ソフトウェア工学 46－5
（１９８６．２．６）

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

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）

１．はじめに

近年、関数型言語の処理系としてグラフ簡約(Graph Reduction)を利用する方法が注目されている。一方では項書き換え系(Term Rewriting System)や、組合せ論理(Combinatory Logic)に基く簡約系である組合せ子(combinator)簡約系の研究も活発である[1]。これらの研究にはグラフの書き換えが密接に関連している。当研究室で開発されたグラフ処理言語ＧＭＬ（例えば[2][3]）はグラフの書き換えを主目的とするプログラミング言語であり、組合せ論理の組合せ子に関する簡約をグラフの書き換えと看做すことで組合せ子簡約系の実現にＧＭＬを直接利用できる見通しが期待されるが、それを実証することが本稿の目的の一つである。本稿のもう一つの目的は、関数型言語から対応する組合せ論理式コードへのコンパイラをやはりＧＭＬで実現し、これに算術／比較演算等の関数評価系と上述の組合せ子簡約系とを連結させることで構築された関数型プログラム実行系が、再帰的プログラムの当該コンパイラによる目的コードを（スタック機構等を用いずとも）実行することが可能か否かを調べることである。

２．グラフ処理言語ＧＭＬ

ＧＭＬ(Graph Manipulation Language)はグラフの書き換えを図形としてのグラフで記述することを最大の特徴とするプログラム言語である。ＧＭＬでのグラフ処理は一連のグラフ書き換え規則から成る。グラフの書き換えに関する基本操作は、与えられた入力グラフ内に、指定されたグラフ(MRG)と同形の部分グラフが含まれているか否かを調べること、および、その部分グラフを別に指定されたグラフ(ERG)で置き換えることであり、これらの操作はそれぞれマッチング、エンベッディングと称する。そして、グラフ書き換え規則は前述の２種類のグラフの対（MRG、ERG）を図形としてのグラフで表現することで定義される。ここで、MRG、ERGはそれぞれマッチング（ルール）グラフ、エンベッディング（ルール）グラフと呼ばれ、一括して規則グラフと称される。入力グラフ（IG）が当該MRGに同形な部分グラフＳを含む場合、Ｓは当該ERGに同形な別のグラフＴで置き換えられる。直観的に言えば、IGがMRGを含むならば、IGは書き換えによりIG－MRG＋ERGになる。

ＧＭＬプログラムの処理対象は、点や辺にラベルを有するグラフであり、これを（点／辺）ラベル付きグラフと呼ぶことにする。ラベルは　X(a1,a2,...,an)の形式であり、ここでＸは文字列を、ai（サフィックスとも称する）は整数を、それぞれ想定している。ただし、規則グラフの場合には、aiは整数値変数（とくにラベル変数と称する）も許される。

ＧＭＬの制御構造の表現方式としては、ブロック線図（流れ図の一種）を用いるものと、FORTRANをホスト言語とするものとがあるが、本稿では後者のみを利用している。ＧＭＬで記述されるプログラムは、FORTRANテキストと規則グラフから成る。テキストをコンパイルし、ＧＭＬパッケージ（初期化、マッチング、エンベッディング、等に関するFORTRANサブルーチン群）とリンクしたものを、入力グラフと共に実行することにより、結果（即ち、出力グラフ）を得ることになる。グラフィックス上で規則グラフや入力グラフを定義したり、出力グラフを見たりするために、グラフを外部表現（即ち、画面上の図形）と内部表現（即ち、データ構造）との間で両方向に変換するユーティリティである“グラフ・エディタ”が用意されている。

３．組合せ子簡約系

組合せ子は組合せ論理(Combinatory Logic)に登場する演算子であり、それに付随する引数類と共に以下のような変換を行なうものである。ここで大文字は組合せ子、小文字は引数（即ち、組合せ論理で許される項）をそれぞれ表わす。Ｙは不動点組合せ子と称し、後述の再帰的プログラムでは重要な役割を果たすことになる。

```
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)
```

組合せ子簡約系は、与えられた組合せ論理式に対して、上記の左辺の形（レデックスと称する）が見出されると対応する右辺の形で置き換えること（この操作をリダクションあるいは簡約と呼ぶ）を可能な限り続行する系である。それゆえ、グラフ処理言語で組合せ子簡約系を実現するには、組合せ論理式のグラフ表現を決定しておき、あとは一連の組合せ子に関する簡約規則をそれぞれグラフ書き換え規則と看做して記述すればよい。即ち、上記の左辺をマッチング・グラフで、対応する右辺をエンベッディング・グラフで、それぞれ表現すればよい。以上の説明からは系の実現に際して厄介な問題が存在しないように見受けられる。実際には、大別して、括弧の取り扱い、共有部分の取り扱い、レデックスの探索順、という３つの問題が潜在しているのである。

以下では、上述の状況を踏まえてグラフ処理言語ＧＭＬによる組合せ子簡約系の実現について述べる。

3.1　組合せ論理式のグラフ表現

組合せ論理で許される式は、組合せ項(combinatory term)の並びである。組合せ項（の集合）は次のように帰納的に定義される。

（i）　各アトム（即ち、変数または定数）は１つの組合せ項である。

（ii）　ＡおよびＢがそれぞれ組合せ項であれば、（ＡＢ）は１つの組合せ項である。

以上の定義から組合せ論理式に登場する記号要素はアトム記号と括弧記号だけなので、組合せ論理式のグラフ表現としてリスト構造を採用することが自然である。組合せ論理式の場合、組合せ項の並びの終了を示す特殊記号が括弧以外には用意されていないので、リスト構造に変換する際にはリスト項の並びの終了記号である'.NIL'が省略されているものと看做せばよい。あとはリスト構造に関するグラフ表現を決定すればよい。

(1)　終端点はアトムを、非終端点はセルをそれぞれ意味する。前者をアトム点、後者をセル点と呼ぶ。

(2)　辺は有向辺を用いる。セル点からは、２つの有向辺が出ており、一方は辺ラベル 'CAR'を有してそのセルに関する CAR部に向かい、他方は辺ラベル 'CDR'を有してその CDR部に向かっている。

(3)　アトム点のラベルとして、空アトムには 'NIL'を、非空アトムには 'A(n)' を、それぞれ用いる。後者の場合、'A' は 'ATOM' の略記であり、'n' は当該アトム名の先頭文字に対応するコード（整数値）である。

(4)　セル点のラベルとしては 'C'を用いるが、これは 'CELL' の略記である。

(5)　頂上のセル点は特に“ポインタ点”と称し、その CAR辺は与えられた組合せ論理式に相当するリスト構造全体に、その CDR辺は空アトムに、それぞれ向かっている。

(6)　CAR部（即ち、CAR 辺で指される点）に関して、それがセル点の場合はリスト項の並びの開始を意味する左括弧（開き括弧）に対応し、アトム点の場合にはアトムという単一リスト項に対応している。 CDR部（即ち、CDR 辺で指される点）に関して、それがセル点の場合は後続のリスト項に対応し、空アトム点の場合にはリスト項の並びの終了を意味する右括弧（閉じ括弧）に対応している。これら以外の場合（例えば、 CDR部が非空アトムの場合）は、組合せ論理式のリスト表現に登場しないものとする。

以上のグラフ表現に基いて、Ｃ組合せ子に関する簡約規則をＧＭＬの規則グラフで表現すると、図１のようになる。本例によりＧＭＬにおけるグラフの書き換えに関する記述法をも簡単に説明する。同図はグラフ・エディタの画面のハード・コピーであり、左側は画面操作に関するコマンド群のメニューが表示されている。正方形の枠内には１対の規則グラフが表示されており、その左半分はマッチング・グラフ（以下では MRG と記す）に、右半分はエンベッディング・グラフ（以下では ERG と記す）に、それぞれ対応している。各点内の数字は点対応番号と称されるもので、 MRG と ERG との間での点の対応関係を規定する。両グラフでの同一番号の点（例えば点１）は物理的に同じ点を指しており、グラフの書き換え後もその点が存続することを意味する。 MRG のみに存

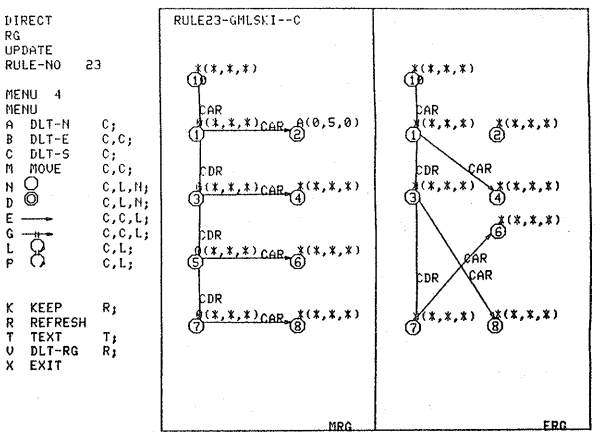

図１．　Ｃ組合せ子の簡約規則

在する番号の点（例えば点５）はグラフの書き換えにより消去されることを意味する。（このとき、その点に付随していた辺も消去される。）また、本例には登場しないが、 ERG のみに存在する番号の点はグラフの書き換えにより付加されることを意味する。点ラベル内にある'*' 記号は、 MRG では“任意の値”(don't care)を意味し、 ERG では MRG 内の同一の点対応番号を有する点の点ラベルの対応する位置にある値を継承することを意味する。各点ラベルが３つのサフィックスを有しているのは、アトムのコード値以外にも各種の制御（例えばグラフを辿る場合）や目印が必要となるからである。MRG の点１、３、５、７の点ラベルは同図では不明瞭だが C(*,*,*) なのでセル点であり、点２はアトム点でその第２サフィックスの値５は文字 'C' に対応するコードとして定めてあるので組合せ子Ｃに相当するアトムである。点４、６、８の点ラベルが *(*,*,*) となっているのは、アトム点またはセル点のいずれかの可能性があるからであり、これらの点は組合せ子Ｃの各引数に対応している。点10の点ラベルが *(*,*,*) になっているのは通常のセル点またはポインタ点（点ラベルを 'R' に定めてある）のいずれかの可能性があるからであり、その CAR部に相当する点１がセル点であることから左括弧で始まることを示している。ERG の点２が孤立点として残留しているのは、後述する共有点の可能性を否定できないからである。以上から、仮に３引数を順にｘ、ｙ、ｚとすれば、同図の MRG は式 '(C x y z' に、ERG は式 '(x z y' に、それぞれ対応しており、Ｃ組合せ子の簡約規則に一致している。なお、左括弧が前置されているのは（同一レベルでの）組合せ項の並びの最左端にあるレデックスを探索するためであり、次節で述べられる括弧の取り扱いにおける錯誤を防止する意味合いがある。

3.2　括弧の取り扱い

組合せ論理式の括弧に関しては、例えば '(((AB)C)D)' は 'ABCD' のように省略してよい習慣がある。このことは一見単純な事柄のようでありながら初心者には誤りをおこしやすい。次の例を参照されたい。

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

上記は、Ｃ組合せ子に相当する振る舞いをする組合せ論理式 S(BBS)(KK) の簡約過程を示したものであり、左側の式は省略可能な括弧をすべて省略した表現形である。そして下線部は簡約が施されるレデックスである。この場合、４行目の左側の式 B(Sx)Kyz を一瞥すると B(Sx)Ky 以外に Kyz もレデックスであるかのように見受けられる。ところが実際には後者はレデックスではない、というのは元来存在している括弧をすべて付加すると B(Sx)Kyz は ((((B(Sx))K)y)z) となり、そこには Kyz に相当する括弧の非省略形 ((Ky)z) が含まれていないからである。従って、組合せ論理式を簡約する際には常に括弧の非省略形を想起しながら行なう必要がある。ただし、幸いなことに計算機上では常に括弧の非省略形に相当するデータ構造を対象としているので、同一レベルの組合せ項の並びでの最左端に位置する組合せ子に関するレデックスに着目すれば、前述のような錯誤は生じない筈である。ところで一般に、組合せ子に関する簡約を適用すると、その直後では上記の右側の式から明らかなように冗長な括弧が残される場合が殆どである。これら冗長な括弧が含まれたままの式でレデックスを探すことは起こり得るレデックス形の場合分けが煩雑になり現実的ではない。それゆえ、冗長括弧を除去する操作は、処理効率の観点からも、組合せ子の簡約に関するグラフ書き換え規則の記述を簡単にする観点からも望まれることである。冗長括弧を除去する操作として用意された変換規則は、次の２つである。

．．．（Ａ）．．． ==> ．．．Ａ．．．
((Ａ．．．）Ｂ．．．） ==> （Ａ．．．Ｂ．．．）

ＧＭＬのグラフの書き換え方式でこれら２つの変換規則を記述するには、１行目の規則に関してはグラフ書き換え規則が１つだけで済むが、２行目のものに対してはいくつか必要となる。というのは、“レデックス”（矢印の左側）の形が組合せ項の任意長の並びから構成されているので、当該レデックスの全体形を１つのマッチング・グラフとして表現することが出来ないため、レデックスの候補対象を局所的に辿りながらレデックスの境界を認識する操作として表現するからである。本稿では省略するが、実際、上記２行目の変換規則は５つの書き換え規則で記述してある。

3.3　共有部分の取り扱い

組合せ子によっては、簡約後の形が同一引数を複数個含むものがある。例えばＳ組合せ子の簡約では、引数

zが２個含まれる形に変換される。この変換をグラフの書き換えとして表現する場合、引数zに対応する点の取り扱いに関して、それを共有する（具体的には１つの点を必要なだけ CAR 辺で共有する）方式とそれを必要なだけ複製する方式とが考えられる。Ｓ組合せ子の簡約に関するグラフ書き換え規則は、共有方式が図２に、複製方式が図３に、それぞれ示されている。ただし、複製方式の MRG は共有方式のそれと同一ゆえ省略してある。図２の場合、MRG の点２は 'S' に相当するアトムであり、点４、６、８はそれぞれ引数ｘ、ｙ、ｚに対応している。ERG では引数zに対応する点８が点３および７からの２つの CAR 辺で共有されている。ERG のみに存在する点11、12はグラフの書き換えで付加されるものであり、とくに点12は空アトムである。同図の MRG は式 '(Sxyz' を、ERG は式 'xz(yz)' を、それぞれ表わしており、確かにＳ組合せ子の簡約を実現する。一方、図３では引数zに対応する点８が２箇所に記述されていることでzが２回登場することが分かる。これは１つの点がグラフの書き換えにより複数個の点に分離（あるいは複製）される現象であることを意味しており、ＧＭＬでは複製されるべき点（本例では点８）に付随する辺が存在するならば、それらも（特に指定がない限り）書き換えにより複製される。例えば複製されるべき点がセル点であれば、その CAR 辺および CDR 辺も同時に複製されるので、結果としてそれらの辺により CAR 部および CDR 部が共有されることになる。即ち、一見したところでは共有化の問題が生じないように思える図３の場合でも、“水面下”には共有部分が生息している可能性があるのである。従って、共有部分の取り扱いは、いずれの記述方式を採用するにしても常につきまとう問題である。以下に共有部分が存在する場合に特に問題になることを指摘する。

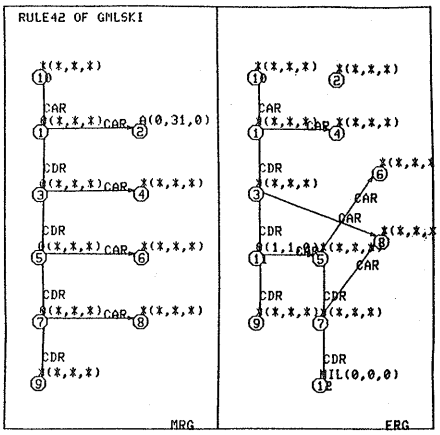

図２．　共有方式のＳ組合せ子

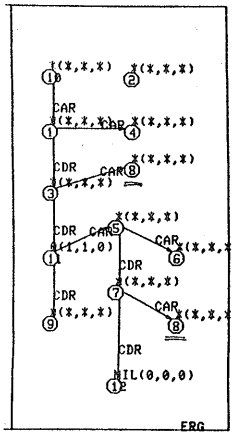

図３．　複製方式

(1) 点（および辺）の消去を慎重に行なう必要があること。

この事実は、既に示されたように、組合せ子に相当するアトム点が書き換え後も残されていることに典型的に反映している。特にＫ組合せ子の簡約の際には、状況によってはかなり大きな構造物に関する消去を取り扱うことになる。これらの被共有候補の構造物は、共有されていない場合には廃棄物になるので、なるべく簡約毎にそれらの有無を点検して適宜除去しなければならない。（実際、試作された簡約系では廃棄物を除去する処理が２つの書き換え規則で記述されている。）

(2) 一般にレデックス用の MRGが同一規則に対して何種類も必要になること。

図３のような複製方式による書き換えを用いる場合には、少なくとも試作された組合せ子簡約系に関する限り、各組合せ子の簡約におけるレデックス用の MRGは１種類だけで済むので、この問題は生じなかった。この事実は、複製方式の大きな利点である。しかるに、図２のような共有方式による書き換えを採用する場合には、組合せ子や引数群の共有関係が特に生じやすくなるので、多引数組合せ子のレデックスになるほど起こり得る共有辺の配置に関する場合分けが増すことになり、このままでは１つの簡約規則のレデックスに対して何通りもの MRGを用意する必要がある。このことを避けるために、次のような方策が採られた。組合せ子とそれが必要とする個数だけの引数を含む骨組み形（これをαとする）でマッチングを試み、不成功の場合には当該組合せ子のレデックスに関する探索は不在として打ち切られる。成功の場合には先ず当該組合せ子のレデックスでの不要な共有辺が全く存在しない形（これをβとする）でマッチングを試み、それが成功すればあとは当該組合せ子の簡約に関する書き換えを遂行すればよい。βでのマッチングが不成功の場合には、αのマッチング成功で指定された箇所において、共有辺を見つけては冗長括弧を付与することで見掛け上共有辺をなくす書き換え操作を反復していき、そのあとは再度βでマッチングを行なえば成功裡に進行する筈である。以上の方策を要約すれば、不要な共有辺を含まぬ形で先ず試み、それが不調に終われば次は前処理的に共有辺を（見掛け上）除去したあとで、再び不要な共有辺を含まぬ形で試みる、というものである。この方策の利点は、不要共有辺の除去という前処理が組合せ子の種類によらず共通であること、実質的に各組合せ子の MRGはαおよびβの２種類だけを用意すればよいこと、等である。

(3) 簡約を連続的に実行させることが（何も手当てをしないと）行き詰まること。

共有辺を生じやすいS組合せ子やY組合せ子の簡約を含む簡約系列を連続的に適用するうちに、予期し難い共有部分を含む構造物が形成されることがあり、尚も簡約を続行しようとすると実行が不可能になるか、あるいは誤りを引き起こすことがある。この対策を直観的に述べると、共有部分を複製して共有辺を解除することである。特に組合せ子の簡約に関しては、組合せ子アトムを CAR部とするセル点が（ CAR辺で）共有されている場合が“危険形”であり、この形が見つかり次第速やかに共有辺を解除するための複製処理が必要となる。試作された組合せ子簡約系では、この複製処理がグラフ書き換え規則を10個近く用いて記述されている。

3.4　レデックスの探索順

組合せ論理式を簡約するべくレデックスを探索する場合には、レデックス候補が複数個存在するときにどれを選択すべきかという問題がある。この問題に関しては従来から種々研究されてきており、例えば並行外側規則は安全であるとか、勝手に選択すると簡約が（ループに陥って）終了しないことがあるとかである。ところで、GMLのマッチング機能はマッチング可能な候補箇所が複数個存在し得るときにどれが選択されるかは恣意的に行なわれる（というよりは何らの規約がない）ので、GMLの利用者が積極的に“文脈情報”を MRGに添えない限り、例えば常に最左端のレデックスを求めるようなことは保証されないことになる。試作された組合せ子簡約系では、3.2 で言及した括弧の取り扱いに関連する誤動作を防止するために、左括弧という“文脈情報”だけが付加されている。従って、同一レベルの組合せ項の並びでは最左端のレデックスを選択するものの、構造全体での最左端のレデックスに相当することの保証はされていないことになる。敢てそのようにしたのは、文脈情報を種々付加することは処理効率が悪くなること、当面はGMLによる組合せ子の簡約が遂行できることを原理的に確認できればよいこと、等の理由による。

3.5　リスト構造のグラフ表現から記号列への変換

組合せ論理式のグラフ表現を、グラフの書き換えで加工することにより簡約過程を視覚的に把握できることは、GMLで実現することの大きな長所である。しかしながら、グラフが巨大化するにつれて組合せ論理式としての読み取りが煩雑になるのみならず誤解しやすくなるのは否めない事実である。そこで、リスト構造に相当するグラフから対応する記号列（本例では組合せ論理式）に変換するルーチン（これを逆変換ルーチンと称する）を用意することにし、やはりGMLで実現したが、グラフ書き換え規則は３つだけで済ませてある。その際、共有構造のグラフが存在している箇所では、記号列での当該共有構造に相当する部分式の前後を '*'と '#'で囲むことで、その旨の表示をするようにした。例えば、図４のグラフはアトム点 'F'と部分グラフ '(S B (K (S I I)))' がそれぞれ共有されているが、これを逆変換ルーチンにより表示すると、 (*(S B (K (S I I)))# *F# (*(S B (K (S I I)))# *F#)) となる。

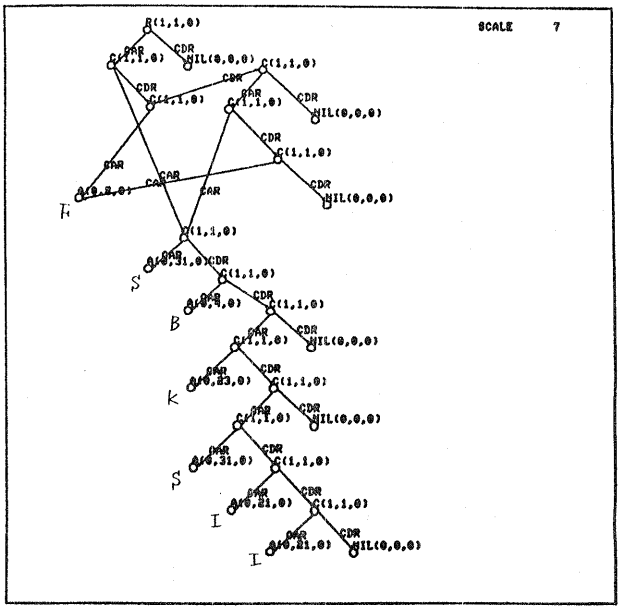

図４．　共有部分を含む構造例

3.6　組合せ子簡約系の処理手順

今まで述べて来たことをもとに、試作された組合せ子簡約系の処理手順を以下に概観する。

(1) 組合せ論理式の初期入力の記号列を、対応するリスト構造に基くグラフ表現に変換する。
(2) 組合せ子簡約に関するレデックスを探索し、見つかれば(3) に、見つからなければ(4) に、それぞれ進む。
(3) 当該組合せ子の簡約を行ない、廃棄物処理を試行し（逆変換ルーチンで記号列を表示して）(2) に進む。
(4) 冗長括弧が存在すればそれらを除去したあと、(2) に進む。
(5) “危険形”の共有構造が存在すれば複製により共有辺を解除したあと、(2) に進む。
(6) 上記の(2) から(5) までの間で何もすることがなくなれば終了する。

4．　関数型プログラムの組合せ論理式による表現

関数型プログラムは、基本的には、関数名と引数（あるいはオペレータとオペランド）という関係の累積物が、括弧を適宜用いることで表現されている。このとき、多引数の１階関数が１引数の高階関数に帰着できるという Curry　変換操作を適用すると、関数型プログラムは高々１引数関数の組合せで記述できる（グラフ上では２項リストあるいは２進木として表現できる）ので、ラムダ計算におけるアプリケーションおよびアブストラクションの概念と極めて相性が良くなる。即ち、１引数関数の形はアプリケーションそのものであり、高階化はアブストラクションと密接に関連する。一方、ラムダ計算と組合せ論理とが（条件付きで）等価になることは周知の事実である。以上の状況を勘案すれば、関数型プログラムを組合せ論理の土俵で処理することの見通しが得られることになる。実際、例えばTurner[4] は、関数型言語 SASL で記述されたプログラムを組合せ子のコード（即ち、組合せ論理式）に変換（これを彼は“コンパイル”と称した）し、それを組合せ論理に基く簡約系により“実行”させる方式について報告している。この方式は、試作された組合せ子簡約系を十分に活用することが期待できるので実験を行なうことにした。上述のように関数型プログラムがグラフ表現に適している事実に着目して、本方式による処理系の一切をＧＭＬにより実現してある。以降では、関数型プログラムを組合せ論理式に変換する流儀について簡単に触れたあと、その“コンパイラ”をＧＭＬで実現した方法について述べる。

4.1　関数型言語表現の組合せ論理式表現への変換

最初に、関数型言語を定める必要があるが、ささやかな実験ゆえ、次のような極めてありふれた関数群から構成される小さな言語を用意した。その言語の“名前は未だない”ので、仮に“猫”のラテン語表現である Feles　にしておく。

定義演算	Def
四則演算（いずれも２引数関数）	Plus,Minus,Times,Divide （左から順に整数に関する加減乗除）
比較演算（いずれも２引数関数）	Eq,Neq,Lt,Gt（結果は真または偽の論理値）
条件演算（３引数関数）	Qcond

ここで、定義演算の詳細形を非括弧表現で示すと、Def e1[= e2] [Where e3[= e4]];のようになる。（ [ ]は省略されることがある部分である。）今回の実験では、関数型言語の非括弧表現ではなくて、既に括弧表現されたものをコンパイラの入力対象と想定してあるので、非括弧表現から（Curry 化変換での）括弧表現までは飽くまで“手コンパイラ”の役割である。そして、条件演算 （Qcond e1 e2 e3） は通常の If e1 Then e2 Else e3 ; に対応している。式 e1 は比較演算または論理定数（真の 'T' か偽の 'F' ）でなければならない。なぜ 'Cond' ではなくてＱが前置されているかについて付言すると、既に述べたようにアトム点の点ラベルでは当該アトム名の先頭文字だけを抜き出して扱うように実現したので、 'Cond' や 'If' は組合せ子のＣやＩと同一視されてしまうためである。もちろん、例えばアトム名の綴り全体をハッシュ・コード化するように実現しておけばこの種の問題は生じない。

以上の準備のもとで、関数型言語から組合せ論理式への変換法について説明する。

(1)　与えられた関数型プログラムを、高々１引数関数の組合せとなるように、即ちCurry 化変換により、２項リストの括弧表現にする。例えば、後者関数の定義 Def pred x = x - 1 は Def pred x = ((Minus x) 1) のように考える。

(2)　定義式の左辺にある引数を“消去”するために、それを右辺に“移項”することで、アブストラクション操作を施せる形にする。例えば、 Def pred x = ((Minus x) 1) は Def pred = [x]((Minus x) 1) のようにする。ここで [x] はラムダ計算のλｘに相当するもので、その右に隣接する式内に出現する総ての当該変数（本例ではｘ）を抽出すること（アプリケーション）を意味する。この結果に当該変数でアプリケーション操作を施せば、もとの右辺の式になる。即ち、([x]((Minus x) 1))x = ((Minus x) 1) となる。

(3)　以下に示すようなアブストラクション操作を可能な限り施すことで、組合せ論理式（含まれる組合せ子はＳ、Ｋ、Ｉのみ）に変換する。

①　[x]（Ｅ１　Ｅ２）　==>　Ｓ（[x]Ｅ１）（[x]Ｅ２）

②　[x] x　==>　Ｉ

③　[x] y　==>　Ｋ　ｙ　　（ただしｙは定数またはｘ以外の変数）

例えば、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) となる。

(4)　得られた組合せ論理式コードを短縮する最適化を行なう。新たに組合せ子ＢおよびＣも用いられる。

① S（K E1）（K E2） ==> K（E1 E2）
② S（K E1）I ==> E1
③ S（K E1）E2 ==> B E1 E2
④ S E1（K E2） ==> C E1 E2

ここで、③は①、②が、④は①、②、③が、それぞれ適用不可能になったあとで初めて試みられる。
例えば、 S(S(K Minus)I)(K 1) ==> S(Minus)(K 1) ==> C (Minus) 1 ==> C Minus 1 となる。

4.2 関数型言語から組合せ論理式コードへのコンパイラ

前節に示した変換法の内で、(3) および (4)の変換に関しては、例えば (4)の①に関する変換規則が図５に示されるように、グラフの書き換えに適したものなので、ＧＭＬにより実現した。若干の工夫としては、例えば変数ｘと［ｘ］とをアトム点の点ラベルにおいて識別するために両者のコードの間に一定のバイアス値を設けたことで、それにより両方向への変換が可能となるようにしたことである。このことは、とくに (3)の②と③を区別する場合に極めて有効である。

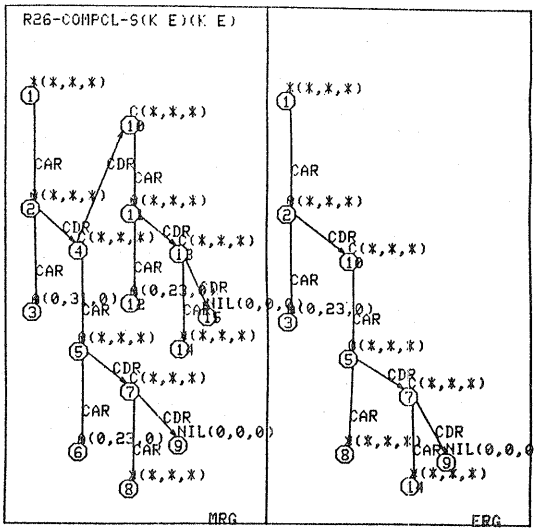

図５．最適化変換規則例

５． 再帰的プログラムの実行

関数型プログラムが組合せ論理式コードに“コンパイル”されたあと、それを“実行”するためには、基本的には組合せ子簡約系があればよい。しかしながら何らかの値を計算するプログラムの場合には、それだけでは不十分であることは言を待たない。それゆえ、算術演算や比較演算はそれぞれ最終的には整数値や論理値に変換される必要がある。また、条件演算は第１引数が論理値として確定されたあとでは、第２あるいは第３引数を選択しなければならない。これらの演算に関する関数形を認識して値を求める処理系を関数評価系と呼ぶことにする。以下では、関数評価系の概要、およびそれと組合せ子簡約系とを有機的に結合させて組合せ論理式コードを実行する方法について述べる。

先ず、関数評価系に関しては、各演算の関数形を認識して、値が求まるならば求めたあと、然るべき形に変換する操作に関して、同様にＧＭＬで実現した。変換規則は次のものである。

① Plus e1 e2 ==> e1 + e2
② Minus e1 e2 ==> e1 - e2
③ Times e1 e2 ==> e1 * e2
④ Divide e1 e2 ==> e1 / e2
⑤ Eq e1 e2 ==> If e1==e2 Then 'T' Else 'F'
⑥ Neq e1 e2 ==> If e1=/=e2 Then 'T' Else 'F'
⑦ Lt e1 e2 ==> If e1<e2 Then 'T' Else 'F'
⑧ Gt e1 e2 ==> If e1>e2 Then 'T' Else 'F'
⑨ Qcond 'T' e2 e3 ==> e2
⑩ Qcond 'F' e2 e3 ==> e3

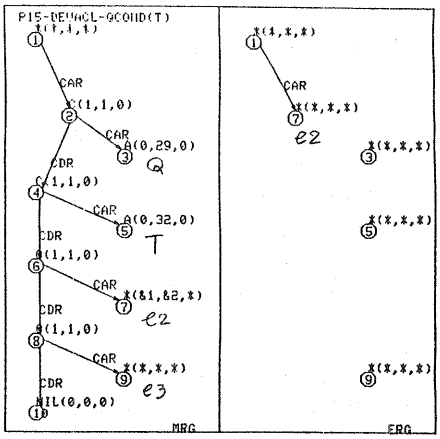

図６．条件演算（真の場合）

図６は上記⑨に関するＧＭＬの書き換え規則であるが、ここでは不要になったものは共有形でない限り除去するという破壊方式を採用している。（これとは対照的に、不要になった部分グラフの各点ラベルにその旨のタグを置く、非破壊方式も考えられる。）点７の点ラベル内の&1や&2はラベル変数であり、それらが置かれている箇所の値の保存や参照を可能にしている。本例では、e2に相当する点７がアトムかセルのいずれであるかをタグ（ラベル変数&1）で調べ、もしアトム点ならばその値（ラベル変数&2）

を求めるために、これらのラベル変数が用いられている。

次に、組合せ論理式の実行系について述べる。組合せ論理式コードを（入力データと共に）実行するには、非常に素朴なプログラムの場合ならば、先ず（当該コードと入力データとのアプリケーション形に）組合せ子簡約系を可能な限り施すことで組合せ子が一切含まれないコードに変換し、そのあと関数評価系で値を求めればよい。例えば、プログラム pred に入力データ '2'を与えて実行させる場合は、次の通りである。

(pred 2) = ((C Minus 1) 2) ==> (C Minus 1 2) ==> (Minus 2 1) ==> (2 - 1) ==> 1

ところが、毎度おなじみの階乗関数のような再帰的プログラムになると、事態は複雑になる。とりあえずは、関数型言語 Feles による階乗関数の定義を与えよう。

Def Fact x = Qcond (Eq 0 x) 1 (Times x (Fact (Minus x 1)))

これにアブストラクション操作とCurry 化操作とを適用すると次のようになる。

Def Fact = [x](((Qcond ((Eq 0) x)) 1) ((Times x) (Fact ((Minus x) 1))))

この右辺のコンパイル結果をαとし、それのCurry 化変換形をα'とする。更に（[Fact] α'）としてコンパイルすると次のようなコードβが得られる。

β = (B (S (C (B Qcond (Eq 0)) 1)) (B (S Times) (C B (C Minus 1))))

かくして、再帰的プログラム Fact は (Y β) として得られることになる。そして、例えば３の階乗は (Fact 3) = ((Y β) 3) = (Y β 3) を実行すればよい。ここで浮上する問題は、このコードを実行するに際し、不注意に組合せ子簡約系を用いると、Y組合せ子に関する簡約が常に適用可能となっているために、ループに陥る可能性があることである。また、Turner [4]に述べられているようなスタック機構の助けを借りなければ再帰的プログラムの実行は不可能なのかという問題もある。以上の問題点を検討するべく、次のような実行方式を試験することにした。

(1) 組合せ子簡約系を適用して、予め指定された回数の簡約を実施させる。この際、Y組合せ子に関する簡約は、他の組合せ子での簡約や冗長括弧の除去等の操作が一切不可能である場合に初めて試みる。

(2) 破壊方式による関数評価系を可能な限り適用していき、ポインタ点の CAR部がアトム点になれば、その点ラベル内の値が求めるものであり、実行を終了する。それ以外のときは、(1) に進む。

(3) 上記 (1)および (2)が（同時に）何の操作も行なわないならば、実行を打ち切る。

以上の方式により、実際に３の階乗を ([Fact](Fact 3))Fact ==> (C I 3)(Y β) ==> (C I 3 (Y β)) により実行させたところ、スタック機構を全く用いずに値６として評価できた。その他、フィボナッチ関数やMc Carthyの91関数の再帰的プログラム等についても、上記の方式により首尾よく値を求めることができた。これらの事実から、上記の方式による再帰的プログラムの実行法は妥当なものと言えよう。

６．　おわりに

グラフ処理言語ＧＭＬにより、視覚的に簡約過程を把握できる組合せ子簡約系を実現し、いわゆるグラフ簡約の効果を直視できた。遺憾ながら、グラフの規模が大きくなるにつれて組合せ論理式としての読み取りが厄介になりがちであるが、グラフから組合せ論理式の記号列を求める“逆変換”ルーチンをもＧＭＬで実現したことは、ＧＭＬ本来の主旨からすればやや不本意な用法であるものの、その広範囲な能力を実証したとも言えよう。そして、やはりＧＭＬで実現された関数型言語から組合せ論理式コードへのコンパイラおよび関数評価系との有機的連係により組合せ子簡約系を有効利用する試みである関数型言語 Feles の処理系は、再帰的プログラムに対しても、実行効率には問題があるものの、スタック等を全く導入せずに実行できることを確認できた。今後の課題としては、手コンパイラの出番を減らすこと、組合せ論理式コードの最適化の検討（一例としてHughes[5] の“超組合せ子”）、データ構造を“値”とする場合への適用、等がある。

参考文献

[1] 片山、二木、他：“特集：非手続き型プログラミングのための計算モデル”、情報処理、24、No.2、1983

[2] 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.427-434,1974

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

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

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