ソフトウェア工学　52 — 4
（１９８７・２・１３）

関数型言語とグラフ還元について

杉藤　芳雄

電子技術総合研究所　ソフトウェア部　言語処理研究室

関数型言語プログラムの処理方式として有力視されているのは、ラムダ計算あるいは組合せ論理の世界での書き換え規則に基づく変換過程である還元によるものである。その還元の内でもとりわけグラフ還元が注目されているのは、共有構造を許すことにより評価済み値への２度目以降のアクセスでは評価が不要になることが大きな理由であろう。

本稿では、関数型言語と還元との緊密な関係を考察したあと、グラフ処理言語で作成された組合せ論理の世界での関数型言語処理系を用いて関数型言語プログラムのグラフ還元による“実行”過程を追跡するという事例研究を通して、とくにグラフ還元に的を絞ってその意義を検討する。

On Functional Language and Graph Reduction
------through the case study about the execution of a functional language program------

Yoshio SUGITO

Language Processing Section, Computer Science Division, ELECTROTECHNICAL LABORATORY
1-1-4, Umezono, Sakura-mura, Niihari-gun, Ibaraki-ken, 305 JAPAN

Reduction, which is transformation process by means of the rewriting rules in the frame of Lambda Calculus or of Combinatory Logic, is one of the most promising approaches for processing functional language programs.

In this report, after debating briefly the close relation between functional languages and reductions, we especially concentrate our attention on so-called graph reduction because of its merits on sharing structure, and discuss its significance through the case study tracing precisely the execution of a functional language program via graph reduction in the Combinatory Logic world with the aid of a reduction system implemented by a graph manipulation language.

１．　はじめに

関数型言語は、記述スタイルとして見ればいわゆる非手続き的言語の系統における重要な一種として類別されるものであり、理論面からはラムダ計算（λ-Calculus）や組合せ論理（Combinatory Logic）等の理論と密接な関係があることにより、多方面より活発に取り組まれている。

有り体に言えば、上記の理論を現実の計算機の世界に結びつける場合には関数型言語の形を採用すると扱いやすいという事実が、関数型言語の人気の大半の本音かもしれず、本稿もその流れに沿っていると言えよう。

すなわち、本稿では、関数型言語とその処理方式として有力視されているグラフ還元（Graph Reduction）とを主題にするが、どちらかといえばグラフ還元に焦点をあてるための引き立て役として関数型言語に登場してもらうことにする。そして、当研究室で開発したグラフ処理言語 GML により作成された組合せ論理に基づく関数型言語処理系を用いて、関数型言語プログラムをグラフ還元方式で実行させながら、視覚的トレースの援用で実行過程を吟味することを中心として、グラフ還元の意義を検討する。

２．　関数型言語と還元

関数型言語[1]の定義は色々与えられようが、その根幹にあるのが関数という概念であることは言うまでもない。一方、関数そのものを数学的対象として発展させたものにラムダ計算や組合せ論理等の理論がある[2]。従って、関数型言語とこれらの理論とが相性が良いことは当然かつ必然であろう。

ラムダ計算や組合せ論理では、それぞれの世界での合法的な“形”（項あるいは式）を書き換え規則により（合法性を保ちながら）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（Reduction）と称している。

それゆえ、関数型言語による表現（すなわちプログラム）をラムダ計算あるいは組合せ論理で許される“形”のコードに変換（この過程を[3]では類推から“コンパイル”と称している）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言語の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。

実際、例えば Turner[3]は、それまで実用的な計算機構としては殆ど注目されていなかった組合せ論理の世界で、コード長の最適化用に導入された新しい書き換え規則を併用して還元することによりほぼ実用的に関数型言語を処理できることを実証した。

また、関数型言語をラムダ計算あるいは組合せ論理の世界で還元させることをハードウェア化する試みとしての還元マシンもいくつか提案されている[4]。

更には、最近では組合せ論理をカテゴリ理論と関連づけることによって新たに導入された書き換え規則（すなわち Categorical Combinatory Logic）で還元することにより関数型言語を処理する方式の提案もある[5]。

以上の諸事実から、関数型言語と還元とは不即不離の関係にあることが理解されよう。

３．　グラフ還元

還元とは、端的に言えば“形”の書き換え系列であるが、“形”のデータ構造表現としての多様性や“形”への書き換え規則の適用箇所あるいは適用順序の多岐性により、いくつかの還元戦略が考えられることになる。

“形”のデータ構造表現としての分水嶺は、部分形の共有（sharing）構造を認めるか否かである。もちろん、共有構造を認めない場合には部分形の写し（copy）を用いることになる。

“形”への適用可能な書き換え規則が同一時点で複数存在する場合の選択問題は、関数とその引数における評価順という問題に対応しており、引数を最初に評価する値呼び（あるいは最内側計算規則）と関数を最初に評価する名前呼び（あるいは最外側計算規則）とに大別される。

[4]によれば、関数あるいは引数の定義（という部分形）を利用すべき各命令（という部分形）が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がストリング還元（String Reduction）であり、一方、定義を利用すべき各命令が当該定義への参照（reference）に関して評価していく方式がグラフ還元である。すなわち、部分形の共有構造を許してポインタ操作を積極的に利用するものがグラフ還元であり、許さないものがストリング還元である。とりわけグラフ還元が注目されているのは、共有を許すことにより評価済み値への２度目以降のアクセスでは評価が不要になること、換言すれば共有部分に関しては１度の評価操作で複数回分のそれを実施したことに相当していることが大きな理由であろう。

上記の各還元の説明の当否は別にして、驚くべきことには、例えば[6]にはグラフ還元なる術語が表題にまで登場しているのに本文ではその定義が述べられていないように、多くの文献では未定義のままグラフ還元を題材としている。また、大著である[7]には、還元の章が存在するのにそこには“グラフ還元”は一切現れず、あたかも認知すらしていないかのようである。

そこで本稿でも、それらのひそみに習い、還元を施すべき対象の“形”のデータ構造表現に共有構造を許しているもの（すなわち、“形”のデータ構造が木に限定されるのではなくてグラフになり得るもの）を漠然と“グラフ還元”と呼ぶことにする。

4．　グラフ処理言語によるグラフ還元の実現

当研究室で開発されたグラフ処理言語 GML(Graph Manipulation Language)[8]は、図形としてのグラフそのものによる定義で記述されるグラフ書き換え規則により、（点や辺にラベルが許された）グラフの書き換えを実現するプログラム言語である。

このグラフ書き換え規則は、書き換えるべき部分グラフをさがすための位置決め用グラフと、見つかった部分グラフを別のグラフで置き換えるための埋め込み用グラフとの対として、表現される。従って、GML のグラフ書き換え規則はグラフ還元等の還元を実現するのに適していることが容易に推測できよう。

現に[9][10]では、組合せ論理に関する還元を実行するシステム（これを[10]では組合せ子簡約系と称した）を GML で作成し、本稿の意味でのグラフ還元を実現している。特に[10]では、関数型言語のプログラムを組合せ論理の合法形コードに変換（コンパイル）し、そのコードを組合せ子簡約系および関数評価系により実行するシステムを GML で作成したことを報告している。

そこで本稿では[10]の延長上で関数型言語およびグラフ還元を捉えることにする。すなわち、関数型言語の処理を組合せ論理の世界で取り扱い、グラフ還元としては関数型言語プログラムの組合せ論理コードへの変換である“コンパイル”過程よりも組合せ論理コードという目的コードの実行過程に相当する組合せ子簡約系での還元を重視する方針である。

5．　グラフ還元の例

組合せ論理における主な書き換え規則は以下に示すものである。ここで大文字は組合せ子（Combinator）と称する“演算子”（あるいは関数）を、小文字は組合せ子の“引数”をそれぞれ表している。

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

関数型言語としては[10]のそれ（すなわち Feles 言語）を採用することにする。Feles 言語は四則演算等の常識的な関数名とそれに必要な引数とを並べたものを括弧で囲むという、極めてありふれたものである。本例では加算としての Plus 関数が用いられている。

Feles 言語プログラムを組合せ論理コードに“コンパイル”するための変換規則およびコード長の最適化規則は、[3]に述べられているように、それぞれ①～③および④～⑦である。ここでＥｉは組合せ論理として合法的な部分形である。手順としては、①～③により Feles 言語プログラムを組合せ論理の（一般に極めて長い）コードに変換したあと、そのコードに④～⑦をこの順序で適用しつつ最適化して短いコードを得る。

① [x]（E_1　E_2）　==>　S（[x] E_1）（[x] E_2）
② [x] x　==>　I
③ [x] y　==>　K　y　　（但しｙは定数またはｘ以外の変数）
④ S（K　E_1）（K　E_2）　==>　K（E_1　E_2）
⑤ S（K　E_1）I　==>　E_1
⑥ S（K　E_1）E_2　==>　B　E_1　E_2
⑦ S　E_1（K　E_2）　==>　C　E_1　E_2

ここで考える問題は、次のように定義された４引数関数（これを仮に Ennuyeux と呼ぶ）に対して、別に定義された３乗関数 Thrice、２乗関数 Twice、２倍関数 Double、および整数１をそれぞれ引数ｐ，ｑ，ｒ，ｓとして施したものを評価することである。

Def Ennuyeux p q r s = ((p (q r)) s)

この式を評価すると次のような値になる筈である。

Ennuyeux(Thrice,Twice,Double,1) = (Thrice (Twice Double))1 = ((2 x 1)²)³ = (2²)³ = 4³ = 64

まず各関数の定義とその“コンパイル”を行うことにする。

Def Double x = ((plus x) x)

Def Double = [x]((plus x) x) = S([x](plus x))([x]x) = S(S([x]plus)([x]x))([x]x) = S(S(K plus)I)I
= S(plus)I = S plus I

Def Twice f x = f(f x)

Def Twice = [f]([x](f(f x))) = [f](S([x]f)([x](f x))) = [f](S(K f)(S([x]f)([x]x)))
= [f](S(K f)(S(K f)I)) = [f](S(K f)(f)) = [f](B f (f)) = [f]((B f) f)
= S([f](B f))([f]f) = S(S([f]B)([f]f))([f]f) = S(S(K B)I)I = S(B)I = S B I

Def Thrice f x = f(f (f x))

Def Thrice = [f]([x](f(f(f x)))) = S B (S B I)

Def Ennuyeux p q r s = ((p (q r)) s)

Def Ennuyeux p q r = [s]((p (q r)) s) = (p (q r))

Def Ennuyeux p q = [r](p (q r)) = (B p q)

Def Ennuyeux p = [q]((B p) q) = (B p)

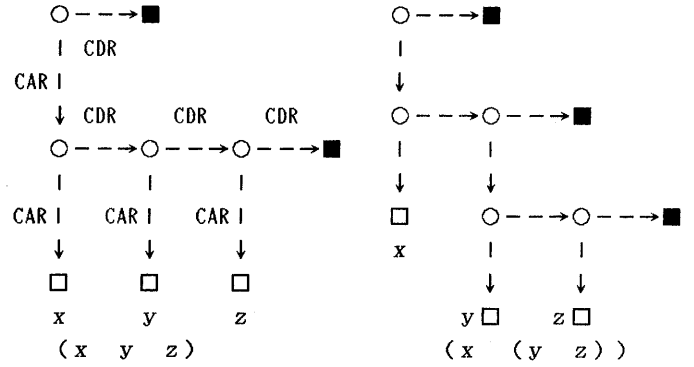
Def Ennuyeux = [p](B p) = B

以上から、Ennuyeux(Thrice,Twice,Double,1) のコンパイル結果は次式である。

(B (S B (S B I)) (S B I) (S Plus I) 1)
↑ Thrice Twice Double s
Ennuyeux

あとは上式を組合せ子簡約系（Plus に関しては関数評価系）により還元していけばよい訳であるが、その途中経過をトレースしつつグラフ還元の効果を観察していこう。

本実験で採用した組合せ論理コードのデータ構造表現は次のような点／辺ラベル付き有向グラフによるリスト構造である。ここで○、□、■はそれぞれセル点、アトム点、Nil アトム点を表している。尚、下右図のように特に辺ラベルを明示しない場合、垂直辺、水平辺はそれぞれ CAR辺、CDR辺を意味することにする。



以下では、本式に関するグラフ還元の実行系列を示す。ここでＳ，Ｂ，Ｉ，()，Ｃｏはそれぞれ Ｓ組合せ子還元、Ｂ組合せ子還元、Ｉ組合せ子還元、冗長括弧の除去、共有部分の複製を意味する。Plus[i,i] は“i + i”なる加算のことである。また、適宜表示される中間結果の組合せ論理コードでは、Ｐは Plus の略記を、@が前置されたものは共有アトムを、下線部は共有部分形を、それぞれ表している。

B,(),S,B,(),S,I,(),(),B,＜１～10ステップ＞
(((S B I) (S P I)) (((S B I) (S P I) (((S B I) (S P I)) 1)))
Co,(),S,I,(),(),B,(),S,I,(),Co,＜11～22ステップ＞
(((S B I) (S @P I)) (((S B I) (S @P I)) (@P ((S @P I) @1) ((S @P I) @1))))
(),S,I,(),(),B,(),S,I,(),(),S,I,(),＜23～36ステップ＞
(@P ((@S @P @I) ((B (@S @P @I) (@S @P @I)) (@P ((@S @P @I) @1) ((@S @P @I) @1))))

((@S @P @1) ((B (@S @P @1) (@S @P @1)) (@P ((@S @P @1) @1) ((@S @P @1) @1)))))

(),B,<37～38ステップ>

(@P ((@S @P @1) ((@S @P @1) ((@S @P @1) (@P ((@S @P @1) @1) ((@S @P @1) @1)))))

((@S @P @1) ((@S @P @1) ((@S @P @1) (@P ((@S @P @1) @1) ((@S @P @1) @1))))))

(),S,<39～40ステップ>

(@P ((S @P @1) (@P ((S @P @1) (@P ((S @P @1) @1) ((S @P @1) @1)))

(@I ((S @P @1) (@P ((S @P @1) @1) ((S @P @1) @1))))))

((S @P @1) (@P ((S @P @1) (@P ((S @P @1) @1) ((S @P @1) @1)))

(@I ((S @P @1) (@P ((S @P @1) @1) ((S @P @1) @1)))))))

I,(),Co,(),S,I,(),Co,<41～48ステップ>

(@P ((S @P I) (@P (P (@P ((S P I) @1) ((S @P I) @1)) (@P ((S P I) @1) ((S @P I) @1)))

((S @P I) (@P ((S P I) @1) ((S @P I) @1)))))

((S @P I) (@P (P (@P ((S P I) @1) ((S @P I) @1)) (@P ((S P I) @1) ((S @P I) @1)))

((S @P I) (@P ((S P I) @1) ((S @P I) @1))))))

(),S,I,(),<49～52ステップ>

(@P ((S @P I) (@P (P (@P (P @1 @1) ((S @P I) @1)) (@P (P @1 @1) ((S @P I) @1)))

((S @P I) (@P (P @1 @1) ((S @P I) @1)))))

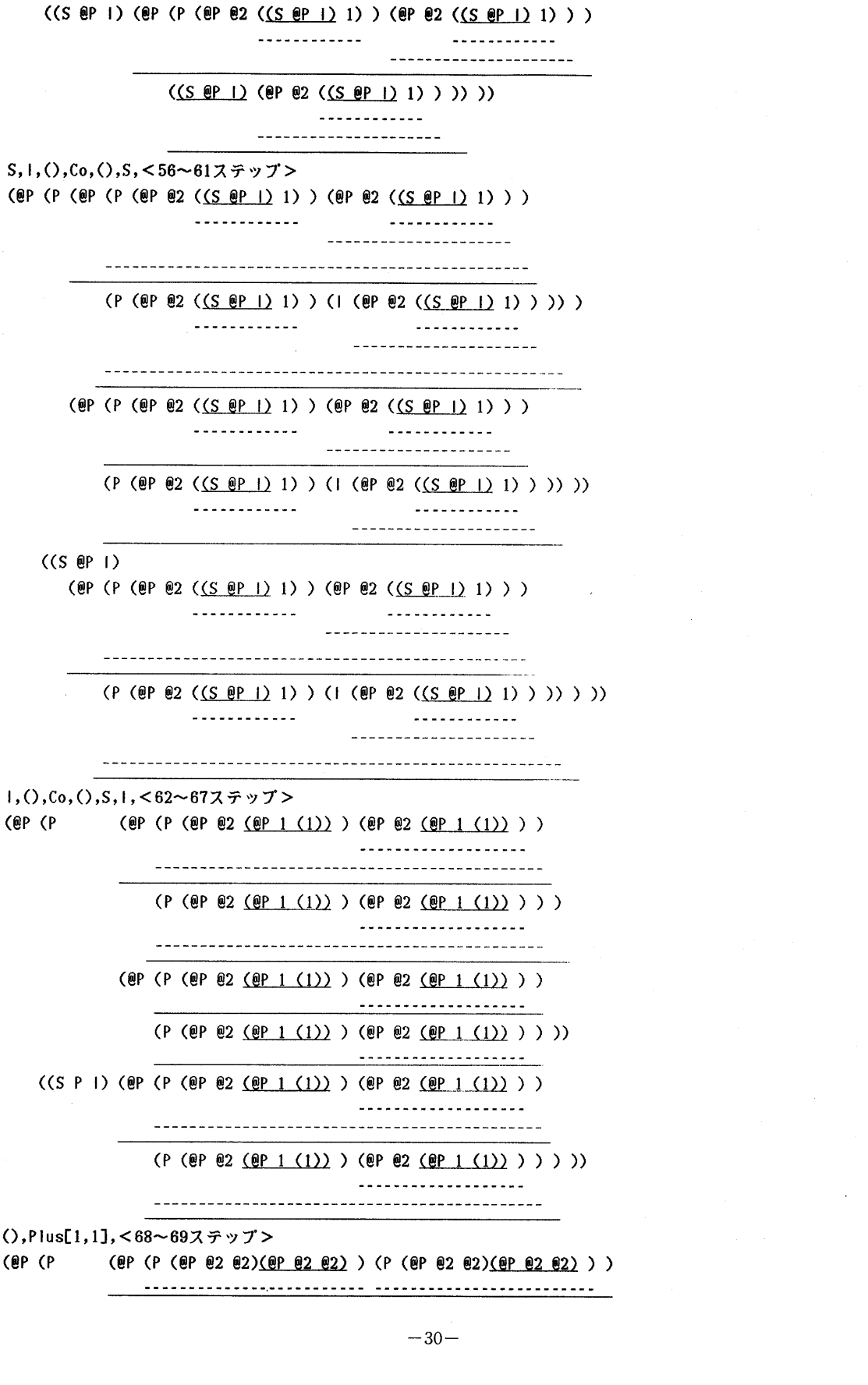
((S @P I) (@P (P (@P (P @1 @1) ((S @P I) @1)) (@P (P @1 @1) ((S @P I) @1)))

((S @P I) (@P (P @1 @1) ((S @P I) @1))))))

Plus[1,1],Co,(),<53～55ステップ>

(@P (S P I (@P (P (@P @2 ((S @P I) 1)) (@P @2 ((S @P I) 1)))

((S @P I) (@P @2 ((S @P I) 1)))))


((S @P I) (@P (P (@P @2 ((S @P I) 1)) (@P @2 ((S @P I) 1)))
((S @P I) (@P @2 ((S @P I) 1))))))
S,I,(),Co,(),S,<56～61ステップ>
(@P (P (@P (P (@P @2 ((S @P I) 1)) (@P @2 ((S @P I) 1)))
(P (@P @2 ((S @P I) 1)) (I (@P @2 ((S @P I) 1)))))
(@P (P (@P @2 ((S @P I) 1)) (@P @2 ((S @P I) 1)))
(P (@P @2 ((S @P I) 1)) (I (@P @2 ((S @P I) 1))))))
((S @P I)
(@P (P (@P @2 ((S @P I) 1)) (@P @2 ((S @P I) 1)))
(P (@P @2 ((S @P I) 1)) (I (@P @2 ((S @P I) 1)))))))
I,(),Co,(),S,I,<62～67ステップ>
(@P (P (@P (P (@P @2 (@P 1 (1))) (@P @2 (@P 1 (1))))
(P (@P @2 (@P 1 (1))) (@P @2 (@P 1 (1)))))
(@P (P (@P @2 (@P 1 (1))) (@P @2 (@P 1 (1))))
(P (@P @2 (@P 1 (1))) (@P @2 (@P 1 (1))))))
((S P I) (@P (P (@P @2 (@P 1 (1))) (@P @2 (@P 1 (1))))
(P (@P @2 (@P 1 (1))) (@P @2 (@P 1 (1)))))))
(),Plus[1,1],<68～69ステップ>
(@P (P (@P (P (@P @2 @2)(@P @2 @2)) (P (@P @2 @2)(@P @2 @2)))

```
          (@P (P (@P @2 @2)(@P @2 @2) ) (P (@P @2 @2)(@P @2 @2) ) ))
             ----------------------- -----------------------
    ((S P I)(@P (P (@P @2 @2)(@P @2 @2) ) (P (@P @2 @2)(@P @2 @2) ) ) ))
             ------------------------ ------------------------
Plus[2,2],Plus[2,2],<70～71ステップ>
(@P (P       (@P (P 4 (@P 2 2) ) (P 4 (@P 2 2) ) ) (@P (P 4 (@P 2 2) ) (P 4 (@P 2 2) ) ))
             ---------------- ---------------      ---------------- ----------------
    ((S P I) (@P (P 4 (@P 2 2) ) (P 4 (@P 2 2) ) ) ))
             ---------------- ---------------
Plus[2,2],Plus[4,4],<72～73ステップ>
(@P (P       (@P (P 4 4) @8) (@P (P 4 4) @8))   ((S P I) (@P (P 4 4) @8) ))
             ---------------                              ---------------
Plus[4,4],Plus[8,8],<74～75ステップ>
(@P (P @16 (@P 8 8)) ((S P I) @16))
Plus[8,8],Plus[16,16],<76～77ステップ>
(P 32 ((S P I) 16))
(),S,I,(),<78～81ステップ>
(P 32 (P 16 16))
Plus[16,16],Plus[32,32]<82～83ステップ>
64
```

以上の経過により、全83ステップで所望の結果である値64が得られた。内訳は、（）、Ｃｏ、組合せ子、および Plus の各ステップ数がそれぞれ 31、６、43、12 である。

留意すべきこととしては、（）や Ｃｏ の変換操作の出現頻度は“形”のデータ構造に依存する面が強く、本実験で採用したものが必ずしも芳しいものではないかもしれないことである。もちろん、“形”のデータ構造表現は GML のグラフ書き換え規則の記述に直接の影響を与え、特に書き換えの際における共有部分の取扱いの記述あるいは処理に関する難易を左右することになる。

共有構造を許すことによる重要な恩恵は、Plus や組合せ子に関する変換操作の出現回数が大幅に減少しやすいことである。実際、引数の値１から出発して加算だけで値64を得るには、最悪の場合63回の加算が必要となるが、本例ではその約５分の１だけで済ませており、大いに注目される効果である。例えば、第53ステップの１回の加算により６箇所に結果が波及していることは、第55ステップ直後の中間結果形での（加算の結果である）“@2”の個数を調べれば明らかである。

同様に、例えば第66ステップのＳ組合せ子に関する１回の変換操作は、実に12箇所に結果が波及していることが、第61ステップ直後および第67ステップ直後のそれぞれの中間結果形を比較することにより判明する。組合せ子に関しては、最悪の場合の出現回数を求めることが煩雑ゆえ、共有構造を許すことによる効果の定量的な評価は省略するが、抽出例から推測する限り、やはり数分の１程度には減少するであろう。

もちろん、共有構造を許すことによる反対給付も当然存在しており、例えば書き換えの行き詰まりを解消するための共有部分の複製操作は、“形”のデータ構造の選択にも依存しようが、本質的に内包している問題点と言えよう。上例では複製操作が６回ゆえ、それほどの弊害ではないように見受けられるが、本実験では GML の特殊性のためか一般に複製操作は時間を浪費するものなので、出現回数が少ないのに越したことはない。

尚、並列処理による還元を遂行する場合には、共有構造を許すことが妨げになることは言を待たない。

最後に評価順に関して付言すると、本実験では書き換え可能な候補が複数個存在する場合、 GML の処理系の特性である「原則的にはどこが選択されるかは処理系まかせ」という点を利用して、同一水準にある部分形では最左端のものが選択されるという制約だけを設けている。従って、計算規則において最内側か最外側かの限定はせず、単に（或る水準での）最左端だけを選択するようにしてある。グラフ還元の効果を高めるためには、書き換え可能な共有部分が増加するような評価順が望まれるが、[4]によればそれは最外側計算規則が該当している。

６．　おわりに

グラフ還元という技法に関して、組合せ論理に基づく関数型言語処理系での事例研究を通して、若干の風穴

を開けるつもりで取り組んで来たが、少なくともその効果の一端を味わうことはできた。定性的よりは定量的な議論を追求したい訳であるが、志半ばの感が深い。

“形”のデータ構造表現の相違による影響を調べることが当面の課題である。それと、今回は意図的に回避した再帰型プログラムでのグラフ還元の効果を検討することも、今後に残されている。

末筆ながら、本研究の機会を与えて頂いた棟上昭男ソフトウェア部長、および好ましい研究環境を醸成して下さる言語処理研究室の皆様に謝意を呈します。

参考文献

[1]P.Henderson: “Functional Programming--Application and Implementation”,Prentice-Hall International,1980

[2]J.R.Hindley,J.P.Seldin: “Introduction to Combinators and λ-Calculus”, London Mathematical Society Student Texts 1, Cambridge University Press, 1986

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

[4]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

[5]G.Cousineau,P-L.Curien,M.Mauny: “The categorical abstract machine”, LNCS 201, pp.50-64, Springer-Verlag, 1985

[6]J.Hughes: “Graph Reduction with Super-Combinators”, Oxford UniversityComputing Laboratory Technical Monograph PRG-28, June 1982

[7]H.P.Barendregt: “The Lambda Calculus--Its Syntax and Semantics”, North Holland,1981

[8]杉藤、真野、鳥居：“グラフ処理言語 GML-56”、情報処理学会論文誌、Vol.20, No.5, pp.427-434, 1979

[9]Y.Sugito, Y.Mano: “Some applications using a graph manipulation language GML”, Proc. of IEEE-1984 Workshop on Visual Languages, PP.53-58

[10] 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)