類推を用いた入出力例題からの論理プログラムの合成手法の提案†

小泉昌紀†† 永田守男†††

プログラム合成はソフトウェア工学の主目標のひとつである．とりわけ，仕様として入出力例題を与える方法は人間の直観に合った有望な方法であるが，これだけでは情報量が少なく，高度な合成が難しいという問題点があった．そこで，本研究では，論理型言語を対象にして，既存のライブラリ中の類似プログラムから類推を行うことで情報量を補い，入出力例題から比較的高度なプログラムを合成する方法を提案する．ここで提案する方法では，まず，与えられた入出力例題と既存の類似プログラムの入出力間に共通の制約を抽出する．このようなことを行うため，制約を表現する方法を工夫し，共通制約の自動抽出アルゴリズムを考案した．次に，プログラムの構造を抽象化した型に注目し，これを用いることにより，構造の中で共通制約と関連づけられた部分を写像する手法を考えた．それ以外の部分は，入出力例題に合わせて構造を修正するようにした．また，類似したプログラム間に共通の構造を一般化することにより新たにスキーマを生成し，作業を進めるに従って合成の効率を向上させてゆくことを可能にしている．論理型言語 Prolog によるリスト処理を行う再帰プログラムを対象として試作したプログラム合成システムを使った実験を行った結果，本提案によって，類似プログラムが存在すれば，これまでの方法よりも高度なプログラムの合成が可能であることが確かめられた．

1. はじめに

近年，必要とされるソフトウェアの量が増大し，それに伴ってプログラミングの作業量も増えている．プログラミングの作業は人間にとって負担の大きな仕事であり，またプログラミングを行うひとによって出来不出来があり，プログラムの品質も一定しない．これらの問題点を解決するための方法の1つとして，プログラムが本来目的とすることを簡単に記述した仕様を与えることによって要求をみたすプログラムを自動的に合成する「プログラム合成」という考え方がある．

プログラム合成手法を考える際に，最も重要なのは仕様をどのような形で与えるかという問題である．従来，仕様の与え方として，我々の用いている自然言語や，プログラムの意味をフォーマルに記述した論理式等が考えられてきた．なかでも，仕様として入出力例題を与える方法は最も人間の直観に合った方法だと考えられている．これは，入出力例題だけを与えれば仕様を記述することに注意を払わなくてもよいので，人間がプログラムを明確に表現する方法としては自然で容易であり，有望な方法である．入出力例題とは，例えば Prolog の表記で書けば，

$p([\,],[\,],[\,])$.
$p([\,],[a],[a])$.
$p([a],[b],[a,b])$.
$p([a,b],[c],[a,b,c])$.

のような形をしたものである．これは2つのリストを結合する例であるが，「2つのリストを結合するプログラム」を作成するよりは，このような入出力例題を与える方が人間にとってはるかに記述しやすい．

この点に注目して，入出力例題からプログラム合成を行う方法論が数多く提案されている[6),13),15)]．これらの方法論の多くは帰納推論をもとにしており，プログラム合成は決められた探索空間の探索という形をとる．したがって，探索を行えば，理論的にはどんなプログラムでも合成できるはずである．しかし，入出力例題に含まれる情報が少ないので，複雑なプログラムを合成しようとすると探索空間が爆発してしまう．したがって，入出力例題によるプログラム合成手法が対象とするのは初歩的なプログラムに限られており，これを実用的にするには何らかの付加的情報が必要になる．

我々は，人間が新しいプログラムを作成する時，過去の類似したプログラムを作成した経験を役立てるプロセスに注目する．このような，いわゆる類推を用いたプログラミングに関する研究は，定理証明を用いた方式[3)]，認知科学的な方式[11)]，誘導類推による方式[1)]，モデル推論を応用した方式[5)]，帰納推論を組み合せた方式[7)]，目的指向の類推を用いた方式[10)] 等がある．し

† An Approach to Synthesizing Logic Programs from Examples Using Analogical Reasoning by MASAKI KOIZUMI (Department of Basic Technologies Research, C&C System Research Laboratories, NEC Corporation) and MORIO NAGATA (Department of Administration Engineering, Faculty of Science and Technology, Keio University).

†† 日本電気(株) C&C システム研究所システム基礎研究部
††† 慶應義塾大学理工学部管理工学科

かし，いずれの方式も人間が関与する部分が多く，プログラムを自動的に合成するシステムに取り入れるのは難しい．

そこで，我々は自動的な合成を目指した仕様・プログラムの表現形式についての研究を行ってきた[8),9)]．本研究では，与えられた入出力例題と類似した入出力例題を持つプログラムから，仕様の特徴を表す制約式やプログラムの持つ構造などを規定する方法によって，合成に役立つ情報を取り出し，仕様としての情報を増やし，目的のプログラムを合成する手法を提案する．本論文では，この類推を用いたプログラム合成を行う手法について述べ，その実験による評価・検討を行う．

以上のことを可能とするためには，まず，仕様の特徴を表現する手段が必要である．また，具体的に類推を行うプログラムの枠組も限定しなければならない．さらに，これら仕様の特徴と枠組との関係を整理しておかなければならない．そこで，本研究では類似したプログラムから情報を抽出する3つの表現手段

1. 入出力例題仕様の特徴を表現する4種類の制約
2. Prolog プログラムの特徴を表現するプログラム構造としての分割統治法表現
3. 制約と分割統治法表現を関連づける規則

を導入し，一定の範囲内で具体的なこれらの記述を与えた．これらの表現手段を使って，類推を用いたプログラム合成の具体的なアルゴリズムを提案する．このアルゴリズムは大きくわけて5つのステップからなっている．

1. 共通制約の抽出
2. 内部構造への変換
3. プログラム構造の写像
4. プログラム構造の修正
5. スキーマの生成

このような手法を用いることにより，従来の研究のように入出力例題のみを使った手法，および純枠に類推の手法を用いるよりも広い範囲のプログラムを合成することが可能となった．本研究では，以上の考え方に基づいて，プログラム合成システムを試作し，その評価を行った．

なお，本研究で，合成の対象とするプログラムは，我々のプログラミングの経験を生かすために，

- 言語：論理型言語 Prolog
- データ構造：リスト
- 制御構造：再帰

とした．ただし，本手法ではパターンマッチングやバックトラックを陽に含むプログラムを対象にしないので，Prolog 以外の論理型言語はもちろん Lisp のような関数型言語にも直接応用できる．手続き型の言語に，本手法を応用するためには3つの表現手段の具体的な内容を入れ替える必要があるが，本研究の全体的な枠組は広く応用できるものと確信している．

本論文の構成は以下のとおりである．

- 2章ではまず，我々自身のプログラミングを分析し，その結果に基づいて，本論文で提案する類推によるプログラム合成の枠組を述べる．
- 3章では，類推を用いたプログラム合成のアルゴリズムを提案する．ここでは，アルゴリズムの説明のために union（和集合を求めるプログラム）から，類推を行うことによって intersect（積集合を求めるプログラム）を合成する例題を使う．
- 4章では，本アルゴリズムに基づく実験システムによって，多様な観点から我々のアイデアを評価し，本研究と関連する研究との比較を行い，本研究の有効性と限界について検討する．
- 5章では，本研究のまとめについて述べる．

2. 類推を用いたプログラム合成の枠組

人間は新しいプログラムを作成する時，過去の類似したプログラムを作成した経験を有効に利用している．本研究で行うべきことは，この類推を何らかの形で模倣することである．ここでは，このような類推によるプログラミングのモデルを提案し，このモデルを計算機上で実現するに当たって解決すべき点を検討する．

2.1 類推を用いたプログラミング・モデル

人間は新しいプログラムを作成する時，過去の類似したプログラムを作成した経験を有効に利用している．プログラミングの初心者の場合は，全く新しいプログラム構造を作り出すことは少なく，例題からの類推により，プログラミングを行っている[11)]．また，熟練者は，多くのプログラミングの経験や定石とから，やはり類推をプログラミングで活用している[1)]．本研究では，このような人間のプログラミングの作業を，機械学習における類推推論の技法を用いて実現する．文献4)によれば，機械学習における類推推論の基本要素は次の4段階から成り立つ．

1. 認識（Recognition）
   ターゲットの記述を与えて，候補となる類比またはソースを認識すること

2. 写像 (Elaboration)
   ソースとターゲットの間に類推写像を行うこと (類推推論を含む)
3. 評価 (Evaluation)
   使用している文脈で写像と推論を評価し，写像の正当化，修正，拡張を行うこと
4. 強化 (Consolidation)
   類推の結果が他の文脈で有効に利用できるように，類推の出力を強化すること

この4つのプロセスを，人間が類推によりプログラミングを行う場合を対象にすると，以下のように言い替えられる．

1. 新たなプログラム作成時に，過去に作成した類似プログラムを検索する
2. 過去に作成した類似プログラムの中で，新たなプログラム作成に利用可能なプログラム構造を写像する
3. 写像したプログラム構造以外の部分を修正する
4. 新しいプログラム作成に成功すると，それをプログラミング知識として覚える

ここで，最も重要なのはステップ2で，既存プログラムのどの部分を写像するかというところである．人間が入出力例題からプログラムを作成する場合，ある種の視点から入出力例題を観察し，その視点に従ってプログラムを作成している．類推によるプログラミングで普通に行っているのは，「仕様が似ているならプログラムも似ている」という考え方である．これを実際に実現するために，入出力例題とプログラムに関する視点を用意し，類似性を表現することが必要である．

2.2 入出力例題に対する視点

入出力例題に対する類似性の観点は多様であり，類似したプログラムの検索とその後の合成のために，仕様を見る視点を表現する手段が必要である．例えば，union, intersect, append の入出力例題

S1: union ($[a,b,c],[b,c,d],[a,b,c,d]$).

S2: intersect ($[a,b,c],[b,c,d],[b,c]$).

S3: append ($[a,b,c],[d,e,f],[a,b,c,d,e,f]$).

を考える．S1 と S2 は入力中の要素が共通であるという所が類似している．一方，S1 と S3 は入力の2引数目が出力の後部になっているところが共通している．このような類似性に対する視点は多様であり，自動的に類似部分を抽出するには，入出力例題の典型的な特徴を表す語彙を洗い出し，この語彙を使って類似性の視点を表現したものをシステムとして用意する必要がある．

本論文では，具体的なプログラムとして，Prolog プログラムでリスト処理を行うものを対象として考えている．そこで，多くの Prolog プログラムとその入出力となりうるものを分析して，次に示すような4つの観点から制約を整理した．このように整理したのは，我々は入出力例題を分析するのにまず入出力モードやデータ型などの表面的な情報に注目し，次に意味的な情報に注目していると考えたからである．意味的な情報としては，いろいろなレベルの制約が考えられるが，ここではなるべく多くのプログラムに利用可能な基本的で小さな単位のものを意味制約として用意した．本研究では，これらの制約を Prolog で記述してあるので拡張することは可能である．

1. 入出力モード
   引数の入出力．入力を＋，出力を－で表現する．
2. データ型
   リストの型を分類したもの [表1].
3. 意味制約
   データ型よりも，プログラムの意味を表現する制約 [表2].
   ここで eqlen (X, Y) は X と Y の長さが等しいこと，perm (X, Y) は X は Y の permutation であること，eqocc (X, Y) は X と Y には同じアトムが出現すること，joint (X, Y) は X と Y は共通要素が存在することを表している．本システ

表1 データ型
Table 1 Data types.

データ型	意味
atom	アトム
list	線形リスト
set	出現が重複しない線形リスト
ordset	順序関係がある線形リスト
struct	リスト
alist	連想リスト

表2 意味制約
Table 2 Semantic constraints.

演算型	内容	対象となるデータ型	例
len	長さ	list	eqlen (X, Y), …
order	順序	ordset	perm (X, Y), …
occ	出現	struct, alist	eqocc (X, Y), …
rel	関係	set	joint (X, Y), …

ムでは，このような制約を 20 個用意している．

4. 統合方法
複数入出力例題の制約の統合方法．すべての入出力例題についてその制約が成り立つ and とある入出力例題についてその制約が成り立つ or がある．

2.3 プログラムに対する視点

プログラマはプログラミング戦略に基づいてプログラミングを行っているので，プログラム構造に対する視点はある程度限られている．例えば，和集合をもとめるプログラム

```
union ([ ], Ys, Ys).
union ([X|Xs], Ys, Zs) :-
    member (X, Ys), union (Xs, Ys, Zs).
union ([X|Xs], Ys, [X|Zs]) :-
    not member (X, Ys), union (Xs, Ys, Zs).
```

を過去に作成した経験があれば，積集合をもとめるプログラム

```
intersect ([ ], Ys, [ ]).
intersect ([X|Xs], Ys, [X|Zs]) :-
    member (X, Ys), intersect (Xs, Ys, Zs).
intersect ([X|Xs], Ys, Zs) :-
    not member (X, Ys), intersect (Xs, Ys, Zs).
```

を作成するのは容易である．これらはリストの分解の仕方，終了条件，分岐条件（member）等の点で類似しているからである．とりわけ，リスト処理を行う再帰プログラムに限定すれば，プログラム構造はある枠の中に納まる．つまり，リストの分解の仕方，終了条件，分岐条件等の特性を表す手段があればよい．

本研究では，再帰的プログラムを対象としているので，考えられるプログラム構造としては分割統治法・生成検査法・二分探索法などがある．この中で，リスト処理を行うプログラムには分割統治法が用いられることが多いので，ここではプログラム構造として分割統治法を用いる．分割統治法とは，クイックソート，マージソート等のアルゴリズムを実現するのに使われている方法で，その原理は「もし問題への入力が充分単純なものであれば直接それを解き，そうでなければ，入力を分解し副問題を解き，解を合成する．」というものである[14]．

本稿では分割統治法を広義にとらえ，Prolog で，

P←Decomp, Guard, Q, P, Comp.

のように表現することにする．これは，入力を2つに分解（Decomp）し，分岐条件（Guard）に当てはまれば，補助述語（Q）と再帰（P）にかけ，出力を合成（Comp）するということである．分割統治法を用いれば，プログラムは4つ組〈Decomp, Guard, Q, Comp〉で表現できる．4つのパラメータにはそれぞれ述語を用いる．Prolog の節を分割統治法の形に変換したものを DC 表現*と呼ぶ．DC 表現は，Prolog プログラム中を流れる入出力例題のデータフローを明示的に表現したものである．このデータフローを視覚的に表したものが**図1**であり，各ノードには具体的な述語が入る．

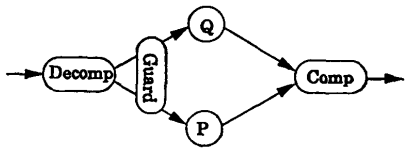

図 1 DC 表現のデータフロー
Fig. 1 Dataflow of DC-representation.

ただし，本研究では入出力仕様を扱い，対象はリスト処理を行う再帰プログラムなので，分割統治法に次のような制限を加えることにする．

- プログラム中に特別なアトムの記述を許すと，そのようなアトムすべてをあらかじめ語彙として用意しておかなければならないが，そうすることは事実上不可能である．よって，入出力例題の中で出力中に現れるアトムはすべて入力中に含まれることとし，パラメータに新しいアトムを出力するような述語を用いないという制限をつけてあらかじめ語彙を準備しておかなくてもよいようにした．
- 分割統治法は入出力例題のデータフローを表しており，データに対する演算は述語で行う．このため，述語内の引数は変数だけであり関数記号は含まないこととする．Prolog で car と cdr を表す関数記号“|”を表現するためには次の述語を導入する．

```
cons (X, Xs, Xs1) :- Xs1=[X|Xs].
firstrest (Xe1, X, Xs) :- Xs1=[X|Xs].
```

なお，これらの制約を付けてもリスト処理を対象とした再帰プログラムで十分に汎用性を失わないことは検討のところで示す数字によって確認できた．

プログラムを DC 表現で記述した例を以下に示す．

（例1） union プログラム

union プログラムの DC 表現は**図2**に示すとおり

* Divide & Conquer の略である．

```
P(Xs1,Ys1,Zs1) :-
        nil(Xs1),                          ...Decomp0
        id(Zs1,Ys1).                       ...Comp0
P(Xs1,Ys1,Zs1) :-
        firstrest(Xs1,X,Xs),id(Ys1,Ys),    ...Decomp1
        member(X,Ys),                      ...Guard1
        P(Xs,Ys,Zs),                       ...P1
        id(Ys,Ys1).                        ...Comp1
P(Xs1,Ys1,Zs1) :-
        firstrest(Xs1,X,Xs),id(Ys1,Ys),    ...Decomp2
        not member(X,Ys),                  ...Guard2
        id(X,Z),                           ...Q2
        P(Xs,Ys,Zs),                       ...P2
        cons(Z,Zs,Zs1).                    ...Comp2
```

図 2 union の DC 表現
Fig. 2 DC-representation of union.

である．図中，「...」の右側がパラメータ名であり，パラメータについている添字は節の番号である*．左辺はパラメータを満足する具体的な述語である．例えば，第1節のパラメータ Guard を満たす述語は member である．

DC 表現では，プログラムの述語名が変数になっている．この場合，述語名 union を変数 P に置き換えている．

次に，*Decomp* と *Comp* を明確に分離するために，入力引数を出力引数中で使う場合には新しい変数を導入している．例えば，第2節は X という変数が入力引数と出力引数で共有されているので，出力変数 Z を導入することにより，入力変数は X，出力変数は Z と分離できる．Q に id を用いて変数の同一化を行う**．

また，節の頭部中で関数記号を含む項は，述語を導入して本体に付加している．入力引数の場合は，本体に *Decomp* を付加する．例えば，頭部の $[X|Xs]$ は新しく導入した変数 $Xs1$ に置き換え，*Decomp* の述語 firstrest を本体に付加する．均一な表現にするため，頭部の Ys も変数 $Ys1$ に置き換え，id を本体に付加する．出力引数の場合は，本体に *Comp* を付加する．頭部の $[Z|Zs]$ は変数 $Zs1$ に置き換え cons を本体に付加する．

（例2） マージソート

msort ([ ], [ ]).

msort (Xs, Ys) :-
split (Xs, $Xs1$, $Xs2$),
msort ($Xs1$, $Ys1$), msort ($Xs2$, $Ys2$),
merge ($Ys1$, $Ys2$, Ys).

マージソートでは，split で入力リストを2つに分割し，それぞれのリストに対して再帰的にマージソートを行い，その結果をマージする．ここでは1つの節の中に再帰呼び出しが2回現れるが，1つ目の再帰呼び出しを Q，2つ目の再帰呼び出しを P，$Q=P$ であると考える．これを DC 表現を簡略化して記述すると以下のようになる．

{<Decomp, Guard, Q, Comp>} =
{<[ ], [ ]>, <split, _, p, merge>}

3. アルゴリズム

本章では，2章で述べたような入出力例題とプログラムに対する視点を定式化し，アルゴリズムとして記述する．

入出力例題によるプログラム合成とは，入出力例題の列 $S_{Target}=\{S_1, \cdots, S_n\}$ を与えると，プログラム P_{Target} を合成するが，S_{Target} の情報だけでは，単純な P_{Target} しか合成できない．そこで，本研究では類似したプログラムからの類推により，情報を増やすアプローチをとる．

本研究のプログラム合成の流れを**図3**に示す．次のような5ステップを経て，目的プログラムの入出力例題 S_{Target} から目的プログラム P_{Target} を合成する．

1. 共通制約の抽出
   目的プログラムの入出力例題 S_{Target} が与えられると，これらの例題で成立している制約と同様の制約が成立している類似プログラム S_{Target} を検索した後，S_{Target} と類似プログラムの入出力

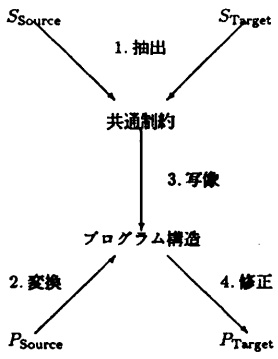

図 3 本アプローチのアルゴリズムの流れ
Fig. 3 An algorithm of our approach.

* 節の番号は0から数え始め，第0節，第1節…と呼ぶことにする．
** id(X, Z) :- $X=Z$.

例題 S_{Target} の間に成り立つ制約を抽出する．

2. 内部構造への変換
類似したプログラム P_{Source} を内部構造へ変換する．
3. プログラム構造の写像
内部構造の中で共通な制約と関連づけられた部分を写像する．
4. プログラム構造の修正
その構造を修正することにより目的プログラム P_{Target} を合成する．
5. スキーマの生成
類似したプログラム間に共通の構造を一般化することにより新たなスキーマを生成する．

以下，この5ステップにおけるアルゴリズムを詳しく示す．ただし，本論文を通じて，intersect（積集合を求めるプログラム）を合成する例を用いる．あとで実験結果で示すように，より複雑なプログラムを合成することが可能であるが，説明のために単純な例を用いる．さて，入出力例題は次のような形で与える．

mode intersect (+, +, −).
intersect ([ ], [ ], [ ]).
intersect ([ ], [*b*, *c*], [ ]).
intersect ([*b*], [*b*, *c*], [*b*]).
intersect ([*a*, *b*], [*b*, *c*], [*b*]).

新しく入出力例題が与えられると，制約の中で最も基本的な入出力モードをキーにして類似しているプログラムをライブラリから検索する．これは検索の第一ステップなので，検索のための計算量が少なく必要な類似プログラムを検索しないことを防ぐことだけを考えている．ここでは，intersect の入出力モード情報（第1引数，第2引数が入力（+），第3引数が出力（−））をキーにして，S_{Source} として union（和集合を求めるプログラム）を検索するとして話を進める．

3.1 共通制約の抽出

入出力例題の間で，意味制約によって表現された部分が多いほど類似したプログラムであると考えられ，その共通点が写像のステップにおいて利用できる．そこで，求めるプログラムの入出力例題と類似のプログラムの入出力例題に成り立つ共通の制約をまず抽出する．この例では，類似プログラムの入出力例題

mode union (+, +, −).
union ([ ], [ ], [ ]).
union ([ ], [*b*, *c*], [*b*, *c*]).
union ([*b*], [*b*, *c*], [*b*, *c*]).
union ([*a*, *b*], [*b*, *c*], [*a*, *b*, *c*]).

と目的プログラムの入出力例題 intersect の入出力例題からそれぞれ

{set (+1), set (+2), set (−1),
　joint (+1, +2), suffix (+2, −1)}
{set (+1), set (+2), set (−1),
　joint (+1, +2), subset (−1, +2)}

なる制約が抽出される．そして，この2つの制約を制約間の階層関係を用いて一般化することにより，共通の制約

{set (+1), set (+2), set (−1), joint (+1, +2)}

を求める．この共通制約は，第1入力引数，第2入力引数のデータ型が set 型，第1出力引数のデータ型が set 型，第1引数と第2引数の間には共通要素があるということを表している．

3.2 内部構造への変換

この例では，P_{Source} として選ばれた union のプログラムは，ヘッド側の単一化を陽にボディ側に出すことにより図2のような DC 表現に変形される．

3.3 プログラム構造の写像

本研究ではプログラムの類似性を反映し，かつ入出力例題から自動抽出が可能で汎用的な仕様の表現方法を提案した．この仕様に含まれる制約の中には，プログラム構造としての DC 表現と関連付けられるものがある．ただし，この関連付けにはインプリシットな知識が必要である．例えば，手続き型言語において，2次元配列という仕様中の制約に対しプログラム構造として2重ループを使う．これは，N次元配列という制約とN重ループという構造は関係しているという知識を用いていると考えることができる．

このような制約と構造の関係を表現した知識を**図4**に示す．構文1はデータ型と述語の関係を表したものである．ある引数のデータ型とその引数を用いている述語は関係していることをしている．例えば，Rule 1 と Rule 2 は構文1を用いたルールの例である．Rule 1は入力引数の型はリストの分解方法 Decomp に関係すること，Rule 2 は出力引数の型はリストの併合方法 Comp に影響することを記述している．

構文2はデータ型よりも詳細な関係を表している．あるデータ型・演算型・統合型と述語の関係を表したものである．例えば，Rule 3 と Rule 4 は構文1を用いたルールの例である．Rule 3 は，入力の2引数の間に，あるデータ型で，ある意味制約があり，or という統合方法を用いているならば，条件分岐 Guard

```
構文1: str(モード,述語の組) :- con(モード,データ型).
構文2: str(モード,述語の組) :- con(モード,データ型,意味制約,統合方法).

Rule1: str([+X],[decomp(P)]) :- con([+X],Type).
Rule2: str([-X],[comp(P)]) :- con([-X],Type).
Rule3: str([+X,+Y],[guard(P)]) :- con([+X,+Y],Type,Sem,or).
Rule4: str([+X,-Y],[decomp(P),comp(P)]) :- con([+X,-Y],Type,Sem,and).
```

図 4 制約と構造を関連付けるルール
Fig. 4 Rules between constraints and structures.

```
Rule1: str([+1],[decomp(firstrest)]) :- con([+1],set).
Rule1: str([+2],[decomp(firstrest)]) :- con([+2],set).
Rule2: str([-1],[comp(cons)]) :- con([-1],set).
Rule3: str([+1,+2],[guard(member)]) :- con([+1,+2],set,joint,or).
```

図 5 union の関係付けに用いられたルール
Fig. 5 Rules used for the explanation of union.

が起こることを表している.

ただし，以上のルールは制約から構造への1対多関係であり，ひとつの制約に対してそれを実現した構造は複数ある．すなわち，ルールは常に成立するわけではない．ソース領域において成立するルールだけが意味を持ち，それをターゲット領域に写像する．

例えば，類似プログラム (union) において，制約と構造の関係付けの際に利用されたルールは**図5**に示したとおりである．Rule 1 は，第1入力引数，第2入力引数は set 型なので，対応する構造として Decomp の firstrest を用いることを示している．同様に Rule 2 は，第1出力引数は set 型なので，対応する構造として Comp の cons を用いることを示している．Rule 3 は，第1入力引数と第2入力引数の間に共通要素がある joint (+1, +2) という関係が成立する場合が存在する (or) ので，対応する構造として Guard の member を用いることを示している．この中で，新しい入出力例題 (intersect) の制約が成立するルールは Rule 1 と Rule 3 であり，これらを写像する．

3.4 プログラム構造の修正

写像されずに残った部分については，順にパラメータのインスタンスを生成することにより，修正を行う．本手法では**表3**のようなインスタンスを用意している．

ここで，Level 0 は cons だけで表現できるもの，Level 1 は代表的な述語，Level 2 はユーザが定義する述語，であり，それぞれ代表的なものを挙げている．Level 0 から順にインスタンスを生成してゆき，メタインタプリタ上で，入出力例題をみたすかどうかをテストする．Level 1 までで目的のプログラムが求まらない場合は，Level 2 の述語をユーザが与える．そして，検査の結果うまくいったパラメータを写像されたプログラム構造に埋め込むことにより，目的のプログラムを求める．

この例では，写像されずに残った3つのパラメータ $Comp_0$, $Comp_1$, $Comp_2$ は，

$Comp_0$: nil ($Zs1$)
$Comp_1$: cons (Z, Zs, $Zs1$)
$Comp_2$: id (Zs, $Zs1$)

のように修正される．このパラメータを写像されたプログラム構造に埋め込むことにより，目的のプログラム

```
intersect([ ], A, [ ]).
intersect([A|B], C, [A|D]) :-
    member(A, C), intersect(B, C, D).
intersect([A|B], C, D) :-
    not member(A, C), intersect(B, C, D).
```

を得る．

3.5 スキーマの生成

目的プログラムの合成に成功すると，2つのプログラムに共通の制約と共通の構造をペアにした新たなスキーマを生成する．この例では，intersect と union からなる集合演算に関するスキーマを自動的に生成する（**図6**）．このようなスキーマをライブラリに蓄えておけば，今後スキーマの前提部にマッチする入出力例題についてはスキーマを利用すればよいのでプログラム合成の効率が上がる．

この集合演算に関するスキーマを学習した後に，例えば

表 3 修正用の述語
Table 3 Predicates used in the modification step.

	Level 0	Level 1	Level 2
Decomp	=, firstrest, sub1	listsplit	partition...
Guard	=	〈, =〈, 〉=, 〉	member...
Q	=	*P*	rev...
Comp	=, cons	append	merge...

```
scheme(
   set_operation,
   [set(+1),set(+2),set(-),joint(+1,+2)]
   [cl([P,Xs1,Ys1,Zs1],
      [Xs1=[],Comp0]),
    cl([P,Xs1,Ys1,Zs1],
      [firstrest(Xs1,X,Xs),id(Ys1,Ys),member(X,Ys),P(Xs,Ys,Zs),Comp1]),
    cl(P(Xs1,Ys1,Zs1),
      [firstrest(Xs1,X,Xs),id(Ys1,Ys),not member(X,Ys),P(Xs,Ys,Zs),Comp2])]
).
```

図 6 union と intersect から一般化されたスキーマ
Fig. 6 The schema of set operation which is generalized from union and intersect.

setdiff ([ ], [ ], [ ]).
setdiff ([ ], [*b*, *c*], [ ]).
setdiff ([*b*], [*b*, *c*], [ ]).
setdiff ([*a*, *b*], [*b*, *c*], [*a*]).

なる仕様を与えると，共通制約とマッチングすることにより，Guard として述語 member を利用できることなどがわかる．

ここで，システムが獲得したスキーマのうち代表的なものを示す．(簡単のため *Decomp*, *Comp* のうち head 側に埋め込めるものは埋め込んだ)

- FULL（2分木を扱う）
  P([ ], [ ]).
  P(*X*, *Y*) :- atomic (*X*), Comp 1.
  P([*X*|*Xs*], *Ys*1) :- *P*(*X*, *Y*),
  P(*Xs*, *Ys*), Comp 2.
- MAP（構造を写像する）
  P(*A*, [ ], [ ]).
  P(*A*, [*X*|*Xs*], [*Y*|*Ys*]) :- *Q*, *P*(*A*, *Xs*, *Ys*).
- PAIR（2つの入力リストを組み合わせる）
  P([ ], [ ], [ ]).
  P([*X*|*Xs*], [*Y*|*Ys*], *Zs*1) :- *Q*, *P*(*Xs*, *Ys*, *Zs*),
  Comp 1.
- FILTER（条件 Guard が成立しない要素に対して Comp を行う）
  P(*X*, [ ], [ ]).
  P(*X*, [*Y*|*Ys*], *Zs*) :- Guard, *P*(*X*, *Ys*, *Zs*).
  P(*X*, [*Y*|*Ys*], *Zs*1) :- *P*(*X*, *Ys*, *Zs*), Comp 2.

4. 検　討

本システムが扱うことができるプログラムに対して，対象となるプログラムの範囲，他のプログラム合成システムとの比較の2つの面から評価を行う．

4.1 対象となるプログラムの範囲

本システムが対象とするプログラムの範囲に対する定量的な評価を行う．客観的な基準として，Prolog プログラミングの教科書[2]，Prolog 処理系のライブラリプログラム[12]の2つを対象に評価する（表4）*．まず，本研究では入出力例題を表現する手段としてリストを選んだが，ほとんどのプログラムがリスト処理を行うプログラムであった．ここで2，3割というのは低い数字に見えるが，リスト処理以外のプログラムはそのままダイレクトに使うものがほとんどで，人間が変更を加えて使うプログラムのかなりの部分はリスト処理と考えられる．

次に，本研究ではプログラムの抽象的な構造として DC 表現を考案し，これを用いて類推を行ったが，リスト処理を行うプログラムのうち，約8割のプログラムが DC 表現に変換できる．

このようにリスト処理という範囲に限れば，本研究ではかなりのプログラムを対象にできることがわかる．

4.2 他のプログラム合成システムとの比較

本システムを入出力例題からのプログラム合成の代表的な3つの手法と比較する（表5）．ここで，条件

表 4 対象となるプログラムの範囲
Table 4 The range of programs that our system can operate.

対　象	総　数	リスト処理	分割統治法
教科書[2]	36	63.9%	82.6%
ライブラリ[12]	252	28.6%	48.6%

表 5 他のプログラム合成システムとの比較
Table 5 Comparison with related systems.

対　象	条件付再帰	二重再帰	補助述語
GAP[6]	×	○	扱えない
THESYS[15]	×	×	扱える場合あり
MIS[13]	○	○	与える
本システム	○	○	与えなくて良い

* 表中の総数は述語名の総数である．同じ意味の述語に対して複数の名称が存在する場合があるので重複もある．

付再帰とは *Guard* が存在する再帰である．例えば，union の第1節は *Guard* として述語 member を用いた条件付き再帰である．二重再帰とは $Q=P$ となる再帰である．例えば，msort の第1節は二重再帰である．

まず GAP はあらかじめ用意されたスキーマとのパターンマッチングによりプログラム合成を行う．しかしながら，GAP で扱うスキーマは数種類であり成長性もない．例えば，二重再帰のスキーマは存在するので二重再帰のプログラムは合成できるが，条件付き再帰のスキーマはないので条件付き再帰のプログラムは合成できない．本システムでは，必要に応じてスキーマを生成，精密化できるという点で GAP より広い枠組を提案したと言える．GAP が合成したプログラムについては，適切な類似プログラムを与えてやれば本システムで合成が可能であった．

次に THESYS で扱うプログラム構造はリストの要素をマッピングするような関数だけであり，条件分岐を含むようなものはお手上げである．なぜならば，プログラム構造として

$$f[x]=a[f[b[x]]\,;\,x]$$

という種があるからである．これは分割統治法で表せば，

$$\{<\text{Decomp, Guard, } Q\text{, Comp}>\}$$
$$=\{<[\,],[\,]>, <b, \_, \_, a>\}$$

であり，条件付き再帰や二重再帰は扱えないので，分割統治法の方が広い枠組であるといえる．THESYS の特長としては，帰納推論がうまくいかない場合，変数付加 (Variable Addition) というテクニックを用いて補助プログラムを導入している点が挙げられる．

MIS は帰納推論による方法のため，補助述語を理論名辞としてユーザが明確に与えなければならない．本研究では，補助述語は類推により求めるので MIS よりは人間に近い枠組を提供していると言える．また，MIS ではプログラム合成を常に零から行っており，学習効果がない．本システムではプログラム合成に成功するとスキーマを作成するので，この点で有効であると言える．ただし，本手法を用いても，MIS の合成例をすべて合成できなかった．これは，本手法はリスト処理を対象にしているため，木構造などのデータ構造に対する述語をサポートしていないからである．また，MIS では入出力例として正例，負例の両方を用いて推論を行っているが，本手法では正例しか扱えない．これらについては今後の課題である．

5. ま と め

論理型言語 Prolog を対象に，以下の知識

- 入出力例題の性質を表現した制約
- プログラムを抽象化した構造としての DC 表現
- 制約と構造を関係付けるルール

を整理しておき，既存のライブラリ中の類似プログラムから類推を行うことにより，プログラムを合成する手法を提案した．この手法を用いれば，入出力例題から自動的に抽出でき，かつ汎用性が高く，その一部がプログラム構造の写像に利用可能な制約を整理しておくことができたので，類似プログラム中の補助述語を自動的に同定し写像することができる．その結果，リスト処理を行う再帰プログラムという限定された領域ながらも，人間が補助述語・制御情報など付加的な情報を与えることなしに，入出力例題のみからプログラムを自動的に合成する枠組を実現した．

ただ，推論形態として類推を用いているため，目的プログラムと類似プログラムの差異が大きい場合は適切な合成ができない場合がある．また，Prolog 特有の非決定的プログラム，あるいは数値処理を含むプログラムなどは，分割統合法だけで表現できない．これらの問題点を解決するために，以下の事項が今後の課題である．

- 複数ソースからの類推
  目的プログラムと類似プログラムの差異が極端に大きい場合は，適切な合成ができないので，複数の類似プログラムからの類推を行うことにより差異を縮める．
- プログラム構造の拡張
  分割統治法以外のプログラム構造として生成検査法・二分探索法等を定式化し，また，複数のプログラム構造の組み合わせも扱うこと．

謝辞 本研究をまとめる機会を与えてくださった，日本電気(株)後藤敏 所長代理，吉村猛 部長，宮下洋一 課長，および中島震 主任に感謝いたします．

参 考 文 献

1) Carbonell, J. G.: Derivational Analogy, *Machine Learning*, Michalski, R. S. et al. (eds.), Vol. 2, pp. 371-392, Morgan Kaufmann (1986).
2) Clocksin, W. F. and Mellish, C. S.: *Programming in Prolog*, Springer-Verlag (1981).
3) Dershowitz, N.: Programming by Analogy, *Machine Learning*, Michalski, R. S. et al. (eds.), Vol. 2, pp. 393-421, Morgan Kaufmann (1986).

4) Hall, R. P.: Computaional Approaches to Analogical Reasoning: A Comparative Analysis, *Artif. Intell.*, Vol. 39, pp. 29-120 (1989).
5) 原口：類推の機械化について，古川(編)，知識の学習メカニズム，pp. 125-154，共立出版 (1986).
6) Hardy, S.: Synthesis of LISP Function from Examples, *Proc. of 4th IJCAI*, pp. 240-245 (1975).
7) 今中，上原，豊田：類推，帰納の概念を導入したプログラミング知識の学習メカニズム，情報処理学会研究報告，88-AI-13 (1988).
8) 小泉，永田：トレースと設計戦略を用いた Prolog プログラムのトップダウンな合成方法，第39回情報処理学会全国大会論文集，pp. 1056-1057 (1989).
9) 小泉，永田：類推を用いた Prolog プログラムの合成手法の提案，ソフトウェア科学会第7回大会，pp. 297-300 (1990).
10) 沼尾，志村：説明に基づく Prolog プログラムの合成，人工知能学会研究会，SIG-FAI, pp. 11-19 (1988).
11) Pirolli, P. L., Anderson, J. R. and Farrell, R.: Learning to Program Recursion, *Proc. of the 6th Annual Conference of the Cognitive Science Society*, pp. 277-280 (1984).
12) Quintus Prolog User's Guide, version 13, Quintus Computer System, Inc. (1988).
13) Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press (1983).
14) Smith, D. R.: Top-Down Synthesis of Divide and Conquer Algorithms, *Artif. Intell.*, Vol. 27, pp. 43-96 (1985).
15) Summers, P. D.: A Methodology for LISP Program Construction from Examples, *JACM*, Vol. 24, pp. 161-175 (1977).

(平成2年12月19日受付)
(平成3年 6 月13日採録)

小泉 昌紀（正会員）

1964年生．1988年慶應義塾大学理工学部管理工学科卒業．1990年同大学院理工学研究科（管理工学専攻）修士課程修了．同年日本電気(株)入社．C&C システム研究所勤務．現在，知的ソフトウェア設計支援に関する研究開発に従事している．プログラム合成，プログラム認識，機械学習，知識獲得などに興味を持つ．人工知能学会会員．

永田 守男（正会員）

1948年生．1971年慶應義塾大学工学部管理工学科卒業．1973年同修士課程修了．1974年同大学工学部助手．専任講師を経て，現在，慶應義塾大学助教授（理工学部管理工学科）．工学博士（慶應義塾大学）．同大学計算センター日吉計算室長を兼務．数式処理と定理の自動証明およびそれらの各方面への応用，ヒューマンインタフェース，人間のコミュニケーションへの情報技術の活用などを研究．電子情報通信学会，日本ソフトウェア科学会，日本経営情報学会，ACM，IEEE 各会員．