欠点がある。

下降型による　Ｓ（ＢＢＳ）（ＫＫ）ｘｙｚ　の表現は次のようになる。



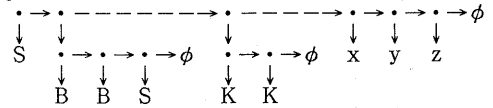
このデータ構造上で実際にコンビネータの書き換え規則を適用するには、やはりそのデータ構造向きの（正確にはそのデータ構造で表現された）書き換え規則が必要であり、そのために以下のものを用意する。



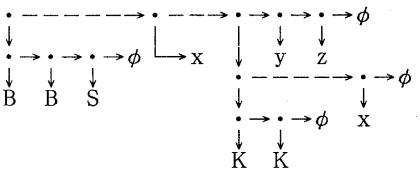
そして、前章の例題をこのデータ構造上で上記の書き換え規則により還元させると（これがまさにグラフ還元に他ならない）以下の還元例の様に所期の経過を辿りつつ

最終値に到達することが分る。厳密に言えば、上記の各書き換え規則のリデックス（即ち、左辺）内のコンビネータを左分岐とする点、即ちリデックスの左上隅の点は、項の開始（即ち、同一深さにある項の内で最左端のもの）を指示する点であることを想定している。例えば下記の還元過程の(5)でKの書き換え規則が適用されないのは、Kを左分岐とする点が項の開始位置にないからである。

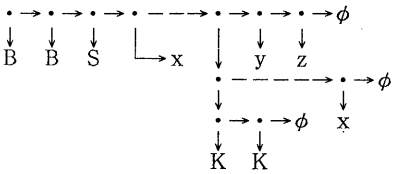
(0) S



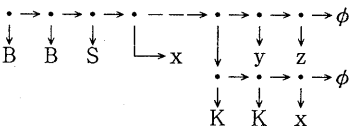
(1) ()



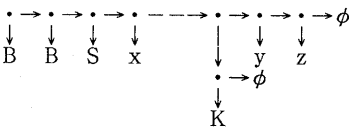
(2) ()



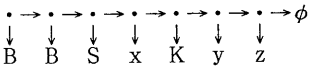
(3) K



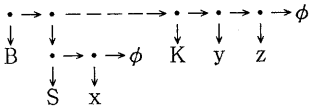
(4) ()



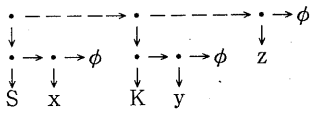
(5) B



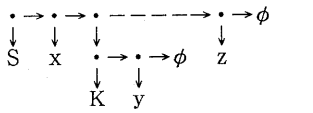
(6) B



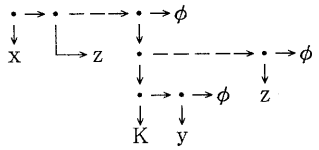
(7) ()



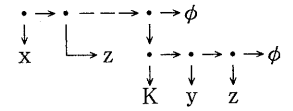
(8) S



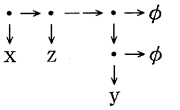
(9) ()



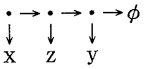
(10) K



(11) ()

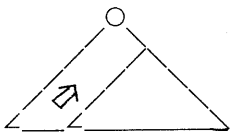


(12)

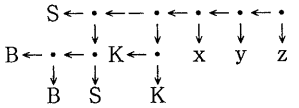


3．2　通常利用されるデータ構造

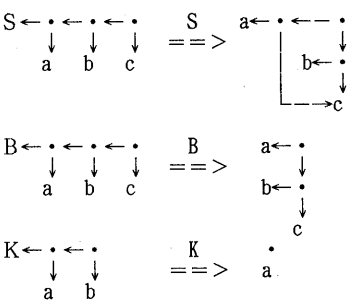
文献[3]等に登場するような、通常用いられる記号列のデータ構造は、前節のものとは異なっている。厳密に言えばこれは不正確な表現であり、両者とも2進木という骨格は全く同じで、単にその上に式を配置する方法が異なるだけである。しかし、その違いは決して無視できるものではないことが後述される。さて、本節で検討する記号列のデータ構造は、下図のように2進木の左下端から根に向けて式を左→右に配置していく方式であり、上昇型とよぶことにする。上昇型では、根から最左端沿いの経路にある点からの左分岐点がアトムのときは最上位の最左項であり、非アトムのときは最上位の項がその右分岐点にあることを意味する。当該右分岐点自身がアトムのときは最上位項アトムであり、非アトムのときは括弧で囲まれた項に対応する部分木の頂点である。上昇型は、密に式を充填できる（例えばφ項のようなものは不要）利点があるものの、式の読み取りに多少の慣れを必要とする欠点がある。



上昇型による S（BBS）（KK）xyz の表現は次のようになる。

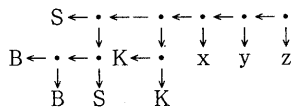


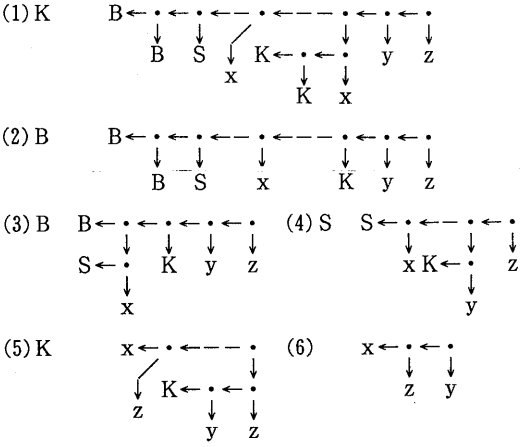
そして、このデータ構造に基づくコンビネータの書き換え規則は次のような表現になる。



やはり、前章の例題をこのデータ構造上で上記の書き換え規則により還元させると（これも勿論グラフ還元である）、一例として以下の経過を辿りつつ同じ結果を得る。

(0) S





3．3　データ構造と還元との関係

両方式のデータ構造の差異は、還元の最終結果（標準形）が異なるものになるというような致命的な事態を招くことには勿論至らないが、驚くべきことには（と少なくとも筆者には思えるが）いわゆる冗長括弧を除去するための操作が後者（上昇型）では一切出現していない。

手計算では必ず登場する筈の冗長括弧が還元の過程に全く出現しないことは、不自然さの傾向は免れないものの、還元の段数をかなり減少させるという利点（実際、上例では下降型が12段であるのに上昇型では半分の６段）が上昇型にはある。

上昇型のデータ構造がよく利用される理由は、式を密に充填できるという利点のほかに次のことが考えられる。式の表現において省略可能な括弧対というものは本質的には無用な存在の筈であり、無用な存在を含むような還元過程が機能的に優れた効果を生じる可能性は期待できそうにもないということである。

筆者は手計算との同一性を重視して長らく下降型のデータ構造を採用してきた。今回、下降型および上昇型のデータ構造がもたらす各還元を比較検討したところ、還元の中間結果では括弧対の有無に関する（それなりの）差異が見られるものの最終結果では（本質的な）差異は存在しないとみなせることが判明したので、以降では、紙数の制約上から上昇型のデータ構造のみで考察していくことにする。

4．関数型言語の組合せ論理へのコンパイル

組合せ論理が、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親和性が良いことは充分に予想されることである。実際、関数型プログラムを組合せ論理で許される“形”（コンビネータを含む組合せ項表現）のコードに変換（この変換を[3]では“コンパイル”と称する）したあと、このコードに２章で述べたような組合せ論理の還元を施すことにより（標準形という）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。

ここで注意すべきは、単なるコンビネータの書き換え規則に関する還元だけではプログラムに含まれる四則計算や比較や制御構文（例えばif文）等の演算を“実行”できないことである。そこで、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関数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、階乗計算の関数型プログラムに普通登場する演算の関数評価系に関する書き換え規則は次の様なものである。

```
minus   e1  e2   ==>  (e1 − e2)  の実行値
times   e1  e2   ==>  (e1 * e2)  の実行値
eq      e1  e2   ==>  e1=e2 のとき   “T”
                      e1≠e2 のとき   “F”
cond e1 e2 e3    ==>  e1=“T”のとき   e2
                      e1=“F”のとき   e3
```

関数型プログラムを組合せ論理コードに“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は単一の組合せ項アトム、Ｅiは組合せ項である。

(1) 与えられた関数型プログラムを、２項を単位に括弧対で囲むような表現にする。（Curry化）
例：　Def pred x = x − 1　　==>
　　Def pred x = ((minus x) 1)
(2) 定義式の左辺にある引数を“移項”して(3)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、以下の組合せ論理に関するアブストラクション操作を可能な限り適用して、組合せ論理コードに変換する。

```
[x] (E1 E2)  ==>S ([x] E1) ([x] E2)
[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) 組合せ論理コードを短縮する以下の最適化操作を行う。

```
①  S (K E1) (K E2)   ==>  K (E1 E2)
②  S (K E1) I        ==>  E1
③ (①②が不可能のとき)
   S (K E1) E2       ==>  B E1 E2
④ (①②③が不可能のとき)
   S E1 (K E2)       ==>  C E1 E2
```

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

尚、この目的コードを“実行”するには、入力データを後置して組合せ論理／関数評価系の還元を施せばよい。例えば、pred(5) を実行するには、pred の目的コードである（C minus 1）に入力引数の５を後置させたもの(C minus 1)5 を還元すればよい。これは次のような経過で値が得られる。
(C minus 1)5 ==> C minus 1 5 ==> minus 5 1 ==> 4

5．再帰プログラムとグラフ還元

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。

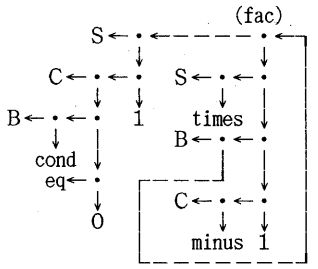
還元方式の内でとりわけグラフ還元が注目されているのは、共有部分が一旦評価されると２度目以降のアクセスでは再評価が不要になるという時間面の利点、一般に共有構造のために記憶容量が軽減されるという空間面の利点、等が大きな理由であろう。

関数型プログラムが再帰形の場合、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、その目的コードのデータ構造化に関して、よく見受けられる議論は[3]を踏襲した以下の様なものである。

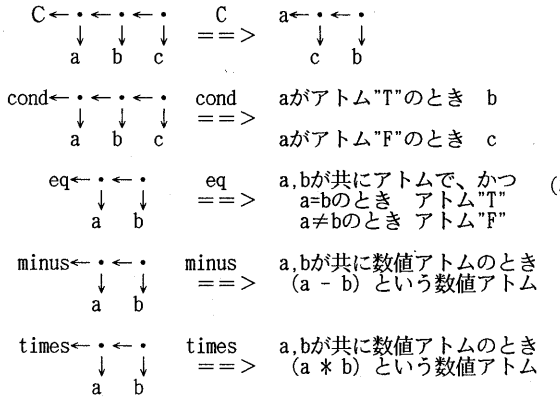
先ず、与えられた階乗プログラムをCurry化したあと、組合せ論理コードにコンパイルする。

```
Def fac n = if n=0 then 1 else n * fac(n-1)
= (cond (eq 0 n) 1 (times n (fac (minus n 1))))
= (((cond((eq 0)n))1)((times n)(fac((minus n)1)))
Def fac   = [n](((cond ((eq 0) n)) 1) ((times n)
                  (fac ((minus n) 1)))
          = (S (C (B cond (eq 0)) 1) (S times
                  (B fac (C minus 1))))
```

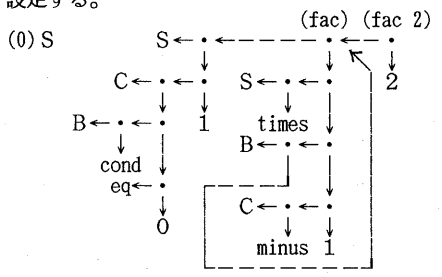
ここで右辺の組合せ論理コードを２進木構造として表現するが、その際に右辺内の fac という再帰呼出しは木構造の頂点である根（即ち、fac の定義部全体）に有向辺を向けることで表現できる筈なので下図のようになる。このように共有構造をもたせると、木構造は木からグラフに変身することになる。



このデータ構造で、例えば fac(2) を評価するためには３．２節で利用したＳやＢの書き換え規則の他に、次の様な書き換え規則を用意しておく。



さて、fac(2) に相当するデータ構造 (fac 2) を次の様に設定する。



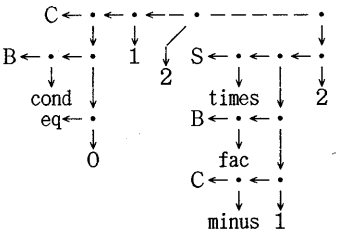
このあと、３．２節に登場したようなＳに関する書き換え規則等の例を示しながら、このような要領で上図に対して各種の書き換え規則を適用して還元を施せば所望の fac(2) に相当する値が得られる…という説明で終えている。

ところが、上図の最初の段階でＳに関する書き換え規則が適用可能なのでそれを遂行しようとすると、早くも共有構造に破綻をきたすことになる。即ち、fac 定義の頂点への共有辺が文字通り宙に浮く。

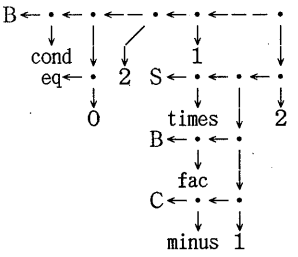
そこで、何らかの手段でfac 定義への辺が常にfac 定義本体の構造を保持していると仮定して、図上では単に fac を指示する矢印として表現することにすれば、以後の還元が順調に作動していくことは以下の図解に示される通りである。

このfac 定義への共有辺方式を文献[3] で図示している Turner が上記の問題点をいかに解決あるいは回避しているかは、[3] を読む限り不明瞭である。共有辺の活用というグラフ還元の利点が、再帰プログラムの場合の再帰呼出しの状況でも生じれば申し分ないが、残念ながら共有構造の写し（コピー）を利用せざるを得ないというのが筆者の現時点での結論である。

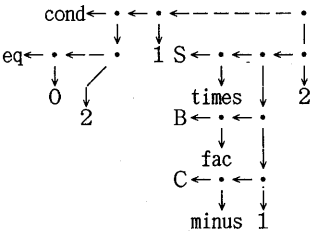
(1) C



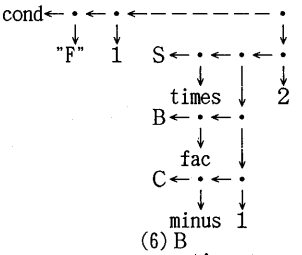
(2) B



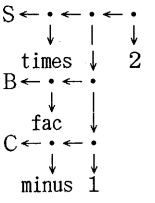
(3)eq



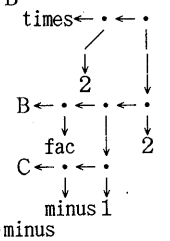
(4)cond



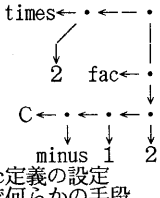
(5) S



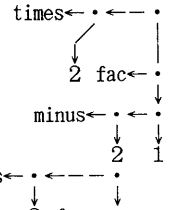
(6) B



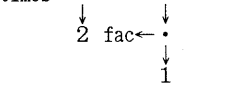
(7) C



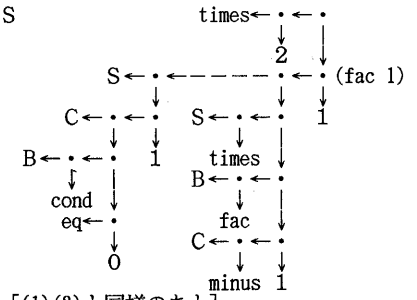
(8)minus



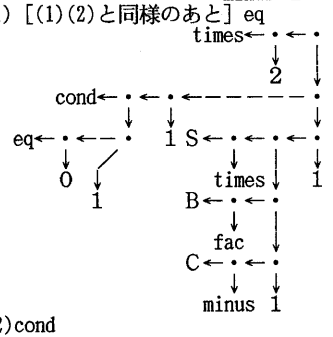
(9)fac定義の設定
ここで何らかの手段によりfac 定義本体を配置できるものと仮定する。



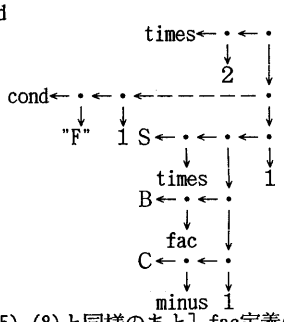
(10) S



(11) [(1)(2)と同様のあと] eq

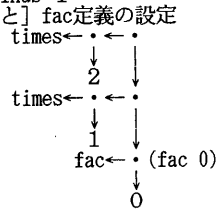


(12)cond

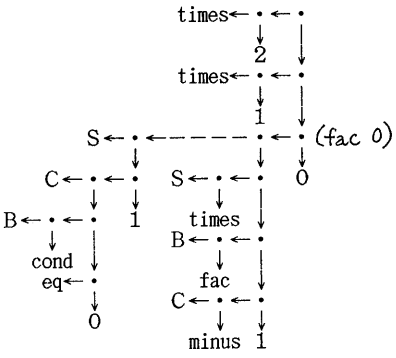


(13) [(5)-(8)と同様のあと] fac定義の設定

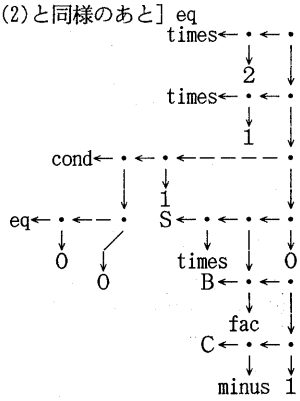
ここで何らかの手段によりfac 定義本体を配置できるものと仮定する。



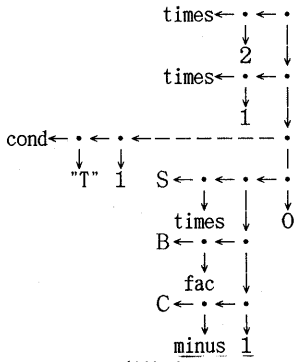
(14) S



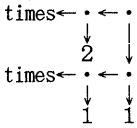
(15) [(1)(2)と同様のあと] eq



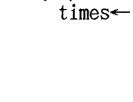
(16)cond



(17)times



(18)times



(19)

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は変りがない。

しかし、スタックのような機構を用いずにあくまでも組合せ論理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再帰プログラムの場合にも踏襲する方法として、Yコンビネータの書き換え規則を利用することが[6]で述べられている。そこで次章でその方法による再帰プログラムの還元の図解を簡単に紹介する。

6．素朴な処理方式

前章では再帰プログラム fac の組合せ論理コードへの“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 cond (eq 0)) 1)(S times
             (B fac (C minus 1))))
```

いま、この組合せ論理コードを仮に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することにより、新たに得られる組合せ論理コード（それをγとする）には fac が登場しないようにする。

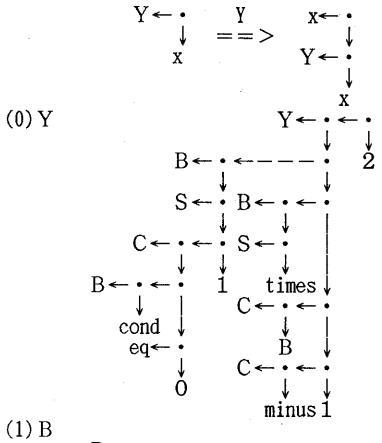
```
[fac]β = [fac]((S ((C ((B cond) (eq 0))) 1))
                 ((S times) ((B fac) ((C minus) 1))))
       = (B (S (C (B cond (eq 0)) 1) (B (S times)
           (C B (C minus 1))))
       = γ
```

このγにYコンビネータを前置させれば再帰的な組合せ

論理コード（再帰プログラム本体）が得られる。
かくして再帰プログラムの組合せ論理コードは(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(2) を求めるには、(Y γ 2) とすればよい。
あとは以下に記すような戦略で還元を施していく。

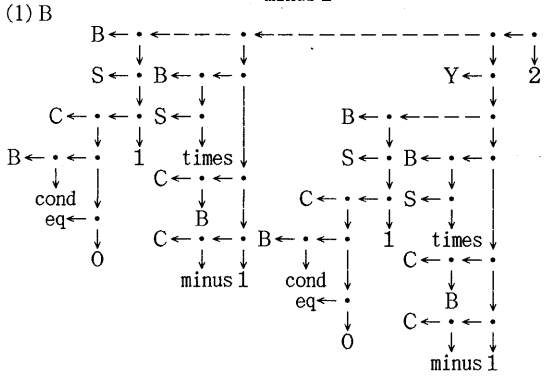
(1) 組合せ論理の書き換え規則に関する還元を指定回数以内で可能なだけ適用する。
（但し、Yコンビネータに関する書き換え規則は、他のコンビネータに関する書き換え規則の適用がすべて不可能な場合に初めて、その都度１回だけ試みられる。）
(2) 関数評価の書き換え規則に関する還元を可能な限り行ない、最終的に単一のアトム点になれば、そのアトムの値が求めるものであり、実行を終了する。それ以外では (1)に戻る。
(3) 上記 (1) および (2) の操作がいずれも全く実施できない状態になれば、実行を打ち切る。

以上の戦略は、与えられた組合せ論理コードが、正しく記述された（即ち、停止性が保証された）関数型プログラムから正しく“コンパイル”されたものであることを前提にしているので、この前提に沿わない場合には暴走する場合がある。
とくに、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。また、(1) で“指定回数”を用意したのは、組合せ論理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組合せ論理コードに引数２を与えて“素朴に”還元する過程を示す。
Yコンビネータに関する書き換え規則は次のような非共有形式を採用する。

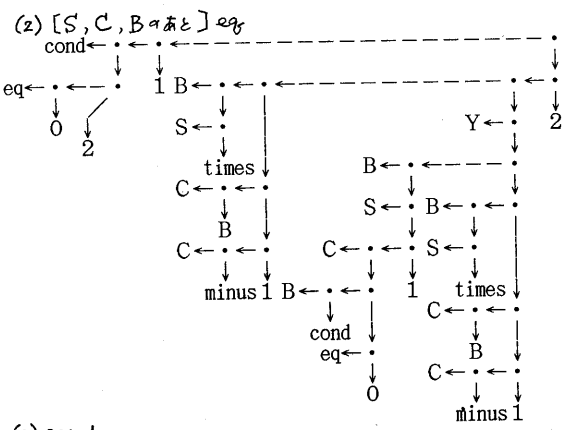


(0) Y

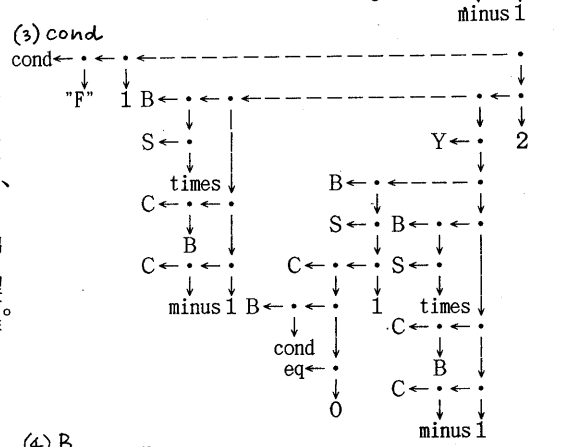
(1) B



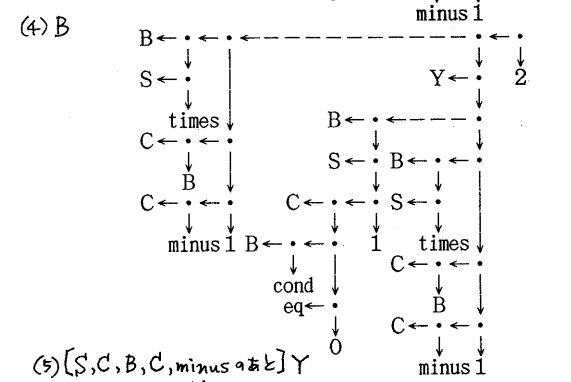
(2) [S, C, Bのあと] eq



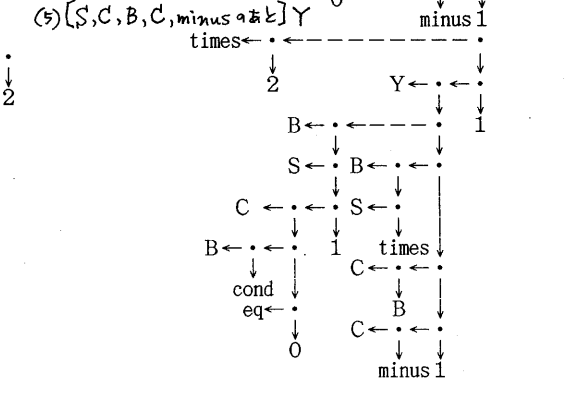
(3) cond

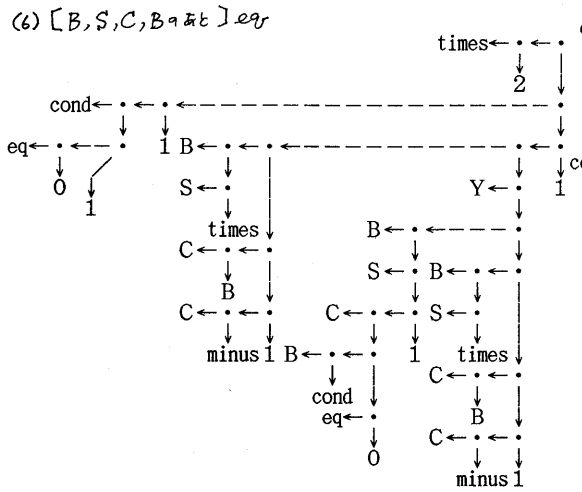


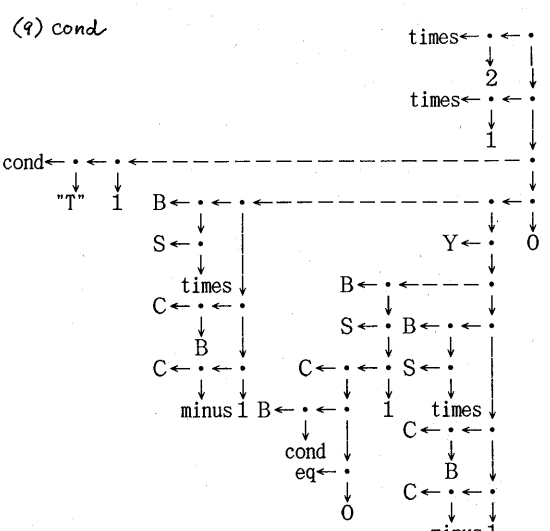
(4) B

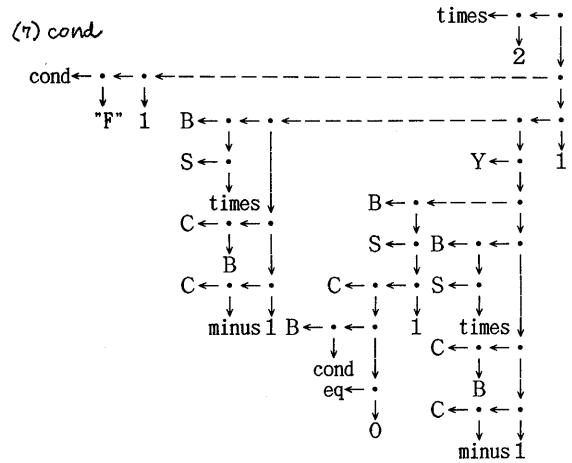


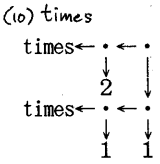
(5) [S, C, B, C, minusのあと] Y

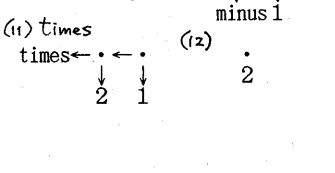








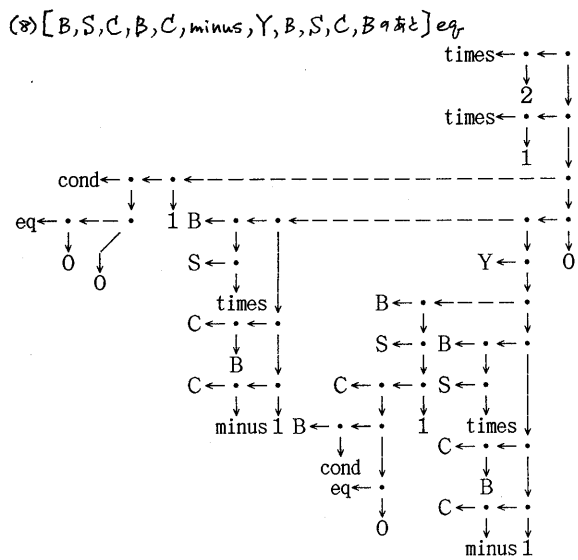




7. おわりに

本稿で述べたデータ構造に関する事柄は些細な事実の集まりにすぎないかもしれない。しかしながら、これらの事実を陽に指摘した文献類を知らないので、敢えて率直に公開することで、資料としての価値に期待する他に、今後の本分野における真の意味での共有形式での再帰プログラムのグラフ還元が実現可能か否かの問題にいささかでも寄与できれば幸いである。

末筆ながら、本研究の機会を提供される棟上昭男情報アーキテクチャ部長、および当研究室各位をはじめとする関連諸氏に感謝したい。



参考文献
[1]例えば 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.H.Fasel,R.M.Keller(Eds.): "Graph Reduction", Lecture Notes in Computer Science, No.279, Springer-Verlag, 1987
[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]杉藤: "コンビネータとグラフ還元"、電子情報通信学会ソフトウェアサイエンス研究会資料SS87-29 (1988.2.12)