ソフトウェア基礎論　36−2
（１９９０．　９．　２１）

関数型プログラムとグラフ還元

杉藤　芳雄
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

あらまし　関数型プログラムの処理系を組合せ論理の還元に依存して実現することを考える場合、特にいわゆるグラフ還元の利用が注目される。Turner がこの方式の有効性を示す際に、再帰呼出しのデータ構造としてサイクル構造を活用できることを挙げたが、その具体的な方法には言及していない。本稿ではその主張の可否をサイクル構造問題と称して検討する。その結果は、彼の提案するスタックとグラフ表現の併用という枠組みで階乗プログラムを例題として試行する場合、書き換え規則における置換セルの管理法に留意すれば、彼の主張が成立することを導ける。

On Functional Program and Graph Reduction

Yoshio SUGITO
Computer Language Section, Computer Science Division
ELECTROTECHNICAL LABORATORY
1-1-4 Umezono, Tsukuba-shi, Ibaraki-ken, 305, JAPAN

Abstract　When we intend to implement a system of processing functional programs with the aide of a reduction scheme based on the combinatory logic, it may be remarkable to adopt what we call 'graph reduction'. D.A.Turner showed the effectiveness of a system using a graph reduction, and claimed as one of its merits an availability of a cycle structure for a recursive call. Unfortunately it was not a concrete explanation, it remained some doubt about its straightforward realization. We call the doubt 'a cycle structure problem', and in this report we investigate its right or wrong in his framework made of a special stack and a graph representation, taking as an example a factorial calculation program. The result is that if we carefully decide the graph rewriting rules so as to keep its replaced parts properly, his assertion will hold.

1 はじめに

関数型言語で記述したプログラムを関数型プログラムと称することにし、その処理系を還元 (reduction) の実行で実現する際に遭遇する問題点について、特にいわゆるグラフ還元 (graph reduction)[1] を利用する場合のサイクル構造の取扱いに焦点をあてて検討する。

関数型言語とはという大上段な議論を展開しなくても今更疑問を呈する向きはあるまいと思えるほど、その存在意義が確立しているのが関数型言語の今日であろう。そこで以降では関数型言語そのものの詳細な説明は省略し、それは関数名とその引数という組が構文上の基本単位になっており、引数として関数をも許容することで意味論的にも興味深いものになっている言語であるとだけ付言しておく。

関数型プログラムの処理系 [2] という問題を考えるとき、関数型言語という用語が登場する以前から“関数”という概念と直接取組んできた研究分野である組合せ論理 (combinatory logic)[3] や、ラムダ計算 (λ-calculus)[4] の成果を、関数型プログラムの処理系に直接あるいは間接に反映させるのは極めて正統的な接近法であろう。

本稿で取扱う処理方式はこのような“本流”に位置するもので、Turner[5] が先駆的に行なったものである。彼は、関数型プログラムという原始コードを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、組合せ論理に基づく還元により評価（“実行”に相当）するという処理方式に関して、目的コード長の最適化や正規グラフ還元 (normal graph reduction) を導入すること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。

ここで還元とは、入力記号列に対して予め用意された書き換え規則の集まりの中から適用可能なものを施しつつ変換していく過程のことであり、適用された書き換え規則の時系列として定義される。

そして還元の一形態として、書き換え規則を適用する際に入力記号列のデータ構造を積極的に利用するグラフ還元と称するものがある。上述の Turner の文献に登場する正規グラフ還元は勿論その一例である。

以降では、組合せ論理の還元に依存する関数型プログラムの処理方式を概観したあと、Turner が文献 [5] で概観的に述べている特殊スタック併用のグラフ還元方式をやや詳細に展開し、その方式による階乗プログラムの例題を図解で試行する。そして、彼がやはり文献 [5] でグラフ還元の長所として言及している再帰呼出しに対するサイクル構造の活用という主張に関して、前述の試行をもとに検討する。

2 関数型プログラムと還元

組合せ論理は、組合せ項 (combinatory term) を基本構成要素とする記号系列として表現され、特にコンビネータと称する組合せ項が“演算子”あるいは“関数”の役割を果していて、後続のいくつかの組合せ項の並びを“引数”のように扱いながら、記号系列を変換（即ち、還元）することにより、ラムダ計算における λ 変換と同様の作用を、ラムダ計算特有の束縛変数という（やや扱いにくい）概念を導入することなく実現するものである。

この説明だけでも、関数型プログラムの処理に組合せ論理が向いていることが推察できるが、コンビネータが関数的な役割を果すことを見るために、代表的なコンビネータの書き換え規則を以下に例示する。

ここで記されている英字のうち、各大文字はコンビネータという組合せ項であり、各小文字はコンビネータの“引数”に相当する組合せ項である。矢印の右辺は、左辺に登場するコンビネータと“引数”との並び［これをリデックス (redex) と称する］に関する作用の結果であり、この効果を書き換え規則の適用という。

```
S f g x  ==>  f x(g x)
K f g    ==>  f
I f      ==>  f
B f g x  ==>  f(g x)
C f g x  ==>  f x g
Y f      ==>  f(Y f)
```

ここで注意すべきは、コンビネータの書き換え規則に関する還元だけでは、当然ながら関数型プログラムに含まれる四則計算（例えば sub 文）や比較（例えば eq 文）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、各演算に関する書き換え規則を追加しておく必要がある。

例えば、後述の階乗を求める関数型プログラムに普通登場する演算は、次のような書き換え規則として表現できる。

```
sub    f  g   ==>  (f - g) の実行値
tim    f  g   ==>  (f * g) の実行値
eq     f  g   ==>  "true"          f = g   のとき
                   "false"         f ≠ g   のとき
if  f  g  x   ==>  g               f = "true" のとき
                   x               f = "false" のとき
```

関数型プログラムを組合せ論理コードに“コンパイル”するための具体的な手順は [5] 等に譲ることにして、ここでは与えられた引数 n の階乗を求める関数型プログラムと、そのプログラムの組合せ論理コードへのコンパイル結果［目的コード長の最適化も施してあるもの］との例を示しておくだけにする。

関数型プログラム（原始コード）は例えば次のように記述できる。

```
fac n == if n=0 then 1 else n*fac(n-1)
```

これを組合せ論理コードにコンパイルするには、2項関係が明示的になる括弧対表現が望ましいので、その形式による表現も述べておく。

```
fac n == (((if ((eq n) 0))1)((tim n)(fac ((sub n)1))))
```

上記プログラムに対するコンパイル結果である組合せ論理コード（目的コード）は、次のようになる。

```
fac == S(C(B if(eq 0))1)(S tim(B fac(C sub 1)))
```

この目的コードには引数 n に相当する項が登場していないことに留意していただきたい。実は、このコードを“実行”するには、引数の具体値を目的コード本体に後置させることにより適用形 (application form) を構成したあと、組合せ論理／演算評価系に基づく還元を施すのである。
例えば、極めて簡単な例題として 0 の階乗を求めるには、上記目的コードに引数 0 を後置させて (fac 0) としたあとで還元を施せばよく、次のような経過となる。

```
fac 0 ==   (S(C(B if(eq 0))1)(S tim(B fac(C sub 1))))0
      =S=> (C(B if(eq 0))1)0((S tim(B fac(C sub 1)))0)
      =C=> (B if(eq 0))0 1((S tim(B fac(C sub 1)))0)
      =B=> if((eq 0)0)1((S tim(B fac(C sub 1)))0)
      =eq=>if true 1((S tim(B fac(C sub 1)))0)      勿論 ((eq 0)0)=true
      =if[true]=> 1
```

即ち、0 の階乗が値 1 として評価されたことになる。この還元過程の記述は、2項関係の括弧対が左詰めの場合や最も外側にある場合には適宜省略されるという習慣にほぼ従っている。

3 組合せ論理とグラフ還元

それでは一般に正整数 N の階乗を求める場合について、前章と同様に (fac N) として還元することを考える。

```
fac N ==   (S(C(B if(eq 0))1)(S tim(B fac(C sub 1))))N
      =S=> (C(B if(eq 0))1)N((S tim(B fac(C sub 1)))N)
      =C=> (B if(eq 0))N 1((S tim(B fac(C sub 1)))N)
      =B=> if((eq 0)N)1((S tim(B fac(C sub 1)))N)
      =eq=>if false 1((S tim(B fac(C sub 1)))N)   N>0 ゆえ ((eq 0)N)=false
      =if[false]=> ((S tim(B fac(C sub 1)))N)
      =S=> tim N((B fac(C sub 1))N)
      =B=> tim N((fac((C sub 1)N))
      =C=> tim N((fac(sub N 1))
      =sub=> tim N(fac(N-1))
```

ここで fac 自身の再帰呼出しに取組まざるを得なくなる。記号列上での対処は、本体自身のコピーを挿入することと仮定すれば、次のように進行していくことになる。

```
tim N(fac(N-1))
  =copy=> tim N((S(C(B if(eq 0))1)(S tim(B fac(C sub 1))))(N-1))
  ==>......
  =sub=> tim N(tim (N-1)(fac(N-2)))
  =copy=> tim N(tim (N-1)((S(C(B if(eq 0))1)(S tim(B fac(C sub 1))))(N-2))
  ==>......
  =sub=> tim N(tim (N-1)(...(tim 1(fac 0)...)
  ==>......
```

```
=if[true]=> tim N(tim (N-1)(...(tim 1 1)...)
=tim=> tim N(tim (N-1)(...(tim 2 1)...)
=tim=>......
=tim=> N!
```

今まで述べてきた組合せ論理コードに関する還元の形態としては、本章で扱ういわゆるグラフ還元を用いることが少なくない。しかし、グラフ還元という概念に関する統一見解は見当らないので、以降では、次のように仮定する。

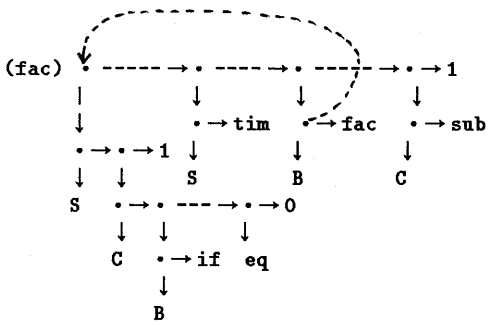
即ち、ある特定のデータ構造（一般にグラフ）で記号列を表現する状況を想定し、その記号列に施されるべき各書き換え規則が当該データ構造を反映して設定されている場合の還元形態を“グラフ還元”と称する。

組合せ論理のような2項関係の並びで構成される記号列を表現するためのデータ構造としては、ごく自然に2進木(binary tree)の利用が考えられるが、その場合でも各項の配置の仕方は一意的ではない。

そこで、ここではグラフ還元関連の文献で通常利用されている“上昇型”[6]を採用することにする。これは簡単に言えば、記号列を左から右に走査しつつ遭遇する各組合せ項を、左部分木→右部分木→根の順序で部分木を構成しながら上昇していく(これが上昇型の命名由来)過程において、各終端点に配置していく方式である。

例えば、前章の階乗計算の関数型プログラムの組合せ論理コードを上昇型で表現すれば次のようになる。

```
fac == S(C(B if(eq 0))1)(S tim(B fac(C sub 1)))
```



このときTurner[5]を注目したいのは、再帰呼出しfacの表現として、左上角の点（即ち、facプログラム本体の根）への有向辺（ポインタ）というサイクル構造を用いていることである。（上図の点線矢印を参照のこと。）

それの意味するところが、単なる論理的説明のためなのか、あるいは物理的にそのような構造を採用していることの表明のためなのかが不明瞭なのである。後述のように、このようなサイクル構造を実際に設定してグラフ還元を素朴に施すと、やがて不測の事態となるかもしれない筈であるが、そのあたりの対処法が文献[5]では読取れないのである。

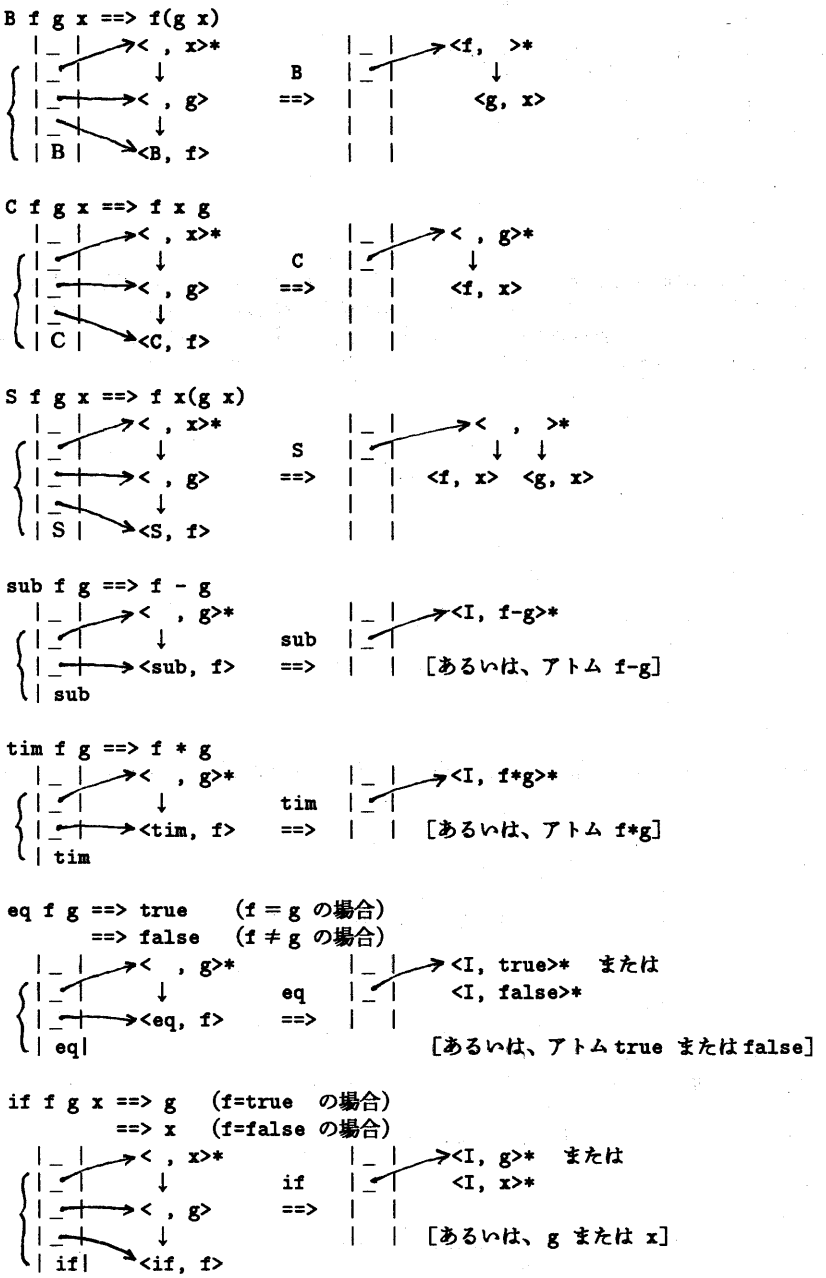
Turner[5]は、この2進木上でグラフ還元を施すために左祖先スタックLAS(left ancestor stack)を用意し、その支援のもとに次のように評価を進めている。

根から左部分木を下降して辿りつつ、点がセル［非終端点］の場合には、そのセルへのポインタをLASに積み込んで更に下降する。点が終端点［アトム］の場合、左部分木ゆえコンビネータか関数名かのいずれかの筈であり、そのアトムをLASに積み込むと共に、そのコンビネータ／関数が必要とする引数の個数分だけ遡ってLASの内容を調べる。LASの各段はセルへのポインタだから、その右部分木の内容（即ち、ポインタの指示するセル内の右項）を見ながらコンビネータ／関数としての評価を試みる。必要な引数の個数分だけLAS内を遡ることができさえすれば、コンビネータの場合には常に評価が可能であるが、関数の場合には例えば四則計算のようにアトム引数を要求することがあるので必ずしもその時点では評価が可能とは限らない。いずれにせよ評価が可能ならば書き換え規則に従って変換を行ない、その結果へのポインタがLASに格納される。［厳密に言えば、結果がアトムの場合には必ずしもLASには格納されない。］評価が不可能の場合には、未評価のまま右部分木に移動して、更に続行する。以上の操作がすべて不可能となる時点で評価が終了する。

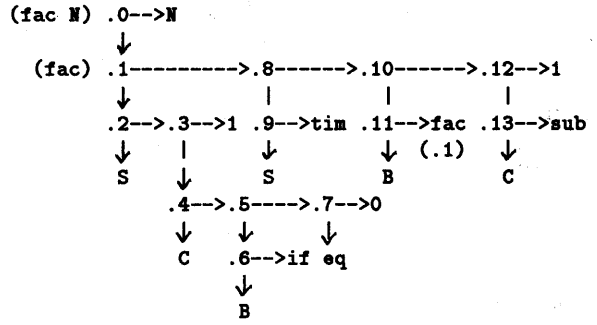
筆者は従来このサイクル構造問題を検討する際、より純粋にグラフ還元の枠組みを設定し、書き換え規則の適用時にはスタック等の支援を仰ぐべきではないと考えてきた（例えば[6]）。そのため、必要以上にサイクル構造の謎が深まるきらいがあったので、本稿では文献[5]に忠実に沿ってスタックの併用を容認することとして、グラフ表現および書き換え規則を定め、この階乗プログラムに関するグラフ還元を施すことを試みよう。

以下に示すのは、このようなLASを併用させた書き換え規則の例である。梯形は上部をスタックの底とするLASである。角括弧対は2項セルを表わし、角括弧対内の空白項と下向き矢印で他の角括弧対へのポインタを示してある。角括弧対内の左項にコンビネータIを置く機会が生れるのは、四則計算のように結果が単一アトムとなる際に形式的に角括弧対表現を存続するための便法である場合や、if演算のように結果が部分木となり得る際の継承セルの間接

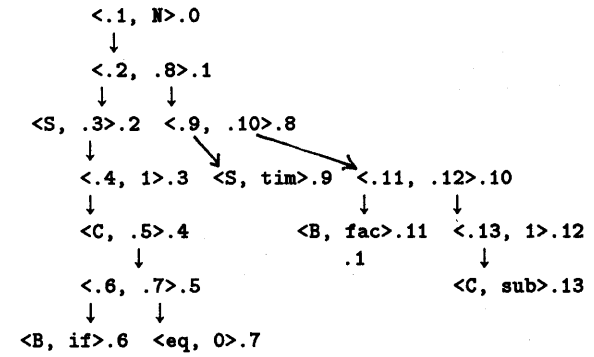
点化のための場合がある。f，g，xは部分木（の根）を表わし、状況に応じてアトムの場合もありうる。尚、＊印のある角括弧対は書き換えの前後で同一のセル（継承セルと称する）を意味している。



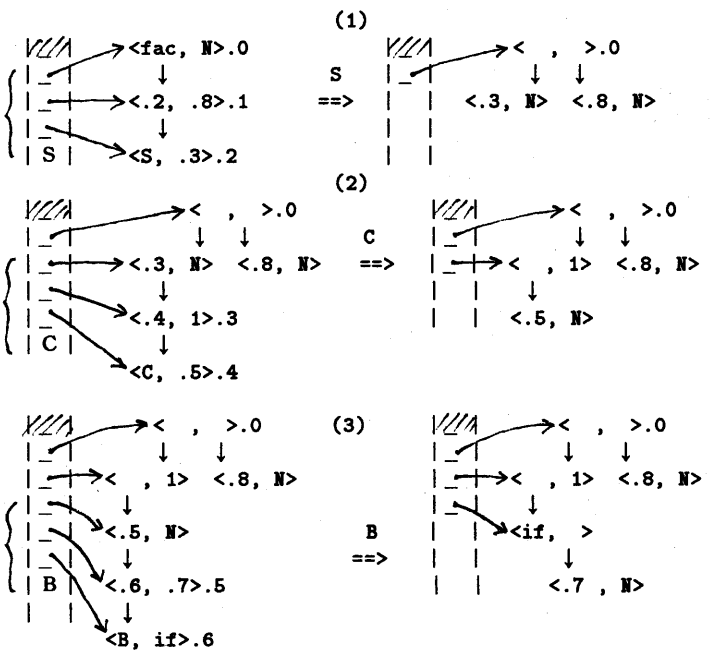
以上の準備のもとに、(fac N) を評価する問題を考え、次のような2進木表現を設定する。(fac N) 木は、前述のfac木を左部分木とし、引数Nを右部分木とする木である。そして記述を明確にするため、非終端点にはpreorder順序［木を根→左部分木→右部分木と辿る順序］で0を出発値として昇順で番号付けしていき、m番目の非終端点は“.m”として記すことにする。

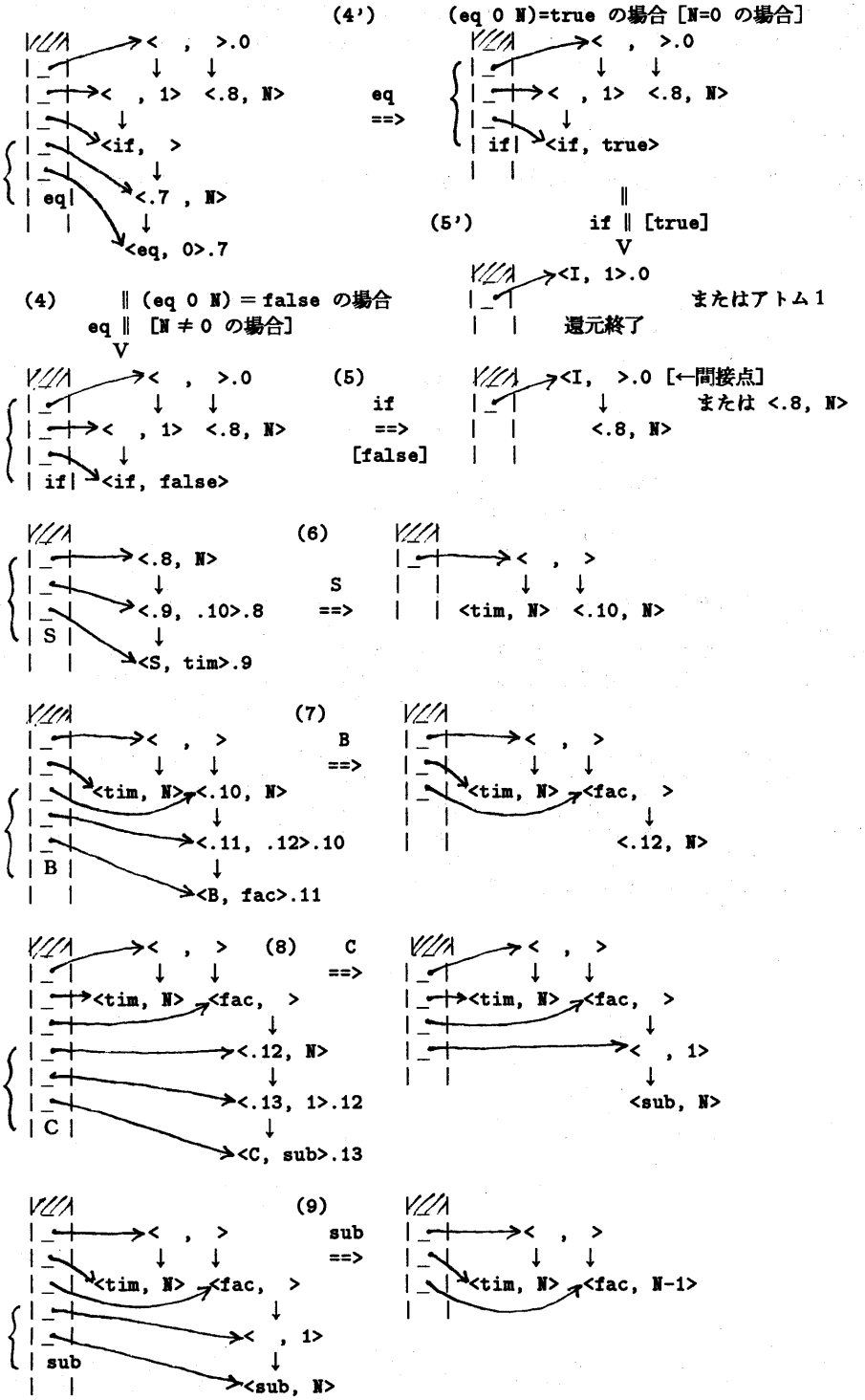


この非終端点番号は、いわば当該非終端点の番地に対応しているので、角括弧対で他のセルを参照する場合のポインタ表現としても利用できる。そこで、とくに出発状態の木に含まれる非終端点が存続していることを明示する意味あいからも、この記述法を採用する。ただし、非終端点番号が角括弧対の右隣に添えてある場合は、その角括弧対自身（もちろん非終端点）の番地であり、角括弧対内部にある場合は、その非終端点へのポインタである。



それでは、N の階乗を求めるグラフ還元を、上述の書き換え規則およびデータ構造を用いて実施する過程を以下に示す。

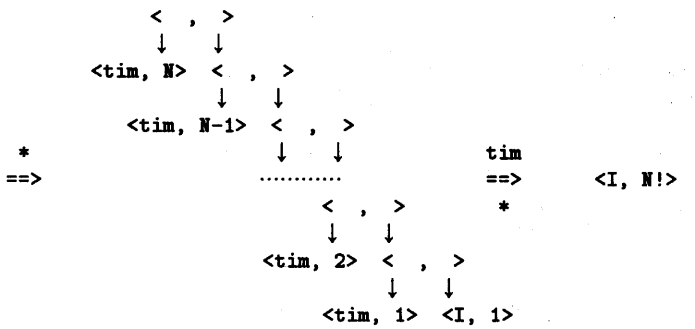



(4')
eq
==>
(eq 0 N)=true の場合 [N=0 の場合]
(5')
if ‖ [true]
(4) ‖ (eq 0 N) = false の場合
eq ‖ [N ≠ 0 の場合]
(5)
if
==>
[false]
または アトム 1
還元終了
<I, >.0 [←間接点]
または <.8, N>
(6)
S
==>
(7)
B
==>
(8)
C
==>
(9)
sub
==>

ここでfacの再帰呼出しという、暗礁に乗り上げる事態となるかもしれない難所にさしかかることになる。この状況でのサイクル構造の解釈の仕方が本稿の核心課題であり、グラフ還元を素朴に施すことの是非が問われているのであるが、文献[5]では具体的には言及されていないのである。

4 サイクル構造とグラフ還元

前章のサイクル構造の難所が何らかの方法により回避されるならば、階乗プログラムのグラフ還元は次のように進行していく筈である。



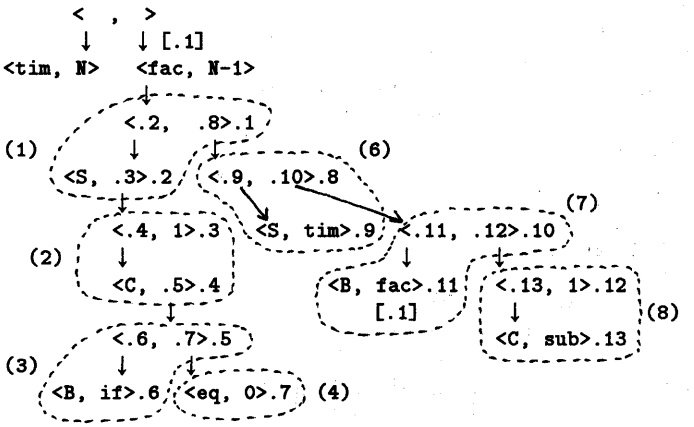
この問題の解決の鍵は、手近なところに存在していたのである。即ち、fac本体の根（つまり“.1”）に対して忠実に接近すればよいのである。

あとは、前章に示した書き換え規則において、書き換えの前後で継承されるセルとそうではないセルとの区別の仕方に関する配慮が必要なだけである。その配慮とは、継承されないセルだからといって直ちに廃棄するのではなく、再生され得るように保存しておくことである。あるいは極めて当然な言い方をすれば、再生可能なような書き換え規則を用意することである。

その再生可能な書き換え規則とは、意外にも実に簡単なものであり、継承されるセル（*印がついているもの）以外のセルをすべてそのままの状態で保存しておけばよいのである。［後述するように非継承セルのなかには廃棄可能なセルも存在するので、より実用的には、保存／廃棄をきめ細かく指定すべきである。］

実際に今の問題で、そのように書き換え規則を設定してあることを仮定すれば、fac本体の根に到達することにより、あとは芋蔓式に各段階の保存セルをたぐりよせればfac本体が再生あるいは復元できるのである。

より具体的には次のような順序でfac本体が復元されることになる。



ここで(n)として囲まれている部分は、前章の還元過程のn段階における書き換え規則により非継承セル（あるいは保存セル）となるものである。5段階と9段階における非継承セルは保存してもfac本体の再生に貢献していないので、廃棄可能であることが分る。

5段階での非継承セルには(if, false)が、9段階における非継承セルには(sub, N)が、それぞれ含まれているが、falseやNの値は演算環境（即ち、還元過程の進行状況）に応じて変化し得るものである。このことから一般に、非継

承セルの中に、アトムとして評価される値が演算環境に応じて可変であり得るものを含む場合には、それらの非継承セルは廃棄可能であることが帰結される。

逆に言えば、非継承セルの中に含まれるアトム類が、すべて目的コードに登場するアトム記号（コンビネータ、関数名、定数など）だけから成る場合には、これらの非継承セルは保存する必要がある。従って、各書き換え規則における非継承セルへの保存／廃棄の指定は文脈依存（あるいは環境依存）ということになる。

例えば、上記の還元過程がさらに進行して (fac 0) の評価に到達すると、それはアトム 1 という結果になり、あとは (tim p p-1) という形（ここで p や p-1 は整数）の乗算の連続で斜左上に上昇しつつ各段階でアトムを形成していく。そして、各段階での非継承セルはいずれも演算環境に依存して可変であり得るため、いずれも廃棄可能ということになり、最終的には N! という値のアトムだけが残される。

5 おわりに

組合せ論理の還元に基づく関数型プログラムの処理方式においてグラフ還元を利用する場合、Turner[5] が図示した組合せ論理コードのデータ構造表現に登場するサイクル構造の実際上の意義について検討した。

その結果、彼の提案した特殊スタック LAS を併用する状況では、各書き換え規則に出現する各セルの継承／非継承の区別および非継承セルにおける保存／非保存の区別を適宜指定することにより、サイクル構造を維持しつつ再帰関数を評価することが可能であることが判明した。

この事実を具体的に指摘している例を筆者の知る範囲の文献等では見出していないものの、本事実が既知あるいは周知であることを危惧するものである。しかし、たとえ既知であるとしても、本稿のように図解して積極的に示すことぐらいの意義はある事実であると考える。

サイクル構造に関して残された問題としては、書き換え規則に共有構造を含ませる状況、Turner の LAS のようなスタック類を一切用いない枠組みの状況、等においてサイクル構造の影響を検討することが挙げられる。

前者の状況については、本稿の階乗プログラムの例題では共有構造をもち得る唯一の書き換え規則である S コンビネータにはアトムの共有の可能性のみが存在していたので本稿では敢えて検討を控えたが、共有構造に関しては [7] を参照されたい。Takeichi[8] は共有構造をスタックとの併用で効率的に取扱う方法を提案しているが、サイクル構造問題との関連は不明である。

後者の状況に関しては筆者の懸案事項であるが、今回のささやかな光明を手がかりに検討を進めたい。

謝辞　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、およびご助言等をいただく当研究室各位をはじめとする関連諸氏に対して謝意を呈する。

References

[1] J.H.Fasel,R.M.Keller(Eds.):"Graph Reduction", Lecture Notes in Computer Science, No.279, Springer-Verlag, 1987.

[2] A.Diller:"Compiling Functional Languages", John Wiley & Sons LTD, 1988.

[3] 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.

[4] H.P.Barendregt:"The Lambda Calculus: Its Syntax and Semantics", North-Holland, 1984.

[5] D.A.Turner:"New Implementation Techniques for Applicative Languages", Software-Practice and Experience, Vol.9, pp.31-49, 1979.

[6] 杉藤：“グラフ還元とデータ構造”、電子情報通信学会ソフトウェアサイエンス研究会、SS88-44(1989 年 3 月 10 日).

[7] 杉藤：“グラフ還元における共有構造の効用と限界”、電子情報通信学会ソフトウェアサイエンス研究会、SS89-17(1989 年 9 月 8 日).

[8] M.Takeichi:"An Alternative Scheme for Evaluating Combinator Expressions", Journal of Information Processing, Vol.7, No.4, 1984.