数理解析研究所講究録
1125巻 2000年 115-122

数式処理系を利用したプログラム変換

The method for solving recurrence of two argument

松谷 将寛† 二村 良彦††
Masahiro Matsuya Yoshihiko Futamura

†早稲田大学大学院 理工学研究科
Graduate School of Science and Engineering, Waseda University
††早稲田大学 理工学部
School of Science and Engineering, Waseda University

概要

与えられたプログラムを自動的に改良するプログラム変換において，数式処理系が利用できる多くの局面が存在する．例えば，どんなデータを扱うプログラムであってもその反復回数は非負の整数であり，数式処理によりプログラムを実行する以前に計算できる場合である．我々は，数値計算をする1または2変数の木再帰プログラムを，閉じた式に最適化するシステムを，Common Lisp (実際には ACL) で作成したので，その方法と性能について報告する．方法の概略は次の通りである： (1) 1変数で定数係数線形の場合は特性方程式による解法を使う。(2) 前記1が適用できない場合は母関数による解法を使用する。なお、処理の過程で必要となる数学的処理には、数式処理系 REDUCE を用いた．

1 はじめに

プログラム変換 [1,4] の目的は，与えられたプログラムを出来るだけ高性能なものに自動変換することである．従って，プログラムが数値を扱う再帰プログラムで，かつそれが閉じた式を解として持つならば，その解を求めることをプログラム変換も試みるべきである．ただし，閉じた式とは非再帰的かつ繰返し回数が有界な反復式を意味する．これは，完全に数式処理 [3] である．閉じた式はその性質（有界性，単調性等）がプログラム自身よりも理解し易く，かつ簡約化や計算がし易い場合が多い．一方，再帰プログラムが閉じた解を持たない場合にはプログラム変換は，より簡単な再帰プログラムに変換することを試みる．そのために，プログラム変換は畳込み (fold) や展開 (unfold) などのプログラム変換操作を行う．さらに，その再帰プログラムが実行のために必要なスタック操作を不要にするために，反復プログラムに変換すること (再帰除去) を試みる．これは，数式処理とは異なる．従来のプログラム変換では，与えられた再帰プログラムが閉じた解を持つ場合にもそれを求めることをしなかった．例えば，n番目のフィボナッチ数を求める再帰プログラム fib(n) が与えられても，それをより実行時間の少ないプログラムに変換するのみで，その解である

$$fib(n) = \frac{\phi + \hat{\phi}}{\sqrt{5}}$$

(但し $\phi = \frac{\sqrt{5}+1}{2}$ かつ $\hat{\phi} = \frac{\sqrt{5}-1}{2}$)

を求めることはしなかった．一方，数式処理のみを用いると，数式処理システムが扱えることが前もって分かっている形の方程式のみを処理することしかできない．例えば次の様な再帰プログラム f[2] の解を直接求めることの出来る数式処理システムを我々は知らない．

(1.1) f(m,n)=0 if m<n or n≤0.
(1.2) f(m,m)=1.
(1.3) f(m,n)=(m-n)f(m-1,n)+f(m-1,n-1)

この再帰プログラムが数式処理システムで直接処理できるようにするために，展開，畳込み等のプログラム変換技術を駆使する．また数式処理の結果を利用してプログラム変換をさらに進めるシステムについて本稿で議論する．

2 母関数による解法の基本手順

母関数による解法は以下の5段階に分けることができる。

1. 定係数化
   再帰式の係数を定数にする。この処理はプログラム変換（一般化）により行われる。変換手順は後で使うため保持しておく。
2. 再帰式生成処理
   プログラム変換の技術を用いて、母関数の再帰式生成する。生成された再帰式の引数の数は1つ減少する。
3. 特性方程式による解法の適用
   定係数線形の再帰式は、簡単な数式処理にて解を求めることができる。この処理はREDUCEに行わせる。
4. 級数展開・抽出
   級数展開し、先の母関数定義と比較して、中身を抽出する。結果として引数の数が1つ増加する。
5. 定係数化の逆変換
   先の定係数化で保持しておいた変換手順を基に、逆変換して元の再帰式に戻す。

3 基本手順の流れ

$$F(m,n) = \text{再帰式}\ [m \geq 0, n \geq 0]$$

が入力されたとする。

1. **定係数化**
   再帰式中の関数Fの係数を見て定数でないならプログラム変換（一般化）により定数化を実行する。定係数化により、
   $G(m,n) = \text{定係数再帰式}\ [m \geq 0, n \geq 0]$
   が得られたとする。

2. **再帰式生成処理**

$$H(n) = \sum_{m=0}^{\infty} G(m,n) Z^m \qquad (*)$$

と定義し、右辺にプログラム変換を施すことにより

$$H(n) = \text{定係数再帰式}\ [n \geq 0]$$

が得られる。この処理については、あとで詳しく説明する。

3. **特性方程式による解法の適用**

$$f(m) = a_0 f(m-1) + a_1 f(m-2)$$

(但し、a_0とa_1は定数)
という再帰式は

$$t^2 - a_0 t - a_1 = 0$$

という特性方程式の解α ,β を用いて、

$$f(m) = c_0 \alpha^m + c_1 \beta^m$$

(但し、c_0とc_1は定数)
となることが分かっている。
この解法を用いてH(n)の閉じた式H'(n)を得ることができる。なお、この解法は定数項があっても変換により、適用できることが分かっている。

4. **級数展開・抽出**
   級数展開は特定の変数と環境を基準に級数の形を生成する処理で、REDUCEにより行われる。処理手順は、(a) 式を分解し、(b) その各部分を基本公式を使って級数展開にして、(c) 統合し、(d) 簡単化する。
   ex) z/(1-z) を変数zと環境n≥0を基準に級数展開する場合
   (a) zと1/(1-z)に分解
   (b) $z \Rightarrow \sum_{n\geq 0}(\text{if n=1 1 0}) z^n$
   $1/(1-z) \Rightarrow \sum_{n\geq 0} z^n$
   (c) $\sum_{n\geq 0}(\text{if n=1 1 0}) z^n * \sum_{n\geq 0} z^n$
   $\Rightarrow \sum_{n\geq 0}(\sum_{0\leq k\leq n}(\text{if k=1 1 0})) z^n$
   (d) $\Rightarrow \sum_{n\geq 0}(\text{if } n \geq 1\ 1\ 0) z^n$
   以上の手順(a)から(d)で閉じた式H'(n)を級数展開し

$$H'(n) = \sum_{m=0}^{\infty} G'(m,n) Z^m$$

が得られる。この本体を抽出し閉じた式G'(m,n)を得る。このG'(m,n)は(*)よりG(m,n)と同一視できる。

5. **定係数化の逆変換**
   定係数化で行った変換の逆を行い、初めの式に還元し、F'(m,n) を得る。これで、F(m,n) の閉じた式 F'(m,n) が得られたことになる。

4 f(m,n)=(m-n)f(m-1,n)+f(m-1,n-1) の例

以下、例を示すが見易さを配慮し、記述を以下のように定める。

1. ≤,≥,=,+,−,*,/,^は infix 記法とする。
2. and は,(カンマ) で表す。
3. 処理中に定義された関数は「関数名'('引数,...,引数')'」の形式とする。
4. その他は、LISP の形式のままとする。

入力式は初期条件も含めて以下のようにする。
関数名 f、引数 (m,n)、環境 t、
本体

```
(if m>=0,n>=0
    (m-n)*f(m-1,n)+f(m-1 n-1)
    +(if m=0,n=0 1 0) 0)
```

まず、最初の定型化が行われ、

```
(ifpls
 (if m=0,n>=1
     (m-n)*f(m-1,n)+f(m-1,n-1) 0)
 (if m>=1,n>=0
     (m-n)*f(m-1,n)+f(m-1,n-1) 0)
 (if m=0,n=0 1 0))
```

となる。if 文の条件部がそれぞれ互いに素であることに注意。

―――<定係数化処理>―――

両辺を (m-n)!で割ることで定係数化すると、

```
(ifpls
 (if m=0,n>=1 f(m-1,n-1)+f(m-1,n) 0)
 (if m>=1,n>=0 f(m-1,n-1)+f(m-1,n) 0)
 (if m=0,n=0 1 0))
```

となる。この変換手順は、後で逆に適用する必要があるため、記憶しておく。

―――<再帰式生成処理>―――

プログラム変換の技術を用いて、(infsum f(m,n) m t z) に再帰式生成処理を行うと、以下の 3 つの関数定義が生成される。

```
g0(n)=
 (ifpls
  (if n=0 1/(1-z) 0)
  (if n>=1 z*g2(n)+z*g2(n-1) 0))
g1(n)=0
g2(n)=
 (ifpls
  (if n=0 1/(1-z) 0)
  (if n>=1 z*g2(n-1)/(1-z) 0))
```

ここでは再帰式 g2(n) を解く必要がある。詳しい内容は 9 章で述べる。

―――<特性方程式による解法の適用>―――

これで、g2(n) は 1 引数定係数再帰式なので、特性方程式による解法を適用すると、

```
g2(n)=(if n>=0 ((z/(1-z))^n)/(1-z) 0)
```

となる。

(ここで、g0(n) の閉じた式を得るために、g0(n) の定義内の関数 g2 の部分に求めた閉じた式を展開する。というのは、この直後の級数展開の対象となるのが関数 g2 ではなく関数 g0 だからである [9 章を参照]。しかし、関数 g0 と関数 g2 は同じ閉じた式となる。)

―――<級数展開・抽出>―――

if 文本体である((z/(1-z))^n)/(1-z) を級数展開すると

```
(infsum
  (if m>=0
   (sum
    (if 0<=n<=lc15
        bin(lc15-1,lc15-n) 0)
    lc15 0<=lc15<=m) 0)
  m t z)
```

となる。lc15 は新たな局所変数である。infsum 文から本体を抽出して

```
(if n>=0 (if m>=0
  (sum
   (if 0<=n<=lc15
       bin(lc15-1,lc15-n) 0)
   lc15 0<=lc15<=m) 0) 0)
```

が得られる。

―――<定係数化の逆変換>―――

定係数化では (m-n)!で割ったので、今度は逆に掛けてやり、その後定型化を行うと、

```
(ifpls
 (if m>=0,n=0 fact(m) 0)
```

```
(if m>=n,n>=1 fact(m-n)*bin(m,n) 0))
```

となる。ここでは、

$$\sum_{m<k<n} bin(k-1,k-n) => bin(m,n)$$

という変換をしている。これで、

```
f(m,n)=
 (ifpls
  (if m>=0,n=0 fact(m) 0)
  (if m>=n,n>=1 fact(m-n)*bin(m,n) 0))
```

という解が得られたが、さらなる簡単化により

```
(if m>=n,n>=0 product(k,k,n+1,m) 0)
```

となり、(m+1)(m+2)···(n-1)n という簡単な式が得られていることが分かる。

5 システムの概要

大きく分けて3つの部分からなる。

- Gsolver 核
  再帰式を手順に従い解く部分
- 定型化処理系
  式の定型化・簡略化を行う部分
- REDUCE 制御
  REDUCE の呼び出し、LISP-REDUCE 間の式変換を行う部分

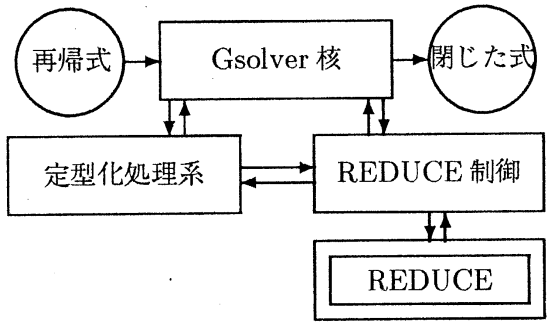

図1：GSolver システム概形

6 システムの詳細

6.1 Gsolver 核

本システムの中枢にあたり、ここから定型化処理系、REDUCE 制御に指令を出す。変数定義、関数定義、マップ定義等の重要なデータは KERNEL というクラスに保持させている。後述の処理手順を入力式を見て、実行する。

6.2 定型化処理系

算術式を処理する算術式定型化処理系と、論理式を処理する論理式定型化処理系からなる。

算術式定型化処理系は、算術式を以下の形式のいずれかにする。

```
ifpls-s  :=  ('ifpls' if-s .... if-s)
  if-s   :=  ('if' 論理式 arith-s 0)
arith-s  :=  ('infsum' arith-s n e(n) z) |
             ('sum' arith-s n e(n)) |
             ({'+'|'-'|'*'} arith-s ... arith-s) |
             ('/' arith-s arith-s) |
             ('bin' arith-s arith-s) |
             ('sqrt' arith-s) |
             ('fact' arith-s) | atom
  atom   :=  数字 | シンボル
```

ifpls 文は通常の+とほぼ同じ働きをする関数であるが、引数が if 文のみであり、各 if 文の条件部は互いに素である。(P と Q が互いに素とは、(and P Q)=nil のことである)

infsum(A,n,e(n),z) は $\sum_{e(n)} Az^n$ を表す。e(n) は n の最小値、最大値を一意に規定できるような論理式である。

sum(A,n,e(n)) は $\sum_{e(n)} A$ を表す。e(n) については、関数 infsum と同様。sqrt(n),fact(n),bin(m,n) はそれぞれ平方根 $\sqrt{n}$, 階乗 $n!$, 二項係数 $\binom{m}{n}$ を表す。arith 文は、一回 REDUCE により整形させる。

論理式定型化処理系は、論理式を以下の形式のいずれかにする。

```
      or-s  :=  (or and-s ... and-s)
    and-s   :=  (and 単純論理式 ... 単純論理式)
単純論理式  :=  ({'≥'| '≤'| '='} 算術式 算術式)
```

但し、or 文の各引数は互いに素にし、and 文は任意の変数について最小値と最大値が必ず一意に決まるようにする。>および<は、≥と≤を使って書き変えられ、not 文は変形により消される。本システムでは、無限小および無限大をそれぞれ、ninf および pinf と表している。[注意:infsum 文 (infsum a n e(n) z) について、a,n,e(n),z をそれぞれ、その infsum 文の本体, 基準変数, 環境部, 底と呼ぶことにする]

6.3 REDUCE 制御

Reduce クラスを中心に、LISP-REDUCE 間の式変換、REDUCE エラー処理などを行う。LISP

からは処理命令とともに関数 red-exec を呼ぶことにより簡単に REDUCE 処理結果を LISP 形式で得ることができる。REDUCE を呼び出すことは、他の処理に比べて時間がかかるため、REDUCE を呼ぶ回数を減らすことが、システム全体の処理を速くすることにつながる。実際には、同じ処理をしないために、以前の処理結果をヒストリとして、Reduce クラスに管理させている。red-exec を呼ぶ側も、できるだけ red-exec を呼ばないような仕組みを取り入れる必要がある。

7 再帰式生成処理の概要

1. infsum 文からマップを定義する (DEFMAP)
2. infsum 文の本体を展開する (UNFOLD)
3. infsum 文を分配する (DISTRIBUTE)
4. infsum 文を簡単化する (SIMPLIFY)
5. infsum 文を畳み込む (FOLD)
6. 左辺値について解く処理 (GTRANS)
7. 関数を定義する (DEFUN)

8 再帰式生成処理の詳細

8.1 マップ定義 (DEFMAP)

FOLD 処理で使われる FOLD マップを定義する。

マップは、[新関数名 旧関数名 引数の数 第1引数の最小値 第1引数の最大値 底] という形式をしていて、FOLD 条件と FOLD 後の形式が記されている。

ex) (infsum f(m,n) m q≥m≥p Z) に対しては [g f 2 p q Z] というマップを定義する。g は新たに gensym で生成された関数名である。このマップにより、第1引数の最小値が p 最大値が q である2引数関数 f を本体に持ち、底が Z である infsum 文はすべて関数 g を使って FOLD されうる。

8.2 展開 (UNFOLD)

infsum 文本体を展開する。展開は関数定義を参照するため、その関数が定義されていなければならないが、これは保証されている。

8.3 分配 (DISTRIBUTE)

以下の分配規則に従い、処理が進む。

[+,−,ifpls] 通常の分配でよい。

[if] if 文の条件部の内、infsum の基準変数に無関係なものは if 文の条件部のまま外へ出し、関係のあるものは infsum 文の環境部に追加する。この時、環境部が or 文になった場合は、ifpls 文にする必要がある。(infsum (if p a 0) n e(n) z)
=>(if q (infsum a n (and e(n) pn) z) 0)
(但し、p=(and pn q) とし、pn は n に関係する論理式、q は n を含まない論理式とする)

[/] (infsum (/ a b) n e(n) z)
=>(/ (infsum a n e(n) z) b)
ただし、b が n に依存したらエラー

[*] (infsum (* a1 ... an) n e(n) z)
=>(* a1 .. (infsum ai n e(n) z) .. an)
ただし、$aj[1 \leq j \leq n, j \neq i]$ が n に依存したらエラー

8.4 簡単化 (SIMPLIFY)

以下の簡単化規則に従い、処理が進む。

- 本体が 0 なら 0 を返す
- 最大値、最小値が数字なら sum 文として計算
- (infsum A(n) n (= n k) z) なら $A(k)z^k$ にする
- (infsum A(n-k,d(m)) n e(n) z) は z^k*(infsum A(n,d(m)) n e(n+k) z) にする

8.5 畳みこみ (FOLD)

FOLD 候補の式 (infsum f(m,…) m e(m) z) を簡単な関数に置換する処理である。FOLD 候補の式は FOLD マップを探索し、自分の条件にあったマップを見つけ、それにしたがって FOLD される。

ex)[g f 2 0 pinf z] という FOLD マップが定義されているなら、(infsum f(m,d(n)) 0 pinf z) は g(d(n)) と FOLD される。

もし、FOLD マップが存在しなければそれを定義するために再帰的に再帰式生成処理を行う必要がある。再帰式生成処理により必要なマップは新たに定義され FOLD 可能になる。(実際には、マップを扱うクラスが定義されていて、そのメソッドを呼べば、マップの登録、マップの定義確認、マップの適用などができるようになっている。)

8.6 左辺値について解く処理 (GTRANS)

左辺値について、式を解きなおす処理。この処理は各 if 文毎に行われる。GTRANS 前には必ず、定型化されている必要がある。

ex)f(n)=(ifpls (if p 1 0) (if q z*f(n)+f(n-1) 0)) の GTRANS 処理後は (if q z*f(n)+f(n-1) 0) に対してだけ、f(n)=z*f(n)+f(n-1) より f(n)=(1/(1-z))*f(n-1) となるため (ifpls (if p 1 0) (if q (1/(1-z))*f(n-1) 0)) となる。

8.7 関数定義 (DEFUN)

ここまでで得られた新たな再帰式を定義する。マップ定義で生成された関数はここで初めて定義される。

9 f(m,n)=(m-n)f(m-1,n)+f(m-1,n-1) の例の再帰式生成処理

4 章で f(m,n)=(m-n)f(m-1,n)+f(m-1,n-1) の例を示したが、再帰式生成処理の説明はしていなかったので、ここで説明する。

(infsum f(m,n) m t z) 対して再帰式生成処理を行う。(4 章の再帰式生成処理を参照)

———(infsum f(m,(n) m t z) の開始———

<DEFMAP> [g0 f 2 ninf pinf z] を定義する。

<UNFOLD →⋯→ SIMPLIFY>

```
(ifpls
 (if n≧1
    z*(infsum f(m,n) m m=-1 z)
   +z*(infsum f(m,n-1) m m=-1 z) 0)
 (if n≧0
    z*(infsum f(m,n) m m≧0 z)
   +z*(infsum f(m,n-1) m m≧0 z) 0)
 (if n=0 1 0))
```

<FOLD> FOLD 候補の式は以下の 4 つである。

```
(infsum f(m,n) m m=-1 z)
(infsum f(m,n-1) m m=-1 z)
(infsum f(m,n) m m≧0 z)
(infsum f(m,n-1) m m≧0 z)
```

(infsum f(m,n) m m=-1 z) は FOLD できないため、再帰式生成処理をする。

———(infsum f(m,n) m m=-1 z) の開始———

<DEFMAP> [g1 f 2 -1 -1 z] を定義する。

<UNFOLD → ⋯ → SIMPLIFY> 0 となる。

g1(n)=0 と定義される。

———(infsum f(m,n) m m=-1 z) の終了———

以下の 2 つが FOLD できるようになった。

```
(infsum f(m,n) m m=-1 z) ⇒ g(n)
(infsum f(m,n-1) m m=-1 z) ⇒ g(n-1)
```

まだ、(infsum f(m,n) m m>=0 z) が FOLD できないため、再帰式生成処理をする。

———(infsum f(m,n) m m>=0 z) の開始———

<DEFMAP> [g2 f 2 0 pinf z] を定義する。

<UNFOLD → ⋯ → SIMPLIFY>

```
(ifpls
 (if n>=1
   z*(infsum f(m,n-1) m m=-1 z)
  +z*(infsum f(m,n) m m=-1 z)  0)
 (if n>=0
   z*(infsum f(m,n-1) m m>=0 z)
  +z*(infsum f(m,n) m m>=0 z)  0)
 (if n=0  1 0))
```

<FOLD> FOLD 候補の式は以下の 4 つでありすべて FOLD できる。

```
(infsum f(m,n) m m=-1 z) ⇒ g1(n)
(infsum f(m,n-1) m m=-1 z) ⇒ g1(n-1)
(infsum f(m,n) m m>=0 z) ⇒ g2(n)
(infsum f(m,n-1) m m>=0 z) ⇒ g2(n-1)
```

定型化の後 GTRANS して、

```
(ifpls
 (if n=0 1/(1-z) 0)
 (if n≧1 (z/(1-z))*g2(n-1) 0))
```

となり、

```
g2(n)=(ifpls
        (if n=0 1/(1-z) 0)
        (if n≧1 (z/(1-z))*g2(n-1) 0))
```

と定義される。

———(infsum f(m,n) m m>=0 z) の終了———

残りの 2 つも FOLD できるようになった

```
(infsum f(m,n) m m≧0 z) ⇒ g2(n)
(infsum f(m,n-1) m m≧0 z) ⇒ g2(n-1)
```

これですべて FOLD された。

定型化後 GTRANS して、

```
(ifpls
 (if n=0 1/(1-z) 0)
 (if n≧1 z*g2(n)+z*g2(n-1) 0))
```

となり、

```
g0(n)=(ifpls
        (if n＝0 1/(1-z) 0)
        (if n≧1 z*g2(n)+z*g2(n-1) 0))
```

と定義される。

———(infsum f(m,n) m t z) の終了———

以上で以下の３つが定義された。

```
g0(n)=(ifpls
        (if n＝0 1/(1-z) 0)
        (if n≧1 z*g2(n)+z*g2(n-1) 0))
g1(n)=0
g2(n)=(ifpls
        (if n=0 1/(1-z) 0)
        (if n≧1 (z/(1-z))*g2(n-1) 0))
```

これより、g2(n) の再帰式を解けば g0(n) の閉じた式が得られることが分かる。

なお、実は g0(n) と g2(n) の閉じた式は同じ式となる。これは、g2 で FOLD した部分が g0 で FOLD できることを示している。このようなことは、f の変域と infsum 文の環境との関係で起ることが分かっている。現在はこの状況で FOLD することは行われていないが、行われればさらに簡単に再帰式および閉じた式が求まる。

10 FOLD 依存木と FOLD 可能性

再帰式生成処理は、初めにマップ定義をしてから、UNFOLD-DISTRIBUTE/SIMPLIFY を経て、FOLD 処理または再帰的に再帰式生成処理を行う。FOLD 処理ができるということは、対応するマップが存在することを意味する。以上のことをふまえて、FOLD 依存木を以下のように定義する。

- ノードは１回の再帰式生成処理を表し, 丸の中にマップ定義で新しく定義した関数名を書く
- 定係数化の後、再帰式生成処理を実行する命令がでた時、それが根となる。
- 再帰式生成処理中で FOLD できずに再帰的に再帰式生成処理を呼んだとき子ができ、呼ばれた順に右から並べる

また、再帰式生成処理内で FOLD 処理された場合は、必ず以前にマップ定義されているはずなので、そのマップを定義した再帰式生成処理に対応したノードへの有向辺を FOLD 依存木に加える (FOLD 弧と呼ぶ)。

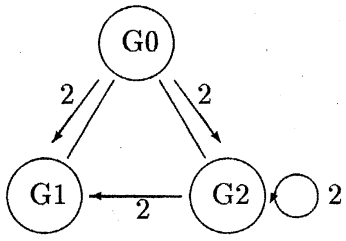

図２：(m-n)*f(m-1,n)+f(m-1,n-1) の場合の FOLD 依存木と FOLD 弧

この木には次のような特徴がある。

- 子の数は再帰式生成処理を再帰的に行った回数、すなわち FOLD に失敗した回数を表す。
- FOLD 弧が後退弧の場合の終点の再帰式生成処理はまだ完了していないため、FOLD 後の式は未定義なので展開も評価もできない。
- FOLD 弧が先行弧および交差弧の場合の終点の再帰式生成処理は完了しているため、マップ定義で生成した新関数が定義済みである。(再帰式生成処理の最後に関数定義を行うことに注意) したがって、FOLD 後の式は展開も評価もできる。
- 葉は、再帰的に再帰式生成処理をしなくても再帰式化できた時にできる。

あるノードから出ている FOLD 弧に後退弧がない場合は、自分自身への FOLD 弧以外は FOLD 後の式が評価できるので、生成された再帰式を解くことができる。しかし、後退弧がある場合は注意が必要である。なぜなら、後退弧を出した FOLD 候補の式は、未定義関数に FOLD されるからである。未定義関数がどんな振る舞いをするかは計り知れないため、生成された再帰式を解くことはできなくなる。しかし、再帰式を求めている最中に閉じた式 (未定義関数を含む) が得られてしまった場合は、そのまま定義して親ノードに処理を返すことができる。この問題は相互再帰が解けない原因にもなっており、解決策を考えなければならない。

11 おわりに

２引数再帰式を解くシステム Gsolver の説明をしてきたが、基本的な考えは三引数以上の場合にも適用できる。本システムで再帰式が解けるか解けないかは以下の４つの壁を突破できるかがカギである。

1. 定係数化できるか

   定係数化不可能な再帰式は、当然特性方程式に

よる解法は使えず、本システムでは母関数による解法も適用できないためエラーとなる。(未対応だが、微分を用いて適用できる場合がある。その場合は微分方程式を解く必要が出てくる)

2. 再帰式生成処理が停止するか
   再帰式生成処理では、マップ定義がないとFOLDできずにいつまでも再帰式生成処理を再帰的に呼ぶ可能性がある。
3. 生成された再帰式に未定義関数が含まれていないか
   生成された再帰式に未定義関数が含まれると、再帰式が閉じた式にならないため即エラーを返し終了する。
4. 級数展開多くの数学的知識をREDUCEに教え込む必要がある。また、どうしても級数化できない場合もある。

今後の課題としては以下のようなものが挙げられる。

- 指数型母関数への対応
- 定係数化可能な式の拡大
- 他の再帰式閉式化プログラムの組込み
- より高度な数式処理系Macsimaの活用
- 相互再帰の対応

最後に、本稿の作成にあたり，貴重な議論をしてくださった早稲田大学理工総研 小西善二郎氏に深謝いたします。

参考文献

[1] Burstall, R.M. and Darlington, J.: A Transformation System for Developing Recursive Programs, JACM, Vol.24, No.1, 1977, pp.44-67.

[2] Graham, R.L., Knuth, D.E. and Patashnik, O.:Concrete Mathematics, Addison-Wesley, 1989.

[3] Hearn A.C.: REDUCE - A Case Study in Algebra System Development, Lecture Notes on Comp. Science, No. 144, Springer-Verlag, Berlin, 1982

[4] Pettorossi, A. and Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs, ACM Computing Surveys, Vol.28, No.2, June 1996, pp.360-414.

[5] 二村良彦，矢農正紀： 実際的プログラム変換の例， 日本ソフトウェア科学会第14回大会，1997年9月

[6] 川本史生，中村充司，小西善二郎，湯浅能史，二村良彦： プログラム変換と数式処理システムの融合， 日本ソフトウェア科学会第14回大会，1997年9月

[7] 川本史生，二村良彦： 数式処理系REDUCEによるプログラム変換， 日本ソフトウェア科学会大会論文集B4-1，1998年9月

[8] 坂本巨樹，二村良彦： 分割統治法に基づくアルゴリズムの計算量自動評価の試み， 日本ソフトウェア科学会大会論文集B4-2，1998年9月