Journal of Mathematical Chemistry 11(1992)133–144

COMBINATORIAL AND NONCOMMUTATIVE LIST-PROCESSING AND ASSOCIATED GRAPHS

M. DJURASOVIĆ and Z. STIPČEVIĆ
Institut za Fiziku, Prirodno-matematički Fakultet, Vojvode Putnika 43, 71000 Sarajevo, Yugoslavia

Abstract

Recently the authors have proposed a list-processing approach to the modeling of algebraic quantum field theory methods in quantum mechanics in which the noncommutative algebra of quantum-mechanical operators is emulated by lists. The processing produces reordered sequences of elements of a ring with a unit commutator and generates dynamic structures which, for some initial arrangements, correspond to partially ordered graphs characterized by recurrence relations and combinatorial identities. Likewise, in another list-processing application to physical problems, a simulation of Feynman diagrams hinged on predominantly combinatorial aspects and demanded explicit generation of certain combinatorial objects. This motivated an investigation into the combinatorial nature of noncommutative list-processing and of recursive algorithms for explicit construction of combinatorial lists, which we now present. The emphasis is also placed on the consideration of associated graphs and the graph-theoretic origin of the appearance of recurrence relations in the reordering theorems of the noncommutative algebra.

1. Introduction

The present contribution examines algorithmic essentials and possible programming realizations of the recently proposed list-processing approach to the modeling of algebraic quantum field theory methods in quantum mechanics [1] and reports on the ongoing investigation into the corresponding graph-theoretic and combinatorial aspects [2].

The aim of the current phase of research is the completion of a software unit capable of manipulating, within the noncommutative ring, reasonably complex algebraic structures which would undergo appropriate rearrangement procedures and thereby manifest combinatorial and algebraic regularity expressible in terms of graphs, recurrence relations, etc. The unit should be sufficiently fast, flexible and manageable to allow for easy emulation of the ring state transformations and the appropriate tracing of the graph origin of query results.

In view of foreseeable quantum-mechanical applications – in chemistry and in many-body quantum physics – we are particularly interested in normally ordered binomial expansions of the two ring generators (on account of the algebraic structure

of most Hamiltonians [3]), and in the processing of combinatorial structures within the ring (because of the physical requests for various partial symmetries imposed on the states created from vacuum).

Related to the dependence of the speed and efficiency of processing on whether one develops the appropriate procedures relying from the outset on the basic unit commutator or makes use of the commutator between different powers of the ring generators [4], we also present some algebraic properties of the corresponding expansion coefficients.

The full reproduction of the processing unit is beyond the scope of this report. Instead, we rather extensively discuss the basic algorithmic features of noncommutative list-processing and elaborate on the involved recursive aspects of logic-programming, restricting the presentation to only basic explicit procedures. The algorithms are readily implemented as a PROLOG query-base, but can also be translated into appropriate PASCAL procedures in a rather straightforward manner, which then may easily be converted from recursive into iterative actions to avoid excessive recursion nesting (and related stack limitations) and increase the processing speed.

2. Review of algebraic properties

Let R denote an associative ring with unit 1 of characteristic zero. For any two ring elements A and B which obey the commutation relation

$$[A, B] = 1 \tag{1}$$

and are referred to as generators, the sequence

$$1, BA, B^2A^2, B^3A^3, B^4A^4, \ldots, B^nA^n \tag{2}$$

is linearly independent over the ring of integers.

The immediate implications of the commutation relation (1) are

$$[A^n, B] = nA^{n-1} \tag{3}$$

and, likewise

$$[A, B^n] = nB^{n-1} \tag{4}$$

or some more involved relations, e.g.

$$[A,(BA)^n] = \sum_{q=0}^{n-1} \binom{n}{q} (BA)^q A. \tag{5}$$

In some cases with symmetric initial arrangements, the associated reordering prescriptions take the form of transformation expansions with integer-valued coefficients, subject to certain recurrence relations. The most useful are the following [5]:

$$(AB)^n = \sum_{q=0}^{n} C(n,q)(BA)^q \tag{6}$$

with the recurrence relation

$$C(n, q) = C(n-1, q-1) + C(n-1, q), \tag{7}$$

then

$$A^n B^n = \sum_{q=0}^{n} L(n,q) B^q A^q \tag{8}$$

with

$$L(n, q) = L(n-1, q-1) + (n+q)L(n-1, q), \tag{9}$$

and

$$B^n A^n = \sum_{q=0}^{n} S(n,q)(BA)^q \tag{10}$$

with

$$S(n, q) = S(n-1, q-1) - (n-1)S(n-1, q). \tag{11}$$

The normal-product expansion of the nth power of $B + A$ has the form

$$(B + A)^n = \sum_{p=0}^{n} \sum_{q=0}^{p} H(n,q) \binom{p}{q} B^q A^{p-q} \tag{12}$$

with coefficients $H(n, q)$, defined for n, any positive integer or zero, and for $q = 0, 1, 2, 3, \ldots, n$, so as to satisfy the recurrence relation

$$H(n, q) = H(n-1, q-1) + (q+1)H(n-1, q+1) \tag{13}$$

and the stopping rule $H(0, 0) = 1$.

Finally, the general powers' commutator has the normal-order expansion

$$[A^m, B^n] = \sum_{q=1}^{n} q! \binom{m}{q} \binom{n}{q} B^{n-q} A^{m-q}. \tag{14}$$

3. Algorithmic and programming aspects

In ref. [1], we have formulated an algorithmic prescription for the list-processing transition to normal ordering. We also included an elementary micro-PROLOG query base and presented a few elementary examples and their graph-theoretic representation. We now examine related topics more extensively, paying special attention to some programming details.

The basic data structure which undergoes processing within the model should be capable of representing an arbitrary linear combination with integral coefficients

of sequences, of arbitrary length and in arbitrary arrangements, of the ring generators A and B. The natural choice for a data structure with needed properties is a compound list whose records contain an integer field for coefficients and a field supporting a linked list of A and B symbols, hereafter referred to as a component list. The processing corresponds to a transition into another appropriate combination of the same type, with the aim of achieving, in keeping with the requirements imposed by the basic commutator (1), a desired arrangement of generators (i.e. normal ordering). The intended algebraic manipulations with creation and annihilation operators are then emulated via state transitions within a vector space of compound lists between an initial state and a final normally ordered state. (The terms initial and final should not connotate temporal associations between these states, but rather refer to required rearrangements within the ring elements.)

Graph-related features of the noncommutative algebra list-processing and the abundant appearance of recurrence relations both originate in the recursive features of data structures and in the recursive nature of processing algorithms. In keeping with this observation and to further promote logic programming where "recursive thinking" plays a key role in the process of inference making, we have carried out algorithmic considerations relying almost exclusively on recursion. These we now briefly present.

Taking a component list as input, one has to look for the first occurrence of an ordered pair (A, B) of neighboring elements and replace the original list with two lists, one with the pair in reverse order and the other with the pair omitted, and then repeat the action until there is no (A, B) pair in the resulting compound list.

To satisfy swapping within the pair (A, B), one may employ a predicate "swap" recursively described as

```
swap([a,b|Y],[b,a|Y]):- !.
swap([X|Y],[X|Z]):- swap(Y,Z).                    (15)
```

and, similarly, deleting a pair (A, B) is tested by a recursive predicate "delab",

```
delab([a,b|X],X):- !.
delab([X|Y],[X|Z]):- delab(Y,Z).                  (16)
```

Since swapping and deleting appear concurrently, it is convenient to have a single predicate "abba" for testing both

```
abba([a,b|X],[b,a|X],X):- !.
abba([X|Y],[X|Y1],[X|Y2]):- abba(Y,Y1,Y2).        (17)
```

Considering a predicate "normorder" as being satisfied when an initial compound list is converted into a final compound list with no (A, B) pairs, in order to collect the component lists one needs the usual concatenation recursive predicate "union"

```
union([],X,X).
union([X|X1],Y,[X|Z]):- union(X1,Y,Z).                    (18)
```

and then may write

```
normorder([X|Y],Z):-
    abba(X,X1,X2),
    !,
    normorder([X1],Y1),
    normorder([X2],Y2),
    union(Y1,Y2,XN),
    normorder(Y,YN),
    union(XN,YN,Z).
normorder(X,X).                                            (19)
```

Note the use of the cut predicate, which prevents backtracking each time the "abba" predicate is satisfied and thus eliminates the need for the stopping rule in the second clause of the normorder predicate. Otherwise one would have to introduce a predicate "noab" which testifies to the total absence of (A, B) pairs

```
noab([]).
noab([a,b|_]):- !,fail.
noab([_|X]):- noab(X),                                     (20)
```

and also to modify the second "normorder" close to

```
normorder(X,X):- noab(X).                                  (21)
```

The predicate "normorder" describes the relationship between an arbitrary initial compound list and its corresponding final, normally ordered compound list. Note that the resulting compound list generally contains many identical component lists which now should be grouped together, determining appropriate coefficients, which is done by application of the usual collecting procedures.

In order to trace the intermediate processing results and introduce appropriate collection of identical elements at each subsequent level it is useful, especially when dealing with the initial configurations which display some degree of symmetry, to express the basic algorithm so that it generates a binary tree, e.g.

```
normordertree(X,t(L,X,R)) :-
    swap(X,X1),
    normordertree(X1,L),
    delab(X,X2),
    normordertree(X2,R),
    !.
normordertree(X,t(nil,X,nil)).                             (22)
```

The described predicates provide a direct programming implementation as a PROLOG query base. If, for reasons stated previously, one prefers to deal with an imperative language – like PASCAL – capable of an easy modification of recursive procedures into iterative procedures, one has to appropriately reexpress (19) as a PASCAL function.

When component lists and compound lists are implemented as PASCAL dynamic structures, their type may be defined as

```
list = ^link;
     link = record
              linfo : char;
              lref : list
            end;                                                    (23)
```

and

```
llist = ^llink;
llink = ^record
          num : integer;
          llinfo : list;
          lref : llist
        end;                                                        (24)
```

respectively. Predicate (19) is then replaced by the following Pascal recursive function:

```
function normorder(L1:llist):llist;
begin
  if noab(L1) then normorder:= L1
  else
    if abtest(first(L1)) then
      normorder:= normorder(fput(swap(first(L1)),
                  fput(delab(first(L1)),bf(L1))))
    else
      normorder:= fput(first(L1), normorder(bf(L1)))
end;                                                                (25)
```

When the binary tree emulation is used, the appropriate PASCAL type may be defined as

```
tree = ^node;
node = record
         tinfo : list;
         left, right : tree
       end;                                                         (26)
```

and (22) may be replaced by a list-into-binary-tree PASCAL procedure:

```
procedure normordertree(L:list; var T : tree);
begin
  new(T);
  T^.list := L;
  if noab(L) then
    begin
      T^.left := nil;
      T^.right := nil
    end
  else
    begin
      normordertree(swap(L), T^.left);
      normordertree(delab(L), T^.right)
    end
end;                                                    (27)
```

The leaves of the resulting tree can then be interpreted as the processing output. Note that it is assumed that the usual list processing functions "first", "bf", "fput", etc., have been previously defined.

The purpose of the developed software is twofold: on the one side it should enable fast and reliable processing of long and complex initial lists, and on the other it should facilitate algebraic and graph-theoretic analysis.

4. Binary trees, networks and combinatorial aspects

When list rearrangements are carried out by repetitive reference to the unit commutator – until the normal order is achieved – the intermediate steps comprise a corresponding binary tree in which the initial list is placed at the root. Each successive left branch produces swapping within the first encountered pair (A, B), while each successive right branch produces appropriate deletion of the pair. The intermediate results are placed in the appropriate nodes on different levels of the tree. The final configuration – the resulting normally ordered list – coincides with the leaves of the tree. In ref. [1], we have presented a few examples which illustrate the rearrangements and display some graph-related aspects of the rearrangement process.

In some cases, judicious collection procedures, sometimes involving different intermediate levels, may transform the tree into networks and manifest various numeric recurrence relations among the appropriate expansion coefficients. In this way, for instance, one can rederive expressions (7)–(11). The associated graphs stem from the presence of equivalent nodes in the binary tree, which prompts for subsequent contractions and yields a corresponding transformation of the binary tree into a more general graph. This is often brought about by some partial ordering, implicit in the list-processing, which implies the existence of recurrence relations

interpreted as rules connecting respective nodes. In general, the appearance of recurrence relations can be traced back to the recursive features of data structures and to the recursive nature of processing algorithms.

Algebraic considerations within the noncommutative ring which led to various recurrence relations, e.g. (7), (9), (11), (13), etc., first presented in ref. [5], possess graphical counterparts and, consequently, can be used to study the related graph-theoretical aspects. Reversely, the related graphs may yield constructive assistance in the detection of some yet unknown algebraic relations. With respect to the algorithmic features of the noncommutative list processing, we observe that both algebraic and graph-theoretic considerations may beneficially influence the development of more efficient algorithms [4].

To be more specific, let us note that the normal ordering of an arbitrary ring element $A^{m_1}B^{n_1}A^{m_2}B^{n_2}A^{m_3}B^{n_3}\ldots$ may be viewed from three different elementary levels: (a) jumping of B over A, (b) jumping of B over A^m, and (c) jumping of B^n over A^m. Algebraically, these correspond to commutators (1), (3) and (14), respectively. Tracing the corresponding intermediate results then leads to binary trees, elementary networks and more complex graphical structures.

As an illustration, consider the binary tree of intermediate transitions for an initial list A^mB^n. Inspecting the repetitive features of this binary tree, one easily observes that leftmost nodes represent progressive motion of B generators from right to left, and that while each B travels from left to right, jumping over each successive A, the neighboring right nodes all contain the same content, namely, the list from which the (A, B) pair under consideration had been removed. Deletion of an (A, B) pair does not change the type of the list structure, it only decrements by one the powers of A and B, and consequently we are faced with a recursive situation which amounts to saying that the processing tree shrinks into a network with multiplicative side weights which, along the slash diagonals, progressing from left to right, amount to $m, m-1, m-2, \ldots$, respectively [1].

If the position of the network nodes is represented by two coordinates: $n = 0, 1, 2, 3, \ldots$, for rows, and $q = 0, 1, 2, 3, \ldots$, for slash diagonals, and the numeric content of the nodes is expressed as a function $M(m, n, q)$, the network implies the recurrence relation

$$M(m, n, q) = (m - q + 1)\, M(m, n-1, q-1) + M(m, n-1, q) \tag{28}$$

with the stopping rule

$$M(m, n, q) = 0, \quad \text{for } q < 0 \text{ and } q > \min(m, n); \quad M(n, 0, 0) = 1. \tag{29}$$

The values of $M(m, n, q)$ serve as expansion coefficients in the A^mB^n normal-ordering, i.e.

$$A^mB^n = \sum_{q=0}^{\min(m,n)} M(m,n,q)B^{n-q}A^{m-q}. \tag{30}$$

Recurrence relation (28) reflects the network multiplicative weight structure in which each node (n, q) receives information from two preceding nodes, $(n-1, q-1)$ and $(n-1, q)$, with factors $(m-q+1)$ and 1, respectively.

The A^mB^n normal-ordering network exhibits the property that all trajectories which lead from the root to any given node (n, q) carry equal contributions amounting to the product of branch factors, namely, $m(m-1)(m-2)\ldots(m-q+1)$. Note that any trajectory, in order to connect the root with the (n, q) node, has to descend a total of n steps, taking left or right branches at will, except for the overall condition that q right-branch selections have to be taken. This is equivalent to saying that the number of trajectories equals the number of combinations $C(n, q)$. These properties allow explicit determination of the coefficients $M(m, n, q)$; namely, noting that the value of $M(m, n, q)$ equals the total contribution from all trajectories passing through the (n, q) node, we have

$$M(m, n, q) = m(m-1)(m-2)\ldots(m-q+1)C(n, q) \tag{31}$$

or, equivalently

$$M(m,n,q) = q!\binom{m}{q}\binom{n}{q}. \tag{32}$$

Solution (31) can be verified by insertion into recurrence relation (28) and subsequent application of elementary combinatorial identities:

$$\binom{n}{q} = \frac{n}{q}\binom{n-1}{q-1} = \frac{n-q+1}{q}\binom{n}{q-1};\quad \binom{n}{q} = \binom{n-1}{q-1} + \binom{n-1}{q}. \tag{33}$$

The algebraic origin of recurrence relation (28) can be established in the usual way by the method of total induction. To that end, one has to start with equality

$$A^mB^n = (A^mB^{n-1})B, \tag{34}$$

make use of (30), apply commutation relation (3), and equate coefficients with equal powers on both sides.

Starting with the equality

$$A^mB^n = A(A^{m-1}B^{n-1})B, \tag{35}$$

however, one arrives at a different recurrence relation which connects each node with the three preceding nodes:

$$M(n,m,q) = M(n-1, m-1, q) + (n+m-2q+1)M(n-1, m-1, q-1)$$
$$+ (n-q+1)(m-q+1)M(n-1, m-1, q-2). \tag{36}$$

Alternatively, writing $A^m B^n$ as $(A^m B)B^{n-1}$ and using (3) yields

$$A^m B^n = A^{m-1} B^{n-1}(n + BA), \tag{37}$$

where subsequent insertion of (30), into both sides of (37), leads to a new recurrence relation

$$M(m,n,q) = (m + n - q)M(m-1, n-1, q-1) + M(m-1, n-1, q), \tag{38}$$

which, unlike (28), keeps neither m nor n fixed. We observe that for the special case $m = n$, and the notation $L(n, q) = M(n, n, n + q)$, relation (38) turns into the recurrence relation (9) for Laguerre coefficients.

Expansion (30) provides the basis for a more efficient version of the normal-order list-processing. Instead of looking for the first occurrence of a pair (A, B) in a general element, say, *AAABBABABBAABABBA*, i.e. *AA(AB)BABABBAABABBA*, it focuses on the beginning sequence *AAABB*, views the elements as *(AAABB)ABABBAABABBA*, uses (30), and recursively repeats transformations of compound lists until the normal order is reached.

As stated previously, the normally ordered expansion of arbitrary power of $A + B$ is, from a quantum-mechanical point of view, potentially most interesting [3]. In order to prepare the compound list, which corresponds to the unordered expansion of $(A + B)^n$, to serve as the initial state for the PASCAL function "normorder" (25), we have added the following "apbton" function which recursively produces the initial binomial expansion:

```
function apbton(n:integer):llist;
begin
  if n = 0 then apbton := fput(1,nil,nil)
else
  apbton := union(attach('A',apbton(n - 1)),
                  attach('B',apbton(n - 1)))
end;
```
(39)

The auxiliary recursive function "attach" is self-explanatory:

```
function attach(a:char;L1:llist):llist;
begin
  if L1 = nil then attach := nil
  else
    attach := fput(a, attach(a, bf(L1))
  end;
```
(40)

As printouts for small values of n indicate, the normally ordered binomial expansion can be expressed in terms of linear combinations of normal products of lower powers of $A + B$, which corroborates the general structure of (14). It also indicates the validity of the equality

$$[A + B]^n(A + B) = [A + B]^{n+1} + p[A + B]^{n-1}, \tag{41}$$

in which $[A + B]^n$ designates the normal product, and $(A + B)^n$ the normally ordered product, and can be easily proved by binomial expansion and elementary combinatorial identities. In its turn (41), by the usual method of total induction, applied to (12), leads to recurrence relation (13) and the corresponding network which yields the explicit expression

$$H(n, n - 2q) = \binom{n}{n - 2q} (2q - 1)!! \tag{42}$$

and the value zero for $H(n, n - 2q + 1)$.

To produce a combinatorially symmetric initial state $S(n, q)$, i.e. the linear combination of $C(n, q)$ ring elements such that each contains q generators of the type B, combinatorially distributed among generators of the type A, we need to introduce a PASCAL function "combab" [6]:

```
function combab(m,n:integer):llist;
begin
  if (m<0) or (n<0) or (n<m) then combab := nil
  else
    if (m=0) and (n=0) then combab := fput(1,nil,nil)
    else
      combab := union(attach('A',combab(m,n - 1)),
                      attach('B',combab(m - 1,n - 1)))
end;
```
(43)

Experimenting with various states produced by (43), as inputs into the "normorder" function (25), we obtain printouts with expressions indicating the following normally ordered expansion:

$$S(n,q) = \sum_{k=0}^{q} \frac{n!}{2^k k!(q - k)!(n + k - q)!} B^{q-k} A^{n-q-k}. \tag{44}$$

To summarize, we reiterate that the noncommutative algebra software enables fast and reliable processing of long and complex lists, facilitates algebraic and graph-theoretic analysis of different algebraic situations and possibly aids in the detection of some yet unknown relations.

Acknowledgement

One of the authors (Z.S.) would like to acknowledge the interest and encouragement of Professor Gordon E. Baird.

References

[1] M. Djurasović, S. Grubačić and Z. Stipčević, J. Math. Chem. 8(1991)137.
[2] M. Djurasović and Z. Stipčević, A list-processing simulation of no-self-loop Feynman diagrams, in: *Proc. 13th Information Technologies Conf.*, Jahorina, 1989.
[3] M.M. Ninan and Z. Stipčević, Amer. J. Phys. 37(1969)734.
[4] M. Djurasović and Z. Stipčević, A short-cut approach to QM list-processing, in: *Proc. 15th Information Technologies Conf.*, Jahorina, 1991.
[5] M.M. Ninan and Z. Stipčević, Gl. mat. 4 24(1969)9.
[6] M. Djurasović and Z. Stipčević, Combinatorics via recursive list processing, in: *Proc. 14th Information Technologies Conf.*, Jahorina, 1990).