知識工学と人工知能　52－6
（１９８７．５．７）

手続きグラフを用いた例題からのプログラム合成

石橋　勇人，西田　豊明，堂下　修司
京都大学　工学部　情報工学教室

複雑なプログラムの合成を行うことを考えた場合、知識ベースの利用は不可欠であると考えられる。しかし、その場合であってもプログラムの低レベルの部分を補う合成手段として、知識にたよることなく一からプログラムを合成できる能力を持つことが必要である。

本論文では、知識ベースに基づくプログラム合成法の基本的枠組みとなる例題からのプログラム合成法について述べる。本合成法では、まず、例題から手続きグラフを作成し、これを一般化することによってプログラムを合成する。手続きグラフを手続きブロックの連結と捉えることにより、知識ベースに基づく合成と基づかない合成を同一の枠組みによって取り扱うことができる。

Program Synthesis from Examples with Procedural Graphs

Hayato ISHIBASHI, Toyoaki NISHIDA and Shuji DOSHITA
Dept. of Information Science, Kyoto University
Kyoto 606, JAPAN

There is no doubt of the necessity of a knowledge-base to synthesize complex programs. Even in the case, the system should have the ability to synthesize programs without knowledges to build the lower part of programs.

In this paper, we describe a program synthesis method from examples which is not based on knowledge-bases, but which makes the framework of knowledge-based synthesis methods. Our method consists of two parts; a given example is described as a procedural graph at the first step, and then it is generalized and comes to a program. This method makes it possible to treat knowledge-based and non knowledge-based program synthesis in the same framework by considering a procedural graph as a connection of procedural blocks.

1．はじめに

プログラム合成には、全く一からプログラムを合成していく方法と、知識ベースにあらかじめプログラム（の一部）を蓄えておき、その組合せによってプログラムを組み立てる方法とが考えられる。

複雑なプログラムを合成するためには知識の利用は不可欠である。しかし、知識ベースに基づいてプログラムを合成する方法では、１）知識ベースに登録されていないプログラムの要素を必要とする場合に対処することができない。２）ある要素から別の要素に対してデータを渡す場合に、要素間のインターフェースをとる必要がある。といった点で知識では解決できない問題を含んでおり、知識ベースに基づいてプログラムを合成するアプローチにおいても知識ベースに登録されている要素を有効に活用できるためには、一からプログラムを合成できる能力を持つことも必要である。

このような点を踏まえて、本論文では知識ベースに基づいたプログラム合成の基本的枠組みとすることを考慮した一からのプログラム合成の方法について述べる。この合成法では、基本的には知識を利用せずにプログラムを合成するが、そこで用いている手法を拡張することによって知識ベースに基づいた合成を自然な形で取り込むことができる。

2．プログラム合成システムＰＲＯＢＥ

2．1　例題からのプログラム合成

プログラム合成システムに対する仕様の与え方のうち、一般ユーザにとって最も自然なものの１つが例題によって仕様を与える方法である。例題からのプログラム合成では、例題の形で与えられたインフォーマルな情報に暗に含まれる対応関係をもとにして、そこから必要な処理を抽出し、一般化することによってプログラムを生成する。

従来の例題からのプログラム合成の代表的なものとして、Hardy[2],Summers[4]などがある。Hardyの手法では、あらかじめプログラムの雛型（プログラムスキーマ）を用意しておき、そのパラメータを例題から決定することによってプログラムを合成していた。この方法では、用意するプログラムスキーマによってシステムの能力が限定されてしまう。

Summers の手法では、与えられた複数の例題間の差分を取ることによって例題の変化のパターンを検出し、それを外挿することによって一般化を行うことでプログラムを合成していた。この方法では、一度生成されたプログラムに対して後から手を加えて修正することは困難である。これは、パターンマッチングからプログラムへの過程が非常に緊密であるために、その途中から新しい情報を加えることが困難であることによる。しかし、例題というインフォーマルな仕様を用いている以上、ユーザの望む仕様を完全に満たすプログラムを１度で合成することは難しく、何等かの形で修正が必要となることは明らかである。従って、例題から得られたプログラムが適切でなかった場合には、適宜情報を追加することによって、生成されるプログラムがインクリメンタルに望むプログラムに近付いていくような性質をシステムが持つことが望まれる。このためには、例題から合成結果（プログラム）に至る過程をシステム自身が把握し、必要に応じて合成されたプログラムを改変できなければならない。

従来のこのような合成法一般に言えるのは、あまり複雑なプログラムが合成できないということである。この最も大きな原因としては、知識の欠如があげられる。プログラム合成の基礎をなす部分として知識を用いない合成は重要であるが、それが知識を用いた合成法とうまく融合できなければ複雑なプログラムの合成は困難である。

本論文において述べるプログラム合成システムＰＲＯＢＥでは、これらの問題を解決するために、ユーザによって与えられた例題から「手続きグラフ」を生成し、次に手続きグラフを一般化することによってプログラムを合成する。手続きグラフは、詳しくは３章で述べるが、例題として与えられた入出力間の関係（入力から出力が構成される過程）のフローを表現したグラフであり、入出力の対として静的に与えられた例題を動的に捉え直したものである。手続きグラフは、一般化の結果（プログラム）に対しても一般化する前と同様に自由に変形を加えることができるので、Summersのような修正の問題は生じない。

また、後述のようにＰＲＯＢＥでは手続きグラフを「手続きブロック」と呼ぶ機能ブロックの集合として捉えている。手続きブロックは処理の単位としてブラックボックス視することができるので、あらかじめ手続きブロックを知識ベースに登録しておき、その組合せによって手続きグラフを構成することができる。この場合であっても、手続きグラフの生成以後の処理は同様に進めることが可能である。つまり、手続きブロックの概念によって、知識ベースに基づく合成と、一からのプログラム合成とを統一的に取り扱うことが可能となっている。

２．２　システム概要

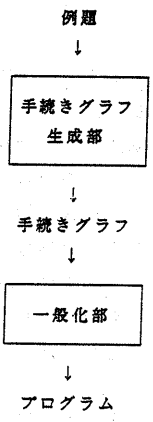

図２．１　ＰＲＯＢＥの基本構成

プログラム合成システムＰＲＯＢＥの基本構成を図２．１に示す。ＰＲＯＢＥは、まず、手続きグラフ生成部によってユーザに与えられた例題を解析し、少なくとも例題を満たす解（＝手続き）を導く。これは、与えられた特定の例題に関してのみ結果が保証されるような解、すなわち特殊解である。次に、一般化部がこの特殊解を一般化することによって、任意の入力に対して正しい出力、すなわち例題を与えたユーザが意図した仕様を満たす解を導く。この解（一般解）が合成結果としてのプログラムとなる。

ここで、次のような例題を考える。

例題：　((A (B)) (C D) E) → (A B C D E)

これは、ネストしたリスト構造をフラットにするプログラム、すなわち、２進木の葉の部分を集めてくるプログラムを合成する例である。

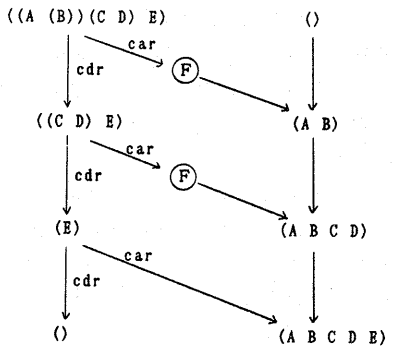

図２．２　手続きグラフ

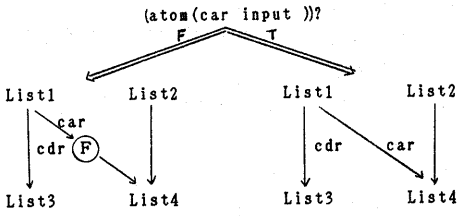

図２．３　縮約された手続きグラフ

```
F = (DO ((COUNT 0 (1+ COUNT)))
        ((= COUNT (LENGTH INPUT)) LIST4)
        (COND ((ATOM (CAR LIST1))
               (SETQ LIST3 (CDR LIST1))
               (SETQ LIST4 (APPEND1 LIST2 (CAR LIST1))))
              (T (SETQ LIST3 (CDR LIST1))
                 (SETQ LIST4 (APPEND LIST2 (F (CAR LIST1))))))
        (SETQ LIST1 LIST3)
        (SETQ LIST2 LIST4))
```

図２．４　合成されたプログラム

この例題をＰＲＯＢＥに与えると、手続きグラフ生成部により特殊解として図２．２のような手続きグラフが得られる。ここで、○に囲まれたＦというノードを関数ノードと呼び、Ｆに相当する部分は別の手続きグラフとして表現される。すなわち、一般的には別の関数として合成されることになる。この場合にはＦにあたる手続きグラフは図２．２の手続きグラフ自身である（つまり、再帰呼び出しとなっている）。

次に一般化部によって手続きグラフの一般化を行う。図２．２の手続きグラフは破線で示したように３つのサブグラフに分割することができる。このサブグラフのおのおのを手続きブロックと呼ぶ。手続きブロックを単位として手続きグラフを縮約することにより図２．３が得られる。これが一般化した結果であり、プログラムに相当する。このグラフに対応するLispコードを図２．４に示す。

本論文では、ＰＲＯＢＥの基本部分、すなわち、知識に基づかずに一からプログラムを合成する部分について、手続きグラフの一般化アルゴリズムを中心に述べる。手続きグラフ生成部については、機会を改めて述べることにする。

現在は、手続きグラフ生成部の負担を軽減するため、タスク領域としてはリストの構造変換プログラムを対象としている。また、例題とは合成しようとするプログラムに対する入出力対を意味している。なお、合成されるプログラムを記述する言語としてはLispを用いている。

以下では、３章において手続きグラフ及び手続きブロックの概念について述べ、４章で手続きグラフの一般化アルゴリズムについて述べる。これに続いて５章、６章ではそれぞれ知識の利用、プログラムの修正について検討を加える。

３．手続きグラフ

３．１　手続きグラフ

プログラム合成システムＰＲＯＢＥでは、まず、例題として与えられた入出力対を解析し、入力から出力を得るために必要な処理の系列を発見する。この処理系列は、「手続きグラフ(procedural graph)」の形で

表現される。手続きグラフは、例題の処理の過程を示すトレースに対して、各過程において適用される処理（関数）を付加することによって、データの受ける作用を明示的に示したものである。

手続きグラフは、例題の入出力関係を満たす１つの特殊解としての処理を１種のデータフローグラフとして表現したものであり、内部状態（内部記憶）を持たない。また、手続きグラフは、特定のプログラミング言語や特定のドメインに依存するものではないので、本論文で用いている一般化手法は、基本的に、言語やドメインに独立のものである。さらに、一般化手続きによって一般化された手続きグラフは、一般解、すなわち、プログラムを表現するものとなる。従って、手続きグラフを用いることによって、特殊解と一般解の両方を特定の言語やドメインに独立した形で記述することができる。

手続きグラフの一例として、

例題：｛入力，出力｝

＝｛(A B C D),((A)(B)(C)(D))｝

に対するものを図３．１に示す。図３．１において、左端の縦１列が例題として与えられた入力リストの変化する過程を表し、右端の縦１列が出力リストが構成されていく過程を表している。これらを、それぞれ入力系列、出力系列と呼ぶ。

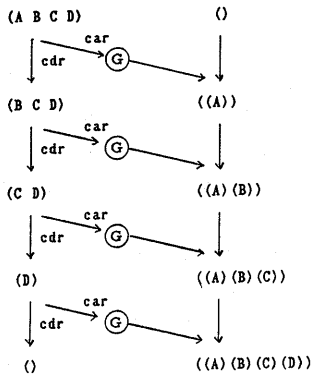

図３．１　手続きグラフ

手続きグラフにおいてノードとなるものには、操作を受けるデータ（リスト）をノードの値として持つものと、関数をノードの値として持つものとがある。前者をデータノード、後者を関数ノードと呼ぶ。図３．１において、○で表されているのが関数ノードである。関数ノードとして現れる関数は、システムにあらかじめ用意されているプリミティブ関数ではなく、別の手続きグラフ（図３．２）として定義され、合成される。すなわち、関数ノードによって手続きグラフのネスティングが実現されている。

図３．２　関数ノードＧに対する手続きグラフ

３．２　手続きブロック

プログラム合成システムＰＲＯＢＥでは、手続きグラフの形に抽象化された例題をもとに、これを一般化することによってプログラムを合成する。手続きグラフの一般化の過程の中心は、ループと条件分岐の形成である。このために、手続きグラフの比較、縮約等の操作をおこなう必要があるが、このような操作を手続きグラフそのものに対して行うのは、非常に手間がかかる。その代わりに、手続きグラフを適当なサブグラフに分割し、そのサブグラフを単位として取り扱うことによって処理が容易になる。

手続きグラフにおける入力系列と出力系列は、基本的に協調して処理のステップとともに進行する。すなわち、典型的には入力系列について１ノード分の処理が進むと、出力系列上に１ノードの結果が得られる。入力系列上のｎノード（ｎ≧２）に対して出力系列上の１ノードが対応するような場合もあるが、入力系列、あるいは出力系列の片方が一方的に進行するようなことはない。この点に着目すると、手続きグラフを、それぞれが処理の単位となるようなサブグラフに分割することが可能である。例えば、図３．１に示した手続きグラフの場合には、破線で示したように分割することができる。この分割された各々のサブグラフのことを、「手続きブロック(procedural block)」と呼ぶ。手続きブロックの概念を用いると、手続きグラフは、いくつかの手続きブロックを連結したものとして捉えることができる。

手続きブロックは、以下のように定義する。

① 手続きブロックは、手続きグラフにおいて入力系列上に存在するノード及び出力系列上に存在するノードを少なくとも２つずつ持つ。

② 手続きブロックは、単独で閉じている。すなわち、手続きブロックの出力端子から手続きブロックの外へと出ていくアークが、手続きブロック内に存在するノードに接続されることがない。

③ ①、②を満たす最小のサブグラフを手続きブロックとする。

図３．３に手続きブロックの例、及び手続きブロックでない例を示す。

４．手続きグラフの一般化

ここでは、ＰＲＯＢＥの基本部分のうち手続きグラフの一般化過程を順を追って述べる。

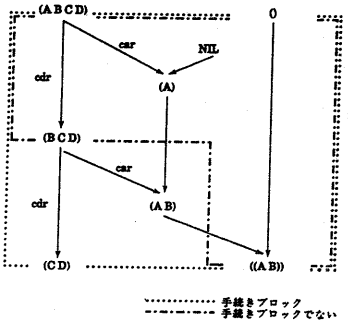

図３．３　手続きブロックの例とそうでない例

４．１　手続きブロックの切り出し

一般化の過程においては、手続きグラフを手続きブロックを単位として取り扱う。従って、まず、手続きグラフを手続きブロックに分割する。

先の定義に従って切り出される手続きブロックは、閉じている最小のものであるから、手続きグラフを取り扱う際にこれ以上小さな単位を考慮する必要はない。

逆に、上の定義によって切り出されるよりも大きなサブグラフを手続きブロックとした方が、プログラム中のループの１ステップという意味では正しい場合がある。このときでも、上の定義による手続きブロックは、最小の単位であるから、“大きなサブグラフ”は手続きブロックの連結として表される。従って、このような場合には必要なだけの連続した手続きブロックを１つとして扱うことによって、手続きブロックが小さ過ぎるという問題を解決することができる。実際には、手続きグラフの柔軟な扱いを可能とするために、手続きブロックを実際に併合することは行わず、次に述べる手続きグラフの縮約の過程において連続する手続きブロックが本来１つのものである可能性を考慮する。

４．２　手続きグラフの縮約

手続きグラフ生成部において例題より得られた手続きグラフは、一般に特定の例題のみに有効な手続き（プログラム）を表している。ここから、例題として与えられた入力データに依存した特定回数の繰り返しの処理を抽出し、ループによって置き換えることによって、繰り返し回数から特定の入力に対する依存性を排除し、一般化することができる。

ＰＲＯＢＥでは、この一般化操作を、手続きブロックを単位として手続きグラフを縮約することによって行う。手続きグラフを縮約することによって同型の処理がループに置き換えられ、これによって一般化が行われたことになる。

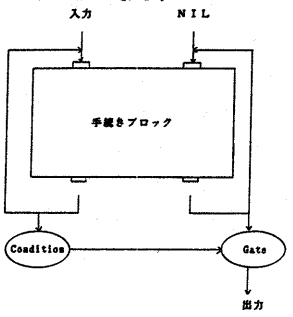

図４．１　プログラムモデル

縮約操作後、手続きグラフ全体が１つの手続きブロックに縮約されたとすると、プログラムは図４．１の様にモデル化される。すなわち、このプログラムに対する入力を入力系列の初期値、ＮＩＬ（空リスト）を出力系列に対する初期値として図４．１のモデルに与えることによって、１サイクル毎に新しい入力系列及び出力系列の要素が生成され、それらが次の入力及び出力としてフィードバックされる。ある時点においてループから脱出するための条件が満たされると、その時点での出力系列の要素がプログラムの最終的な出力としてプログラムの外部へと取り出される。

４．３　縮約のアルゴリズム

手続きグラフの縮約とは、手続きグラフ中の同型の処理を抽出して１つにまとめることであるが、同型の処理の検出は、手続きブロック間のパターンマッチングによって行う。連続するいくつかの手続きブロックがすべてマッチする場合、それらは縮約され、１つのブロック＋ループとなる。図３．１の場合、４つの手続きブロックから手続きグラフが構成されているが、すべての手続きブロックがマッチする（マッチング規則については次節において述べる）。

また、いくつかの連続する手続きブロックを１つのグループとしてみた場合に、マッチするグループが連続する場合にも、このグループを単位として縮約される。この縮約規則によって、先に述べた手続きブロックの拡大に相当する結果を得ることができる。すなわち、いくつかの連続する手続きブロックを１つの手続きブロックにまとめたのと同じ効果が得られる。ただし、グループがマッチするとは、グループ内の対応する位置にある手続きブロック同士がすべてマッチすることを意味する。

４．４　手続きブロックのマッチング

手続きブロックのマッチング規則は、以下の通りである。

①　２つの手続きブロックの内部構造が等しい場合にこれらの手続きブロックはマッチする。但し、

内部構造が等しいとは、手続きブロックのグラフとしての構造が一致し、かつ、各アークにつくラベルがすべて一致することである。

② 2つの手続きブロック間において、データノードのみをノードとして扱った場合のグラフ的構造が一致し、かつ、対応するアークの一方のみに関数ノードが含まれている場合、あるいは、対応するアークの対応する位置にある関数ノードが異なっている場合に、これら2つの手続きブロックは一致する。

規則①によって、完全に同一の構造をもつ処理をループとしてまとめることができる。また、規則②によって、制御の流れとしては同一であるが、データによって適用する処理が異なる場合をループの中に組み込むことができる。

規則②が適用された場合には、異なった関数ノードを持つアークに対応した部分に条件分岐が生成される。

4．5 複数引数のための手続きグラフ

これまでに現れた手続きグラフは1引数関数（手続き）に相当するもののみであったが、複数引数の関数複数引数の関数（手続き）を手続きグラフによって記述するために、入力系列上にあるノードに関してのみ多値を有することを許す。これによって、

例題：｛入力1，入力2，出力｝
＝{A,(B C D),((A B)(A C)(A D))}

のような2引数の場合であっても、図4．2のように手続きグラフによって記述することが可能となる。このように拡張した場合でも、基本的にはこれまでに述べた一般化手法がそのまま適用できる。先に述べたマッチング規則において、多値ノードに対しては同数の値を持つ多値ノードがマッチし、多値ノードから出ているアークについては、すべてのラベルが一致した場合にマッチすると解釈すればよい。図4．2を縮約した結果を図4．3に示す。

4．6 一般化の例

手続きグラフの一般化の例として、次のような例題を取り上げる。

例題： (A B C D) →
((A B)(A C)(A D)(B C)(B D)(C D))

これは、入力として与えられたリストの要素から2つを取る組合せを生成するプログラムを合成する例で

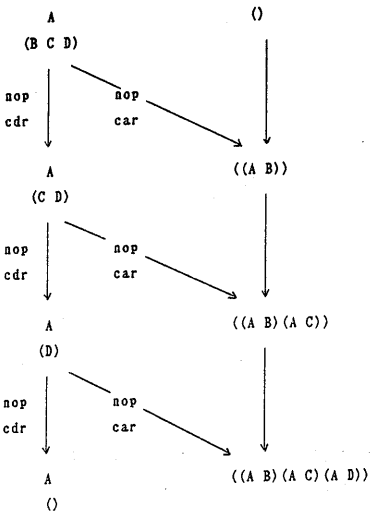

図4．2 2引数の手続きグラフ

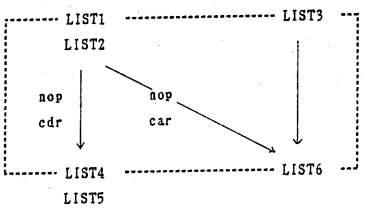

図4．3 縮約の結果

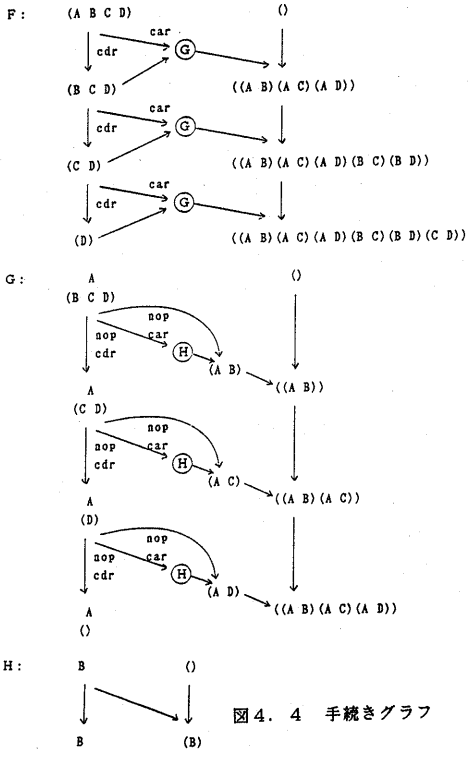

図4．4 手続きグラフ

ある。

この場合には手続きグラフは図４．４のようになる。また、手続きブロックへの分割は破線で示したようになる。これらの３つの手続きグラフのうち、関数Ｇに対応する手続きグラフが２つの引数をとる形となっている。

関数Ｆ、Ｇに対する手続きグラフは、いずれも全く同型の３つの手続きブロックからなっているので、縮約の結果、図４．５のように変形される。Ｈは１つの手続きブロックからなっており、そのままプログラムとなる。

関数Ｆ，Ｇ，Ｈに対応するLispコードは図４．６の通りである。

５．知識の利用

ＰＲＯＢＥにおける手続きグラフ生成、一般化の２つのフェーズのうち、知識の導入が効果的なのは、手続きグラフ生成フェーズである。手続きグラフ生成フェーズにおいては、例題を様々な角度から分析することによって、そこに含まれている情報を抽出し、入出力間の動的関係を推測して手続きグラフを生成する。このとき、その例題のドメインに関する知識をシステムが持っているならば、その知識に基づいて効果的な解析を行うことができる。すなわち、手続きグラフ生成部において手続きグラフを生成する際にドメインに関する知識を用いることにより、例題の持つセマンティクスを考慮して例題を解析することが可能となり、これによって新たな視点が得られるとともに、解析の際の探索空間を限定することができる。

例として、与えられる例題に含まれるリストが表を意味しており、表を操作するプログラムを合成するということがあらかじめわかっている場合を考える。この場合には、表に関する基本的操作を先に考慮する方が効果的である。また、操作が適用される位置についても、対象が表であれば、その行や欄についてある操作が適用されることが多いことから、行や欄を単位として例題を解析することによって効果的な解析ができる。

また、ドメインによって基本的と考えられる操作をシステムのプリミティブ関数として登録しておくことによって、そのドメインに属する問題の解決が、より容易なものとなる。

このように、ドメインに関する知識をうまく導入することによって、より高いレベルの視点から例題を見ることができるようになり、その結果、効果的な合成が可能となる。

これに対して、一般化フェーズでは、すでに例題が手続きグラフの形に抽象化されているため、ドメインに関する知識はあまり有効ではない。逆に、手続きグラフを用いることによってドメインに独立した形で一般化を行うことができる。

６．　プログラムの修正

例題からのプログラム合成の場合、例題は仕様としてはインフォーマルなものであるから、必ずしも目的とするプログラムが１度で得られるとは限らない。すなわち、与えられた例題に関しては正しく動作するが、他の入力に対してはユーザの意図に反する動作をするプログラムが合成されることがある。このような場合には、何らかの形でそのプログラムを修正する必要が生じる。プログラムを修正する方法として、

① 例題を追加することによって情報量を増す

② 合成されたプログラムコードをユーザが評価し、問題点を指摘する

の２通りが考えられる。後者は、前者に対して直接的な指摘が可能なため、効率のよい修正が可能である。しかし、この場合には、ユーザがプログラムに関する知識を持っていなければならないという点に問題がある。また、ある程度以上複雑なプログラムになると、その誤りを指摘するのは極めて困難である。従って、現実的にはあまり有効な手段とは言えない。

これに対して、前者は、例題を与えることのできるユーザであれば修正を行うことが可能であり、プログラムに関する知識を全く必要としないという利点がある。

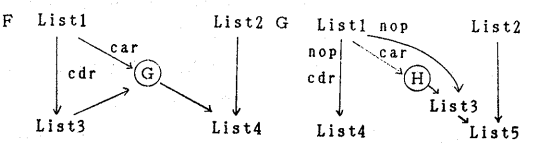

図４．５　Ｆ，Ｇを縮約した結果

```
F = (DO ((COUNT 0 (1+ COUNT)))
        ((= COUNT (1- (LENGTH INPUT))) LIST4)
        (SETQ LIST3 (CDR LIST1))
        (SETQ LIST4 (APPEND LIST2 (G (CAR LIST1 LIST3))))
        (SETQ LIST1 LIST3)
        (SETQ LIST2 LIST4))

G = (DO ((COUNT 0 (1+ COUNT)))
        ((= COUNT (LENGTH INPUT2)) LIST4)
        (SETQ LIST31 LIST11))
        (SETQ LIST32 (CDR LIST12))
        (SETQ LIST4 (CONS LIST11 (H (CAR LIST12))))
        (SETQ LIST11 LIST31)
        (SETQ LIST12 LIST32))

H = (CONS INPUT NIL)
```

図４．６　合成されたプログラム

図６．１は、(A (B C) D) → (A B D)という例題に対する手続きグラフ及びその縮約形（プログラム）である。ここでは、処理しようとする要素がアトムであるか否かによって処理が２つに分類されている。

ここで、修正のために ((A . B)(C D)) → (B C)という例題を追加入力すると、システムは図６．２のような手続きグラフを生成する。この後、この手続きグラフが一般化部に送られると、一般化部では先に得られた手続きグラフと新たに得られた手続きグラフからこれらを２つとも満足するような手続きグラフを作り出すことによってもとの手続きグラフを修正する。この際には、Andreae[1]にみられるようなグラフ間のマッチング手法を利用することができる。この場合には２つの手続きグラフをマージした結果、図６．３のような手続きグラフが得られる。

こうして修正を加えられた手続きグラフが新たなプログラムとなり、再びユーザに提示される。以下、このステップを必要なだけ繰り返せばよい。

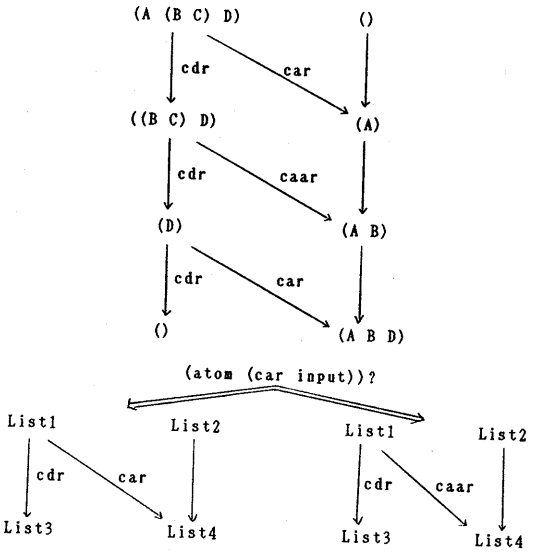

図６．１　手続きグラフとその縮約形

７．おわりに

本論文では、手続きグラフの一般化による例題からのプログラム合成法について、そこで用いる一般化法を中心に述べた。この方法によって文献[2],[3],[4]に掲げられているすべての例題に関して合成が可能なことが確認されている。

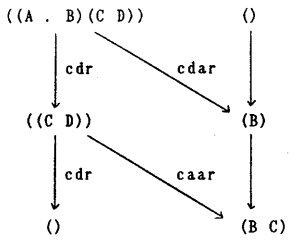

図６．２　追加される手続きグラフ

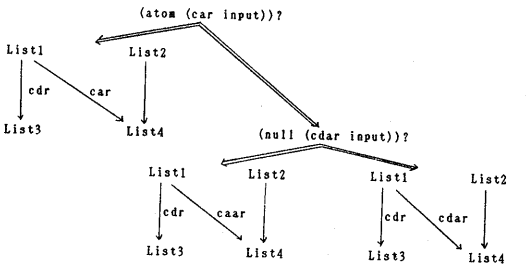

図６．３　マージ後の手続きグラフ

【参考文献】

[1] Andreae, P.M.: Constraint Limited Generalization: Acquiring Procedures From Examples, Proc. of AAAI-84 (1984)

[2] Hardy, S.: Synthesis of Lisp Functions from Examples, Proc. of IJCAI-75 (1975)

[3] Shaw, D.E. et al.: Inferring Lisp Programs from Examples, Proc. of IJCAI-75 (1975)

[4] Summers, P.D.: A Methodology for LISP Program Construction from Examples, J.ACM vol.24 no.1 (1977)

[5] 石橋：手続きグラフの一般化による例題からのプログラム合成，京都大学修士論文 (1987)

[6] 石橋，西田，堂下：手続きグラフの一般化による例題からのプログラム合成，情報処理学会第３４回全国大会予稿集，p.1449 (1987)