数理解析研究所講究録
1514巻 2006年 229-235

Risa/Asir における新しい形式の数式の取り扱いについて

野呂 正行, 高山信毅
(神戸大理)

概要

Risa/Asir における柔軟な数式の取り扱いを実現するため, 木構造で表現された数式である FNODE 構造体を保持する QUOTE 型を扱えるようにした. 標準化をはじめとする QUOTE に対する種々の操作, およびパターンマッチングによる書き換えを実装した. これらを用いたいくつかの例を示す. また, weight を用いた非可換代数における書き換えの可能性について述べる.

1 Risa/Asir における数式の取り扱い

Risa/Asir においては, ユーザにより入力された数式は, いったん FNODE と呼ばれる木構造に変換されたのち, eval() により再帰的に正規内部表現 (Risa オブジェクト) に変換される. Risa オブジェクトとは, 先頭に共通の識別子フィールドを持つ一群の構造体であり, arf_add() などのトップレベル演算関数は, 受け取った構造体の識別子を見て, 適切な関数に振り分けるという操作を行う. Risa オブジェクトとしては, 数, 多項式, 有理式, リスト, 配列など 30 種類弱が定義されている. さらに, 数は, 有理数, 浮動小数, 有限体などさらに細かく分類される. いったん Risa オブジェクトに変換されてしまえば, それぞれ固有の方法により, 効率よい演算が適用できるが, 一方で, たとえば多項式が強制的に展開されてしまうなど, 本来の入力が持っていた情報が失われることもある. また, 原則として多項式の積は可換と仮定されているため, 微分作用素など, 非可換な対象を扱う場合に不自然な操作を強いられて来た.

例 1 dx を $\partial/\partial x$ の意味に使おうと思っても

```
[0] x*dx;
dx*x;
[1] dx*x;
dx*x;
```

のように, 勝手に順序が変えられてしまう.

また, 以前から指摘されている, Risa/Asir に式の簡単化機能が欠如している点についても, あらゆるものを多項式に変換してから簡単化するのは不自然である.

このような, 数式の柔軟な取り扱いは, Maxima, Maple, Mathematica などの得意とするところであり, これまでは, Risa/Asir の目指すところは, 多項式演算の高速処理であるとして, 特にこのような方向の開発は進めてこなかった. しかし, 使われ方が多様化した結果, より多様な数式の取り扱いが必要となる場面が多くなってきたため, より一般の数式の演算および簡単化, 書き換え規則による書き換えの実装に着手した.

2 QUOTE 型

前節で述べたように, Risa/Asir においては, 入力された数式は, Risa オブジェクトに変換される前に, FNODE と呼ばれる木構造で保持されている. この FNODE をボディ部に持つ Risa オブジェクトである QUOTE 型を定義した. これにより, 評価前の木構造を保持できる.

2.1 QUOTE の入力, 基本操作

QUOTE 型に対する四則演算などの基本演算は, 木に対する操作として定義する. さらに, 木に対する一般的な操作 (属性, 子の取り出し, 木の再構成など) を Asir の関数として与えることで, ユーザによる数式の操作が可能となる. QUOTE 型に対する操作は, 実際には FNODE に対する操作である. FNODE は

$$(id\ arg_0\ arg_1\ \ldots)$$

というリストで表現され, arg_i の個数, 型は id によりさまざまである. QUOTE の入力, 変換などの基本操作は次の通りである.

- QUOTE の入力

  QUOTE は quote(*Expr*) または '*Expr* (バッククォートつき) により入力できる.

- QUOTE と Risa オブジェクトの相互変換

  Risa オブジェクトから QUOTE を生成するのは objtoquote(*Obj*), 逆に QUOTE を評価して Risa オブジェクトを生成するのは eval_quote(*Expr*) で行う.

- QUOTE の分解, 合成

  quote_to_funargs(*Expr*) は QUOTE*Expr* の FNODE の識別子, 引数をリストとして返す.

  funargs_to_quote(*List*) は, その逆である.

2.2 FNODE の標準形

FNODE に対するパターンマッチング, 書き換えを容易に行うために, FNODE に対する標準形を定義した. 標準形の計算は qt_normalize(*Expr*[,*Mode*]) で行う. *Mode* は後述する展開モード指定である.

nf	: $formula \mid functor\ (nf\ [, \ldots]) \mid sum\_of\_monom$
sum_of_monom	: $monom\ [+\ \cdots]$
$monom$	: $[formula\ *\ ]\ nfpow\ [*\ \cdots]$
$nfpow$	: $nf \mid nf^{nf}$
$formula$	: Risa object

おおざっぱにいえば, 標準形 nf とは, 標準形のベキ積の Risa オブジェクト係数つきの和である. ここで, 和は FNODE としては, n 項和として表現され, 和を構成する単項式は, ある全順序により整列される. また, 積も n 項積として表現される. すなわち, 標準形は, 入力された数式が, Risa オブジェクトを係数環とする結合代数の元であると見なし, 和の可換性, 積の結合性によりフラットに整理しなおしたものである.

例 2

```
[278] ctrl("print_quote",1)$    /* FNODE をリストで表示 */
[279] '(x+y+z);
[u_op,(),[b_op,+,[b_op,+,[internal,x],[internal,y]],[internal,z]]]
[280] qt_normalize('(x+y+z));
[n_op,+,[internal,x],[internal,y],[internal,z]]
```

2 項演算で表現された式が, 標準形では n 項和で表現されていることが分かる.

これは, Mathematica における標準形 [1] と基本的に同じであるが, 積の可換性を仮定していないこと, および, 係数環をより一般的にしてある点で異なっている[1]. さらに, 標準形への変換時に, 積に関する分配則を利用して展開された標準形を得ることもできる.

[1] Mathematica において積の Orderless 属性を外すことで, 積を非可換にできるが, 簡単化において異常な挙動を示すようになる (Ver. 4). Ver. 5 の初期の版では, 係数まで非可換になったが, 最近のものでは直っているようである.

例 3

```
[289] ctrl("print_quote",2)$    /* FNODE を式で表示 */
ctrl(''print_quote'',2)$
[290] qt_normalize('(x+y)^2);
((x)+(y))^(2)
[291] qt_normalize('(x+y)^2,1);
((x)^(2))+((x)*(y))+((y)*(x))+((y)^(2))
```

2.3 項順序および係数環の設定

単項式順序および係数環は可変であり, それぞれ次のような関数が用意されている.

- 単項式順序の設定

  標準形中の単項式順序は, 指定がない場合には, システムが決める不定元および関数子の順序から誘導される辞書式順序が適用される. その際, 関数呼び出しは, 単なる不定元より順序が上で, 関数子が等しい場合には引数が辞書式に比較される. qt_set_ord(*VarList*) により, *VarList* に現れる不定元を先頭とし, のこりをシステムが決めるという順序が設定される.

- 係数環の設定

  デフォルトでは係数環は数のみからなるが, いくつかのパラメタを係数環の元として扱いたい場合, qt_set_coef(*ParamList*) により指定できる. *ParamList* に指定されたパラメタは, 係数環である可換な有理関数体の不定元として扱われる.

例 4

```
[304] qt_normalize('(b*x+a*y)*b*y,1);
((a)*(y)*(b)*(y))+((b)*(x)*(b)*(y))
[305] qt_set_coef([a,b])$
[306] qt_normalize('(b*x+a*y)*b*y,1);
((b^2)*(x)*(y))+((b*a)*((y)^(2)))   /* a,b が係数環に入った; x*y > y^2 */
[307] qt_set_ord([y,x])$
[308] qt_normalize('(b*x+a*y)*b*y,1);
((b*a)*((y)^(2)))+((b^2)*(x)*(y))   /* y^2 > x*y */
```

3 パターンマッチングによる書き換え

Risa/Asir においては, 不定元とプログラム変数は明確に区別されている. そこで, パターン変数としてプログラム変数を用いることにした. すなわちパターンとは, プログラム変数を含んでもよい QUOTE である. これに対し, いくつかの書き換え関数を用意した.

- nqt_match(*Expr*,*Patten*[,*Mode*])

  QUOTE 式 *Expr* とパターン *Pattern* がマッチしたら 1 を返す. さらに, *Pattern* 中に含まれるプログラム変数にマッチした値が実際に代入される.

- nqt_match_rewrite(*Expr*,*Rule*,*Mode*)

  Rule は [*Pattern*,*Action*] または [*Pattern*,*Condition*,*Action*] である. この関数は, *Expr* が *Pattern* にマッチしたら, *Action* が評価され, その値が返される. その際, *Action* 中のパターン変数が, マッチした値に置き換えられる. *Condition* が指定されている場合には, *Condition* 中のパターン変数が同様

に置き換えられ評価され, 0 でない場合に *Action* が評価される. マッチしない場合には *Expr* そのものが返される.

例 5

```
[318] nqt_match('x*y*z-3*u,'X*Y+Z);
1
[319] [X,Y,Z];
[x,(y)*(z),(-3)*(u)]
[320] nqt_match_rewrite('x*y*z,['X*Y,'X+Y],1);
((y)*(z))+(x)
```

いずれも実行前に引数が標準形に変換されるが, *Mode* はその際に展開を行うかどうかを指示する. マッチングにおいては, 最初にマッチした時点の情報が返される[2]. *Condition* および *Action* にはユーザ定義関数を含めることができる. これにより, 複雑な書き換え規則を書くことができ, また書き換え規則の数を少なく押えることができる. 現状では, Mathematica で可能な, パターン変数にマッチする型の指定ができないため, *Condition* において型判定を行うことになる. このため, QUOTE に対するいくつかの型判定関数を用意した. これらを用いて, 書き換え規則集合を与えて, 書き換え規則が適用できなくなるまで書き換えを続ける関数 qt_rewrite(*Expr*,*Rules*,*Mode*) をユーザ関数として記述した.

例 6 (sl_2 の展開環)

```
[336] Rsl=[['h*e,'e*h+2*e],['h*f,'f*h-2*f],['e*f,'f*e+h]]$
0sec(7e-06sec)
[337] qt_rewrite('e*f^2,Rsl,2);
((f)*(f)*(e))+((2)*(f)*(h))+((-2)*(f))
1.776e-15sec(0.008608sec)
[338] qt_rewrite('h*e^3,Rsl,2);
((e)*(e)*(e)*(h))+((6)*(e)*(e)*(e))
```

4 FNODE の順序づけ

今回の実装の目的は, ユーザが気軽に書き換え規則を与えて, 一般に非可換な代数における計算を気軽に試せるような環境を作ることである. 与えられた書き換え規則の停止性, あるいは合流性に関しては, 項書き換え系の研究者による研究が膨大にあるが, ここでは深入りはしない. ここでは, 無限ループに陥らないような実用的な指針として, FNODE に対する順序づけおよび weight の使用を提案する. この方法は後述するように多項式環や微分作用素環で用いられる weight ベクトルの考え方の自然な一般化であり, 理論的にも興味深い.

例として, 可換性を定義する場合を考える. 数学的には, 任意の X, Y に対し $XY = YX$ でよいが, たとえばこのまま ['$X*Y$,'$Y*X$] という書き換え規則を書くともちろん停止しない. この場合, 最も安直な解決方法の一つは, FNODE 間に全順序を入れて, 書き換えた場合に順序が大きく (小さく) なる場合にのみ書き換えを行うという方法である. この場合, 積を構成する有限個の FNODE の並べ変えの中で最も順序が上 (下) のものに到達すると停止する. *Action* が複雑な場合にはこのように簡単には行かないが, 書き換えの方向性を示すものとして全順序を与えることは有効であろう. よって, 書き換え規則に応じて, 全順序をどう選ぶかが問題である.

4.1 FNODE の weight と書き換え

一般に FNODEf の weight $w(f)$ を

[2]現状では実装が不完全であり, 同一パターン変数が複数現れるパターンに対してはマッチングに失敗する場合がある.

1. f が leaf の場合, 適当な値を与える. 特に係数の weight は 0.

2. f が node の場合, f の子の weight 値を引数とし, 識別子で決められた関数を計算してその値をとる.

により再帰的に決めることができる. 和に対しては max(), 積に対しては和, ベキに対しては積を用いると, 次のようになる.

1. $w(f+g) = \max(w(f), w(g))$

2. $w(fg) = w(f) + w(g)$

3. $w(f^n) = nw(f)$

以下では, このような weight を有限生成の自由結合代数に対する書き換えに応用することを考える.

係数環を K の上で $z_1, \ldots, z_n, h$ で生成される自由結合代数 A を $K\langle z_1, \ldots, z_n, h\rangle$ と書く. h を必要に応じて z_{n+1} と書くこともある.

定義 1 A での書き換え規則 (または関係式, 左辺は必ず単項式) $L_1 \to R_1, \ldots, L_m \to R_m$ が, 同次化 weight ベクトル H について, 同次的書き換え規則であるとは, R_i が 0 であるかまたは, $\deg_H(L_i) = \deg_H(R_i$の任意の項$)$ が成立することである.

ここで $\deg_H(\prod z_i^{e_i})$ は $\prod z_i^{e_i}$ の weight H についての (非可換性を無視した) 次数である. つまり $\deg_H(\prod z_i^{e_i}) = \sum e_i H_i$ と定義する (i は重複してあらわれることもある).

例 7 $z_2 z_1 \to z_1 z_2 + h^2, hz_i \to z_i h$ は $H = (1,1,1)$ についての同次的書き換え規則である. この例は $x = z_1, \partial = z_2$ とした 1 変数の同次化 Weyl 代数にほかならない.

以下 H のすべての成分は正であると仮定し固定する. また $x_1, \ldots, x_n, h$ からなるワードに対する well order $\succ$ を以下ひとつ固定する. 出現する書き換え規則はとくにことわらない限り全て H について同次的書き換え規則である.

例 8 前の例の書き換え規則 $z_2 z_1 \to z_1 z_2 + h^2, hz_i \to z_i h$ にさらに $z_2^{p+1} \to 0, z_1 z_2 \to p h^2$ を加えた規則の集合を R_p と書く. ここで p は自然数である. R_p は $H = (1,1,1)$ についての同次的書き換え規則である.

定義 2 n 次元の weight ベクトル $w \in \mathbf{R}^n$ が同次的書き換え規則 $\{L_i \to R_i\}$ および $\succ$ について有効 weight ベクトル (admissible weight vector) であるとは次の条件をみたすことである. 以下 $\tilde{w} = (w, 0)$ (h に対する weight を 0 にしたもの) とおく.

1. $\deg_{\tilde{w}}(L_i) \geq \deg_{\tilde{w}}(R_i)$

2. 左辺と右辺が同じ w-次数をもつときは 右辺で左辺と同じ w-weight を持つ項たちは順序 $\succ$ でかならず小さい.

書き換え規則がある正数ベクトル H について同次的であることから, これらの条件により書き換えが停止性をもつことが分かる. さらに, G-algebra [2] の条件のうち, well order の存在条件を仮定しなくても, 適当な同次化 weight ベクトル, 有効 weight ベクトルが存在するならば, h を加える斉次化, h を 1 とおくことによる非斉化により, グレブナー基底を計算できるようになると予想される.

この応用に際しては, 与えられた書き換え規則に対し, 有効 weight ベクトル w を見つける必要がある. たとえば一変数ワイル代数の場合 $w_1 + w_2 \geq 0$ の条件をみたさないと有効 weight ベクトルとならない. このとき同時化 weight ベクトルを用いて書き換え規則の右辺を斉次化すれば, 同次的書き換え規則が得られる.

現在の実装においては, weight ベクトルが設定されない限り, weight による比較は行わない. 関数 `qt_set_weight()` により一部の不定元に対して weight が設定されると, 他の不定元の weight は自動的に 0 となる. この weight を用いた 次数の比較後に現在設定されている単項式順序が適用される.

例 9

```
[300] qt_set_ord([z1,z2,h])$
[301] qt_set_weight([[z1,-1],[z2,1]])$
[302] Rule1=[[‘h*z1,‘z1*h], [‘h*z2,‘z2*h], [‘z2*z1,‘z1*z2+h^2]] $
[303] Rule2=[[‘z2*z2,‘0], [‘z1*z2,‘h^2]]$
[304] F=‘z2^2*(h^2+z1^2)$
[305] qt_rewrite(F,Rule1,2);
((z2)*(z2)*(h)*(h))+((z1)*(z1)*(z2)*(z2))+((4)*(z1)*(z2)*(h)*(h))+((2)*(h)*(h)*(h)*(h))
```

注意 1 有効 *weight* ベクトルが負の成分をもつと非斉次化したあとの *reduction* の停止性はいえない.

5 書き換え規則の例

以下に, 書き換え規則の例をいくつか紹介する.

例 10 (可換性)

```
[246] qt_normalize(‘(x+y-z)^2,1);
((x)^(2))+((x)*(y))+((-1)*(x)*(z))+((y)*(x))+((y)^(2))+((-1)*(y)*(z))
+((-1)*(z)*(x))+((-1)*(z)*(y))+((z)^(2))
[247] Rcomm=[[‘X*Y,‘nqt_comp(Y*X,X*Y)>0,‘Y*X]]$
[248] qt_rewrite(‘(x+y-z)^2,Rcomm,1);
((x)^(2))+((2)*(x)*(y))+((-2)*(x)*(z))+((y)^(2))+((-2)*(y)*(z))+((z)^(2))
```

nqt_comp() は比較関数である.

例 11 (外積代数)

```
[249] Rext0=[‘X*Y,‘qt_is_var(X) && qt_is_var(Y) && nqt_comp(Y,X)>0,‘-Y*X]$
[250] Rext1=[‘X^N,‘eval_quote(N)>=2,‘0]$
[251] Rext2=[‘X*X,‘0]$
[252] Rext=[Rext0,Rext1,Rext2]$
[253] qt_set_coef([a,b,c])$
[254] qt_rewrite(‘(a*x+b*y+c*z)*(b*x+c*y+a*z)*(c*x+a*y+b*z),Rext,1);
(-a^3+3*c*b*a-b^3-c^3)*(x)*(y)*(z)
```

行列式の計算に相当する. 変数の積を交代的に書き換える規則を定義している.

例 12 (微分)

```
[255] qt_set_coef([a])$
[256] Rd1=[‘d(X+Y),‘d(X)+d(Y)]$
[257] Rd2=[‘d(X*Y),‘d(X)*Y+X*d(Y)]$
[258] Rd3=[‘d(N),‘qt_is_coef(N),‘0]$
[259] Rd=[Rd1,Rd2,Rd3]$
[260] qt_rewrite(‘d((x+a*y)^2),Rd,1);
(d((x)^(2)))+((a)*(d(x))*(y))+((a^2)*(d((y)^(2))))+((a)*(d(y))*(x))
+((a)*(x)*(d(y)))+((a)*(y)*(d(x)))
```

例 13 (Weyl 代数)

```
def member(V,L) {
  for ( I = 0; L != [] && V != car(L); L = cdr(L), I++ );
  return L==[] ? -1 : I;
}
```

```
def qt_weyl_vmul(X,K,Y,L) {
  extern WeylV, WeylDV;
  if ( member(X,WeylV) >= 0 || member(Y,WeylDV) >= 0 ) return Y^L*X^K;
  if ( WeylV[I=member(X,WeylDV)] != Y ) return Y^L*X^K;
  else {
    K = eval_quote(K); L = eval_quote(L); M = K>L?L:K;
    for ( T = 1, I = 0; I <= M; T = idiv(T*K*L,I+1), I++, L--, L-- )
      R += T*Y^L*X^K;
    return R;
  }
}

[256] WeylV=[`x,`y,`z]$
[257] WeylDV=[`dx,`dy,`dz]$
[258] qt_set_ord(map(eval_quote,append(WeylV,WeylDV)))$
[259] Rweyl=[[`X^K*Y^L,`qt_is_var(X)&&qt_is_var(Y)&&nqt_comp(Y,X)>0,
`qt_weyl_vmul(X,K,Y,L)]]$
[260] qt_rewrite(`((x*dy+y*dx)^2),Rweyl,1);
(((x)^(2))*((dy)^(2)))+((2)*(x)*(y)*(dx)*(dy))+((x)*(dx))
+(((y)^(2))*((dx)^(2)))+((y)*(dy))
```

Action にユーザ定義関数を用いることにより, *Weyl* 代数の書き換え規則を一つにまとめている.

6 まとめ

Risa/Asir における数式の中間的表現である FNODE をユーザ言語から操作するためのインタフェースを実装した. これにより, ユーザが定義する書き換え規則による数式の書き換えが可能となった. 書き換えの効率についてはほとんど考慮できていない. 特に, 標準形への変換と書き換えを並行して行うことが必要と考えており, 今後の課題の一つである. また, パターンマッチング自体もまだ完全なものとはいえず, 改良すべき点が多くある. 目的に応じた標準的な書き換え規則集合をデフォルトで提供することも必要である.

この書き換えと weight ベクトルによる単項式比較を組み合わせることにより, 自由結合代数における一般的な書き換え計算を論じた. ここで提案した一般化は Weyl 代数の同次化の理論を含む. Risa/Asir で新しく導入した, QUOTE に対する一般的な weight ベクトルのメカニズム `qt_set_weight` によりわれわれの理論とアルゴリズムのプロトタイプを容易に試すことが可能である. V. Levandovskyy [2] は G-algebra の概念を導入して, Singular に実装した. われわれのアプローチを発展させ, 同次化をとおして, well order でない場合にも適用できるグレブナー基底の理論を構成すれば, G-algebra より広い範囲の algebra を扱うことが可能となる. 一般的な枠組みの応用として, 将来的には D-加群のアルゴリズムを拡張し, Calderon-Moreno 等の導入した algebra を局所的に扱うなどの応用が見込まれる. また, FNODE をユーザ言語より操作する関数を用いることにより, 入力, 出力のユーザインタフェースを大幅に改善できることにも注意しておきたい.

参考文献

[1] S. Wolfram, The MATHEMATICA Book, Fourth Edition. Cambridge University Press (1999).

[2] V. Levandovskyy, Non-commutative Computer Algebra for Polynomial Algebras: Gröbner Bases, Applications and Implementation. Dissertation, Universität Kaiserslautern (2005).