1N-4

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

小泉 昌紀　永田 守男　（慶應義塾大学理工学部）

1．はじめに

プログラム合成の方法として、例題提示仕様によるものがある。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反面、現実的には小さなプログラムしか合成できないという問題点がある。これは、トップレベルの入出力仕様だけからでは、「いつ補助的述語が必要か」といった情報が抽出できないためである。本稿では、半順序の入出力仕様を、再帰スキーマを用いて記号実行することにより、全順序の入出力仕様を作り、プログラムを合成する方法を提案する。また、この方法を拡張することにより、過去の知識を積極的に利用した効率的な「プログラム合成／再利用の設計戦略」を考える。対象とするクラスは、リスト処理を行うPrologの再帰プログラムとする。

2．例題提示仕様の問題点

例による論理プログラムの自動合成において、最も本質的でかつ難しい問題は、合成の対象となる述語以外の補助的述語をいかにしてシステムが獲得するかという問題である。例えば、リストをクイックソートするプログラムqsort/2 においてpartition/4 append/3 を補助的述語として使うが、qsort/2 に関する正負の例

```
qsort([],[]), true.
qsort([1],[1]), true.
qsort([2,1],[1,2]), true.
qsort([2,1],[1,2]), false.
    ...
```

だけから付加知識なしに補助的述語を発見することは一般に不可能である。

Shapiro のＭＩＳ（モデル推論アルゴリズム）[1] では、補助的述語を理論名辞として与えることでこの問題を回避している。しかし、実際問題として合成可能なプログラムのクラスは、補助的述語が既知の自己再帰述語が限界であり、それ以上になると探索空間が爆発してしまう。ＭＩＳには、他にも、矛盾点追跡の際、負の例は積極的に利用しているが、正の例はプログラムのテストに使っているだけだという問題点がある。

3．基本的なアイデア

ＭＩＳには
1) 補助的述語を与えなければならない。
2) 正の例を積極的に使っていない。
という問題点があった。これに対して、
1) 再帰のパターンを与える。
2) 正例の構造体（トレース）を用いる。
といったことを考えることにする。

1)については、再帰の基本としてDivide & Conquer アルゴリズムを考える。これは

```
P ← Directory_Solve.
P ← Decomp, G, P, Comp.
```

と記述できる。[2] これを、次のように詳細化して利用し、再帰スキーマと呼ぶことにする。

```
Ｒ１（単純再帰）：      P←P.
Ｒ２（条件付き再帰）：  P←G, P.
Ｒ３（２重再帰）：      P←P, P.
Ｒ４（補助述語再帰）：  P←Q, P.
                        Q←Q.
Ｒ５（相互再帰）：      P←Q.
                        Q←P, Q.
```

2) については、トレース全体を与えれば良いのであるが、プログラムが複雑になってくるとそれは無理である。そこで、仕様を

```
qsort([2,1,3],[1,2,3])
      >> 1<2,qsort([1],[1])
            >> qsort([],[])
            >> qsort([],[])
      >> 3>2,qsort([3],[3]).
            >> qsort([],[])
            >> qsort([],[])
```

というように半順序関係で与え、これを全順序化することにする。

以上の考え方を実現したものを、Ｃ－アルゴリズムと呼び、４で述べる。Ｃ－アルゴリズムでプログラムを合成できるが、ライブラリ述語を用いて仕様の記述量を減らすことを考える。次の２つの場合である。
A) 仕様を満たす述語が存在..Ａ－アルゴリズム
B) 類似した述語が存在.....Ｂ－アルゴリズム
ＡＢＣのアルゴリズムを含んだ大域的な戦略を「設計戦略」と呼び５で述べる。

4．合成アルゴリズム

Ｃ－アルゴリズムは、以下のようなステップを経て、半順序の仕様から全順序の仕様（トレース）を得る。

Ｓｔｅｐ１：再帰スキーマの選択
半順序の入出力仕様から、適切な再帰スキーマを選択する。データ構造がＢＮＦで記述できる場合は、それを再帰スキーマとできる。[3]

Top-Down Synthesis of Prolog Program Using Trace and Design Strategy
Masaki KOIZUMI and Morio NAGATA
KEIO Univ. Faculty of Science and Technology

Ｓｔｅｐ２：再帰ラインの作成
入出力仕様を再帰スキーマに当てはめることにより、再帰ラインを作成する。
Ｓｔｅｐ３：再帰ラインの精密化
精密化オペレータを加えることにより再帰ラインを精密化する。
Ｓｔｅｐ４：プログラムの合成
トレースを最小一般化して、プログラムを合成する。

qsort/2 の例だと、Ｇ＝Ｐと特定化することにより補助述語の仕様
```
Decomp(2,[1,3],[1],[3]).   (1<2,3>2を含む)
Comp(2,[1],[3],[1,2,3]).
```
が得られる。これらのトレースを与えて、Ｃ－アルゴリズムを再帰的に適用してゆけばqsort/2 のプログラムが得られる。

５．設計戦略

（１）Ａ－アルゴリズム
以下のステップを経て、与えられた仕様を変換し、ライブラリから述語を検索する。Ａ－アルゴリズムは、精密化オペレータをかける対象が項なので、探索空間は少なくて済む。
Ｓｔｅｐ１：次のオペレータを用いて入出力の組を変換する。
1) Ｐｅｒｍｕｔｅ　（引数の交換）
2) Ｓｐｅｃｉｆｙ　（特殊化）
3) Ａｄｄａｒｇ　（引数の付加）
4) Ｄｉｖｉｄｅ　（分解）
Ｓｔｅｐ２：変換された入出力の組を、検索のための仮説[4] に変換する。仮説のプリミティブは次のようなものである。
1) リストの長さ　|A|
2) アトムの出現要素　A^{-}
3) 出現順序　A<<B

例えば、Ｃ－アルゴリズムで分解された仕様
```
Comp(+2,+[1],+[3],-[1,2,3])
```
はＳｐｅｃｉｆｙを用いて
```
<I1,I2,I3,O> -> <I2,[I1|I3],O>
```
と変換でき、入出力に関する仮説からappend/3 を直接用いればよいことがわかる。

（２）Ｂ－アルゴリズム
以下のステップを経て、類似した述語からプログラムを合成する。Ｂ－アルゴリズムは推論結果の途中までを利用するので、Ｃ－アルゴリズムを直接用いるより効率的である。
Ｓｔｅｐ１：再利用可能部分の抽出
再帰ラインの中で部分的に利用できる「再利用可能部分」を抽出する。
Ｓｔｅｐ２：パラメータの探索
リストに関する制約のもとで、有効なパラメータを発見する。
Ｓｔｅｐ３：パラメータの埋め込み
パラメータを再帰ラインに埋め込むことで、プログラムを合成する。

例えば、過去にintersection/3 を合成済みであるとして、union/3 を合成することを考える。再帰ラインの入力部分が再利用可能であり、
```
P([],Ys,*D*).
P([X|Xs],Ys,Zs) :- member(X,Ys),
      P(Xs,Ys,Zs), *Comp1*.
P([X|Xs],Ys,Zs) :-
      P(Xs,Ys,Zs), *Comp2*.
```
パラメータ {*D*,*Comp1*,*Comp2*} を制約
```
*D* = [] or Ys
*Comp1* ≠ *Comp2*
```
のもとで解くと {Ys,Id,cons} が得られ、これを再帰ラインに埋め込むことで union/3 が合成できる。

これまで述べたＡＢＣのアルゴリズムをまとめた再帰的なアルゴリズムを「設計戦略」と呼び、次に示す。

```
Design-Strategy:
while 解が見つからない
 do begin
   if 既存の述語を利用可能
      then Ａ－アルゴリズムを適用
      else if 類似の述語を利用可能
         then Ｂ－アルゴリズムを適用
         else
              Ｃ－アルゴリズムで仕様を分解
              分解した仕様にDesign-Strategy
              を適用
         end.
      end.
 end.
```

「設計戦略」はＣ－アルゴリズムよりも仕様の記述量は少なくて済む。また、「設計戦略」はプログラム合成と再利用を融合した枠組みと考えることができる。

６．おわりに

以上のアイデアに基づくシステムを現在試作中であるが、次のような２つの問題点がある。
1) 入出力仕様と再帰スキーマの関係が明らかでない。
2) 「冗長な仕様」は扱えるが、「不十分な仕様」や「矛盾した仕様」は扱えない。
これらの解決策として、
1) 再帰スキーマの使い方に関する学習
2) 仕様の十分性のチェック、仕様の矛盾点追跡[5]
などが挙げられる。

【参考文献】

[1] Shapiro,E.Y., Algorithmic Program Debugging, ACM Distinguished Dissertations, The MIT Press (1982).
[2] Smith,D.R., Top-Down Synthesis of Divide and Conquer Algorithms, Artificial Intelligence 27, pp.43-96 (1985).
[3] 小泉,永田, データ構造に注目したプログラム変換システム, 第３７回情報処理学会全国大会, pp.639 (1988).
[4] 深谷,永田, Prologプログラムのデバッグのための検証と説明, 第３９回情報処理学会全国大会（予定）, (1989).
[5] 御宿,永田, 動的解析と静的解析を融合したPrologプログラムのデバッグ, 第３９回情報処理学会全国大会（予定）, (1989).