Help For:

'BIGEBRA 6' - A Maple Package for Clifford and Grassmann Hopf Gebras

Version 1.01 -- designed for Maple 6

(Copyright (c) Rafal Ablamowicz * and Bertfried Fauser§, 1999 - 2002. All rights reserved.)

(*) Department of Mathematics, Box 5054
Tennessee Technological University
Cookeville, TN 38501 USA
Phone: (931) 372-3569, Fax: (931) 372-6353
E-mail: rablamowicz@tntech.edu
URL: http://math.tntech.edu/rafal/ (home of this package)

(§) Fachbereich Physik, Universit¨at Konstanz
Fach M678, 78457 Konstanz, Germany
Phone: +49 7531 883786, FAX: +49 7531 884266
E-mail: Bertfried.Fauser@uni-konstanz.de
URL: http://clifford.physik.uni-konstanz.de/~fauser/

Last revised: November 1, 2002 (BF. & RA)

Calling Sequence:
function(args)                      (if the package was loaded using with(Bigebra); )
Bigebra:-function(args)        (long form without loading the package)

Description:

• The BIGEBRA package supplememts the CLIFFORD package Clifford version 6 for Maple 6. If BIGEBRA is loaded using
   with(Bigebra); it loads automatically the Cliff5 package. BIGEBRA patches the Maple define/skeleton and define/multilinear
   routines of Maples define facility to allow a correct implementenation of the tensor product.

• The main purpose of the BIGEBRA package is to allow computations in tensor products of Clifford and Graßmann algebras. For
   this purpose, a tensor product `&t` is defined which is linear with respect to all non-Clifford elements (constants). This allows to
   perform calculations in Graßmann/Clifford modules and Graßmann/Clifford bundles. Bi- and Hopf algebraic structures as
   co-units, co-products, switches etc. are employed. All structures of Graßmann Hopf algebra and Clifford biconvolution are
   implemented. However, using this device, Graßmann-Cayley algebras and bracket or Peano algebras are also supported.
   Especially the meet (of point fields and of plane fiels in Plücker coordinatization) is implemented here in a very effective way.
   The join (of point fields) is implemented by the wedge of the CLIFFORD package.

• There are several functions which allow the usage of linear operators given in a matrix representation w.r.t. the Graßmann basis.
   Such operators can act on a single tensor slot, i.e. they are from End \Lambda/V, or on two adjacent tensor slots, i.e. they are from End
   (\Lambda/V &t \Lambda/V), where \Lambda/V is the space underlying the Graßmann algebra.

• The BIGEBRA package provides a facility to solve tangle equations [6] for linear operators applied to internal lines of the tangle
   if the tangle equation has n ingoing and one outgoing line (n -> 1 mapping). This simplifies e.g. the search for Clifford antipodes.

• The Clifford product can be defined in terms of Hopf algebras [8]. BIGEBRA uses the Clifford product of CLIFFORD cmul
   which internally uses by default the cmulRS subroutine based on teh Rota-Stein cliffordization tehnicne and Hopf algebraic
   methods. The Clifford co-product is derived from co-cliffordization inthe same way.

• The Clifford co-product needs an additional bilinear form, called co-scalarproduct, which has to be defined as the global dim_V x
   dim_V matrix BI. The dimension has to be specified using the global variable dim_V of CLIFFORD. The Clifford co-product
   needs an initialization which is done by calling once the function make_BL_Id . Some caution is needed here, since dim_V is set
to the maximal value 9 by CLIFFORD and the initialization may take very long in this case, so that dim_V should be set to a
smaller value if possible.

• The BIGEBRA package makes use of some global variables, which are stored in the table _CLIENV. Currently in use are:
   - _CLIENV[_SILENT], default = unassigned. If `true` it suppresses lots of startup output.
- _CLIENV[\_fakenow], a flag used to detect if BIGEBRA was already loaded. Needed for patching define.
- _CLIENV[QDEF_PREFACTOR], default = -1. Puts q-deformation into the Grassmann coproduct, (beware: ONLY there for now, the q-business is not yet officially supported and not well tested).

- BIGEBRA can also serve to provide the user a possibility to define various multilinear functions, i.e. tensor products over arbitrary rings, see define.
- The help pages of BIGEBRA are part of the same Maple database file (maple.hdb) which contains help pages for 'CLIFFORD' and should be located in a directory in Maple's `libname[1]' variable. BIGEBRA is supposed to merge with CLIFFORD in a forthcoming version for Maple ver. 6/7.
- BIGEBRA was already successfully used in deriving mathematically and physically relevant results [1,2,3]. Some references are added to provide information about Clifford Hopf algebras.

Literature:

Load Bigebra in the following way, Clifford (CLIFFORD ver. 6) has to be loaded manually!

You can increase the verbosity level of Bigebra setting infolevel[Bigebra]=3 or higher.

> restart:with(Clifford):infolevel(Bigebra)=3:with(Bigebra):
[Increase verbosity by infolevel`function `]=val -- use online help > ?Bigebra[help]

To initialize the Clifford coproduct type:

> dim_V:=2;
  BI:=linalg[\text{matrix}](\text{dim}_V,\text{dim}_V,\{a,b,c,d\});
  \text{make\_BI\_Id;}();

\text{BI} := \begin{bmatrix} a & b \\ c & d \end{bmatrix}

\text{Cliplus\ has\ been\ loaded.\ Definitions\ for\ type\/climon\ and\ type\/clipolynom\ now\ include\ &C\ and\ &C[K].\ Type\ ?cliproduct\ for\ help}

\text{(Id \&t Id) + a (e1 \&t e1) + c (e2 \&t e2) + (c b - d a) (e1 e2 \&t e1 e2)}

\text{BI\ is\ the\ dim}_V \times \text{dim}_V\ \text{matrix\ of\ the\ co-scalarprouct\ on\ co-one-vectors,\ from\ which\ the\ Clifford\ co-product\ \`&cco\ is\ derived\ by\ Rota-Stein\ co-cliffordization,\ [2,7,8].\ The\ \text{tensor\ product}\ \`&t\ is\ already\ defined\ and\ ready\ to\ use:}

> &t(a\&e1+\sin(\theta)*e3,\&e1); \quad \#\ \text{associativity, i.e. drop \`parentheses'}

\text{&t(e1, e2, e3, e4)}

> &t(a\&e1+\sin(\theta)*e3, b*e3-1/x*e1); \quad \#\ \text{multilinearity}

a b (e1 \&t e3) - \frac{a (e1 \&t e1)}{x} + \sin(\theta) b (e3 \&t e3) - \frac{\sin(\theta) (e3 \&t e1)}{x}

Alphabetic listing of available procedures in 'BIGEBRA':

- \&cco \quad -- Clifford co-product on
• \&gco -- The Graßmann co-product w.r.t. the wedge product.
• \&gco_d -- dotted Graßmann co-product acting on the undotted wedge basis.
• \&gpl_co -- Graßmann-Plücker co-product acting on hyperplanes in Plücker coordinatization.
• \&map -- \&map maps a product, i.e. a Clifford valued function of two Clifford polynomials (a 2->1 mapping) onto two adjacent slots of a tensor.
• \&t -- The tensor product defined in BIGEBRA during loading of the package.
• \&v -- Defined the vee-product, i.e. the meet.

• tensor polynomials.
• bracket -- Defines a bracket in the sense of a Peano space [8].
• cco_monom -- internal use only.
• contract -- Contract maps a cliscalar valued function of two Clifford polynomials onto two adjacent tensor slots.
• define -- Maple 6 'define' still has bugs, so 'define' had to be replaced by a patched code. New option: give a domain for k-multilinearity.
• drop_t -- Drops the tensor sign \&t in expressions like \&t(e1), projects on the first argument in \&t(p1,p2,...).
• eps -- no longer supported.
• EV -- EV is the evaluation of a multi-co-vector on a multivector. Multi-co-vectors are described currently (we are sorry to say) by the same Graßmann basis elements. The user is responsible to take care in which tensor slot co-vectors reside.
• gantipode -- Applies the Graßmann antipode to a tensor slot.
• gco_unit -- The Graßmann Hopf algebra co-unit.
• gswitch -- Graded switch of two adjacent slots of a tensor.
• help -- This page.
• linop -- Linop defines a linear operator acting on the Graßmann algebra, having a 2^dim_V x 2^dim_V co-contra-variant matrix representing it.
• linop2 -- Linop2 defines a linear operator acting on a tensor product of rank two of the Graßmann algebra, having a 4^dim_V x 4^dim_V co-contra-variant matrix representing it.
• list2mat -- List2mat computes from two lists of elements from V^ which are connectet as source and target of an linear operator a (possibly unfaithful reducible) matrix representation.
• list2mat2 -- List2mat2 computes from two lists of elements from V^ \&t V^ which are connectet as source and target of an linear operator a (possibly unfaithful reducible) matrix representation.
• make_BI_Id -- Initialization routine for the Clifford co-product.
• \&map -- \&map maps a product, i.e. a Clifford valued function of two Clifford polynomials (a 2->1 mapping) onto two adjacent slots of a tensor.
• mapop -- Mapop applies a linear operator (element of End V) defined by linop onto one single slot of a tensor.
• mapop2 tensor -- Mapop2 applies a linear tensor-operator (element of End V \&t V) defined by linop2 onto two slots of a tensor.
• meet -- The meet is equivalent to the \&v-(vee)-product.
• op2mat -- Op2mat returns a (possibly unfaithful reducible) matrix representation in V^ of a linear operator given as argument.
• op2mat2 -- Op2mat2 returns a (possibly unfaithful reducible) matrix representation in V^ \&t V^ of a linear operator given as argument.
• pairing -- A pairing of two Clifford polynomials.
• **peek** -- Peek gets a **Clifford polynom** from a **tensor** at a certain position.

• **poke** -- Poke puts a **Clifford polynom** into a **tensor** at a certain position.

• **remove_eq** -- Helper function, which allows to remove trivial equations if tangle equations are solved manually.

• **switch** -- Switch two adjacent slots of a **tensor** (Just a swap).

• **tcollect** -- Tcollect collects **clisclar** coefficients in a **tensor** expression.

• **tsolve1** -- Tsolve1 solves tangle equations with n ingoing and one outgoing line (n--> 1 mappings). It has the ability to solve for operators applied to internal lines of the tangle. Such operators can be defined algebraically or using **linop** and **linop2**.

• **VERSION** -- Displays informations about the current version of BIGEBRA.

---

New Types in 'BIGEBRA':

• **type/tensobasmonom** - A tensor basis monom having no prefactor.

• **type/tensormonom** - A tensor monom which may have a prefactor of type **clisclar**.

• **type/tensorpolynom** - A sum of tensor monoms.

---

See Also: Clifford:-setup, Clifford:-version, Bigebra:-VERSION

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.

Last modified: November 1, 2002/BF/RA.
Function: Bigebra:-`&cco` - Clifford co-product

Calling Sequence:

\[ t1 = \&cco(p1,i) \]
\[ t1 = \&cco(c1) \]

Parameters:

- \( p1 \): a tensorpolynom (element of `type/tensorpolynom`) of rank not less than \( i \) in each factor
- \( i \): the slot number (first slot from the left is 1) on which the co-product acts
- \( c1 \): a Clifford polynom (element of one of these types: `type/clipolynom`, `type/climon`, `type/clibasmon`)

Output:

- \( t1 \): a tensor polynom

Global variables:

- \( BI_Id \): set by `make_BI_Id`
- \( dim_V \): the dimension of the one-vector space \( V \)

WARNING:
The Clifford co-product takes only one 'factor' (and one parameter), the **infix form** makes no sense with this function and yields unpredictable nonsense.

Description:

- Like the Clifford product, Clifford co-product needs a bilinear form defined on the base space of the Grassmann algebra. In the case of the co-product, this form is tied to co-one-vectors, so it is called co-scalar product. Since we deal with finite dimensional spaces, the dimension of the covector space is \( dim_V \), the same as for the vector space \( V \) used by `CLIFFORD`. Hence we use the **global variable** \( dim_V \), which has to be assigned. The matrix of the Clifford co-product w.r.t. the co-one-vector basis (in abuse of language also denoted by \( e1 \), see remarks in \( EV \)) is stored in \( BI \). The elements of \( BI \) can be assigned freely, without any restrictions or relations to the matrix of the Grassmann scalar product \( B \). \( BI \) can be singular or nonsymmetric or even zero, in which case the Clifford co-product reduces to the Grassmann co-product.
- The Clifford co-product is based on Rota-Stein co-cliffordization \([2,3,7,8]\). This is the categorial dual of the Rota-Stein cliffordization of the Grassmann algebra which leads to the Clifford co-product. In Sweedler notation, the formula for Clifford co-product is:

\[
\Delta_{\&c}(x) = (\wedge &t \wedge)(Id &t BI_(1) &t BI_(2) &t Id)\Delta(x)
\]

where \( \Delta_{\&c} \) is the Clifford co-product, \( \Delta \) is the Grassmann co-product. The factor \( BI_Id = BI_(1) &t BI_(2) \) (in fact internally represented by a list of type \{ [sign,BI_(1),BI_(2)], ... \}) has to be precomputed using `make_BI_Id`.
- The Clifford co-product is associative by construction, but it is not (graded) co-commutative.
- Clifford co-products lead to non-connected co-modules, see Milnore and Moore. This is an important difference with Grassmann co-products \([3,6]\).
- There is a sort of asymmetry in the BIGEBRA package, since the Clifford product is not (yet) computed using Rota-Stein cliffordization. The Clifford product is simply taken from CLIFFORD. This may cause problems if one tries to compute with \( q \)-deformed Clifford co-products. However, \( q \)-deformed Grassmann co-products are available by setting the global variable `_CLIENV[\_QDEF_PREFACCTOR]` e.g. to `-q`.

TO DO:

- The Clifford product has to be based on Rota-Stein cliffordization and a \( q \)-wedge product has to be created.

Examples:

\[
> \text{restart:bench:=time():with(Clifford):with(Bigebra):}
\]
Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra

\[ \text{dim}_V := 2; \]
\[ \text{BI} := \text{linalg\[matrix\]}(\text{dim}_V, \text{dim}_V, [a, b, c, d]): \]
\[ \text{# co-scalar product} \]
\[ \text{ml} := \text{make\_BI\_Id}(); \]
\[ \text{# remember this result in ml} \]

Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.

The Clifford co-product of Clifford polynomials needs no slot index:

\[ > \text{c1} := \&cco(e1); \]
\[ \text{c2} := \&cco(&t(e1), 1); \]
\[ \text{c3} := \&cco(&t(e1), 2); \]

Reduction to the Graßmann co-product is obtained by letting all parameters of the co-scalar product go to zero:

\[ > \text{subs}(a = 0, b = 0, c = 0, d = 0, \text{ml}); \]
\[ \text{## &cco(Id) -> &gco(Id)} \]
\[ \text{&gco(Id)}; \]
\[ \text{subs}(a = 0, b = 0, c = 0, d = 0, c2); \]
\[ \text{## &cco(e1) -> &gco(e1)} \]
\[ \text{&gco(e1)}; \]

Reduction to the Graßmann co-product as a test:

\[ > \text{subs}(a = 0, b = 0, c = 0, d = 0, \text{ml}); \]
\[ \text{## &cco(Id) -> &gco(Id)} \]
\[ \text{&gco(Id)}; \]
\[ \text{subs}(a = 0, b = 0, c = 0, d = 0, c2); \]
\[ \text{## &cco(e1) -> &gco(e1)} \]
\[ \text{&gco(e1)}; \]
Note however that acting on different slots of the same tensor gives different answers:

```
> res1 := &cco(&t(e1, e2), 1);

res1 :=

> res2 := &cco(&t(e1, e2), 2);

res2 :=

```

If the index is not in the range of the tensor slots, an error occurs so the user has to account for that.

```
> &cco(&t(e1, e2), 3); ####<<<<-- Intended error

Error, (in Bigebra: &cco) invalid subscript selector

> printf("Worksheet took %f seconds to compute on a 2x 1GHz PIII 1MB RAM machine\n", time() - bench);

Worksheet took 27.030000 seconds to compute on a 2x 1GHz PIII 1MB RAM machine

> simplify(tcollect(%-%));

0
```

See Also:  
Bigebra:-`&gco`, Bigebra:-`&t`, Bigebra:-drop_t
Function: Bigebra:-`&gco_d` - dotted Graßmann co-product for a different filtration

Calling Sequence:

\[ t2 := \&gco_d(t1,i) \]
\[ t2 := \&gco_d(c1) \]

Parameters:

- \( t1 \) : tensor polynoms
- \( i \) : tensor slot to be acted on
- \( c1 \) : Clifford polynom

Output:

- \( t1 \) : a tensor polynom

Global:

- for the transition to the regular wedge basis and back to the dotted basis (both represented as eawebw...) the two global matrices \( F \) and FT are used.

Description:

- The dotted Graßmann co-product is isomorphic to the regular Graßmann co-product on the dotted wedge basis. The function \&gco_d\((t1,i)\) computes this product using the original Graßmann co-product but w.r.t. the undotted basis. It is hence the counterpart to the function Cliplus:-dwedge which computes the dotted wedge product in the undotted basis. The dotted and undotted bases arise from different filtrations of the underlying Graßmann algebra. As Graßmann algebras they are isomorphic, but they are not isomorphic as Hopf algebras!
  - The function \&gco_d\ needs the Cliplus and it will load it automatically if it was not done previously.
- This functionality is simply gained by wrapping the original Graßmann co-product and using internally the two functions \`convert/wedge_to_dwedge` and \`convert/dwedge_to_wedge` from the Cliplus package.
- If \( F \) and FT are antisymmetric arrays which are mutually transposed to each other (negative of each other) this mapping is an isomorphism (\( F \) and FT need not be non-singular!). Hence the dotted Graßmann co-product can be computed in the undotted basis by transforming back and forth.
- From a physical point of view, it is possible to consider the undotted basis as a fermionic time-ordered product and the dotted basis as a fermionic normal-ordered product. Hence, the dotted Graßmann co-product employs the co-product of normal-ordered fields in the time-ordered basis, see [1].
- It should be noted that the loop tangle ‘product \& coproduct’ works out completely differently if one mixes products and co-products for the same algebra with different filtration (ordering).
- References:

Examples:

```plaintext
> restart: bench:=time(): with(Clifford): with(Bigebra):

Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]

Define the dimension of the vectorspace \( V \) under consideration to be 3, and define \( F \) and FT

> dim_V:=3:
  F:=array(antisymmetric,1..dim_V,1..dim_V):
  F=evalm(F);
  FT:=evalm(-1*F);
  w_bas:=cbasis(dim_V); ## the wedge basis
```
Now we invoke for the first time the dotted Grassmann co-product which needs also load the Cli5plus package:

```maple
> with(Cliplus):
&gco_d(e1we2);
```

Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.

Now let us show how the co-product acts on dotted and undotted elements

```maple
> w_p1:=e1we2;             #selection of an element in undotted basis
w_c1:=&gco_d(w_p1);      #action of &gco_d on undotted element e1we2
```

```maple
d_p1:=dwedge[F](e1,e2);  #transformation of e1we2 to dotted basis
d_p2:=convert(w_p1,wedge_to_dwedge,F); #another way to accomplish the same transformation
d_c1:=&gco_d(d_p1);      #action of &gco_d on the image of e1we2 in dotted basis
```

The following examples compose the dotted co-product with dotted and undotted wedge (acting on a wedge basis!!). First, let's show a dotted basis:

```maple
> Grassmann_basis:=cbasis(3);                                    #Grassmann un-dotted basis
dotted_basis:=map(convert,Grassmann_basis,wedge_to_dwedge,F);  #dotted basis
```

We will use the following notation for the dotted basis, e.g., e1We2 = e1we2+F[1,2]*Id, etc.:

```maple
> S:=[e1we2+F[1,2]*Id=e1We2,e1we3+F[1,3]*Id=e1We3,e2we3+F[2,3]*Id=e2We3,
e1we2e3+F[2,3]*e1-F[1,3]*e2+F[1,2]*e3=e1We2e3];
```

```maple
subs(S,dotted_basis); #dotted basis in shorter (dotted wedge) notation
```

Then, we compose dotted co-product with undotted and dotted wedge:

```maple
> for i in dotted_basis do
  d_p1:=&gco_d(i):
  drop_t(&map(d_p1,1,dwedge[F]));
  `action_dwedge_o_&gco_d` = 2^maxgrade(%)*subs(S,%/2^maxgrade(%));
  d_p2:=convert(%,dwedge_to_wedge,-F);
```
Thus, the above shows that \((\text{wedge o Grassmann co-product})(x) = 2^{\text{grade of } x} \times x\) for any \(x\) in the Grassmann basis (Grassmann_basis) shown above and that the same is true, namely, \((\text{dotted wedge o dotted Grassmann} \ldots\).
\text{co-product}(y) = 2^{\text{grade of } y} \cdot y \text{ for any } y \text{ in the dotted wedge basis (dotted\_basis) shown above.}

\begin{verbatim}
> printf("Worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine\n",time()-bench);
Worksheet took 36.784000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra::&gco', Bigebra::&cco', Bigebra::&t', Bigebra::drop_t, Bigebra::&map', Cliplus::dwedge
\end{verbatim}

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
**Function:** Bigebra::\&gco\` - Graßmann co-product

**Calling Sequence:**

\[ t1 := \&gco(t2,i) \]
\[ t1 := \&gco(c1) \]

**Parameters:**

- \( t2 \) : a tensorpolynom (an element of `type/tensorpolynom`) of rank not less than \( i \) in each factor
- \( i \) : the slot number (first slot is from the left is 1) on which the co-product acts

- \( c1 \) is a Clifford polynom (an element of one of these types: `type/clibasmon`, `type/climon`, `type/clipolynom`)

**Output:**

\( t1 \) : is a tensorpolynom.

**WARNING:**

The Clifford co-product takes only one 'factor' (and one parameter), the **infix form** makes no sense with this function and yields **unpredictable nonsense**.

**Description:**

- A Graßmann algebra leads naturally to a multi-vector space \( \Lambda V \). This space has a dual which we call \( (\Lambda V)^* = \Lambda V^* \). There is a natural pairing between one-vectors and co-one-vectors which can be extended to a graded scalar valued (target/domain is the ring \( k \)) action of co-multi-vectors on multivectors called pairing: \[ < A, B^* : V(V)^* \otimes \Lambda V \rightarrow k. \] Let us denote the Graßmann product of co-multi-vectors by \( \Lambda V^* \), i.e using a &v (vee)-product. One obtains by categorical duality (i.e. by reversing arrows in commutative diagrams) the coproduct \( \Delta \). For two-vectors this reads:

\[
< a_1 \vee a_2 | b > = < a_1 \& t a_2 | \Delta(b) >
= < a_1 \& t a_2 | \sum_i (b)_{(1i)} \& t (b)_{(2i)} >
= \sum_i < a_1 | (b)_{(1i)} > < a_2 | (b)_{(2i)} >
\]

Since the co-vectors \( a_1 \) and \( a_2 \) are arbitrary co-multi-vectors, this defines the coproduct on an arbitrary multi-vector Grassmann element \( b \) in \( \Lambda V \). If \( a_1 \) is a co-one-multivector, this turns out to be the Laplace row expansion of the pairing, see [8,3]. The same consideration can be done for columns, i.e. moving the wedge \( \Lambda \) from right to left in the pairing to generate a Grassmann co-product on co-multi-vectors. Since we denote currently vectors and co-vectors by the same vector symbol `e`, the user has to take care of the fact in which slot of a tensor a vector or co-vector resides, see [9,3].

- Expanded in our basis, the above formalism leads to a combinatorial formula using split-sums and shuffles which are internally computed in BIGEBRA, [41]. From this construction, one concludes that the Graßmann co-product enjoys the following properties:

  - The Graßmann co-product is associative, \((\Delta \& t Id) \Delta = (Id \& t \Delta) \Delta\).
  - The Graßmann co-product is graded co-commutative \( \Delta = \tau \Delta \), where \( \tau \) is the **graded switch**.
  - The Graßmann co-product is linear.
  - Together with the Graßmann wedge product one proves this structure to be a Graßmann Hopf gebra, which possesses an **antipode**.
  - The Graßmann Hopf gebra is an bi-augmented bi-connected Hopf algebra, which is also called a noninteracting Hopf gebra [3].

**Examples:**

```plaintext
> restart: bench:=time(): with(Clifford): with(Bigebra):

Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]
Some examples of Graßmann co-products of Clifford polynomials:
> &gco(e1);
```

```plaintext
> &gco(e1);
```
&gco(&t(e1),1); # the same, since
e1 = drop_t( &t(e1) );

(Id &t e1) + (e1 &t Id)

Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.

(Id &t e1) + (e1 &t Id)
e1 = e1

> &gco(e1we2);
&gco(a*e3);
&gco(e1we2+a*e3);

(Id &t e1we2) + (e1 &t e2) - (e2 &t e1) + (e1we2 &t Id) + (Id &t e1we2) + (e1 &t e2) - (e2 &t e1) + (e1we2 &t Id) + a (Id &t e3) + a (e3 &t Id)

Acting on tensor slots:

> &gco(&t(e1,e2),1);
&gco(&t(e1,e2),2);

&t(Id, e1, e2) + &t(e1, Id, e2)
&t(e1, Id, e2) + &t(e1, e2, Id)

> &gco(Id);
&gco(%,1);

Id &t Id
&t(Id, Id, Id)

> &gco(a*&t(e1,e2)+b*&t(e3,e4),1);

a &t(Id, Id, e2) + a &t(e1, Id, e2) + b &t(Id, e3, e4) + b &t(e3, Id, e4)

Checking co-associativity:

> &gco(&gco(&t(e1we2),1),1);
&gco(&gco(&t(e1we2),2),2);

evalb(%-%=0); 

&t(Id, Id, e1we2) + &t(Id, e1, e2) + &t(e1, Id, e2) - &t(Id, e2, e1) + &t(Id, e1we2, Id) + &t(e1, e2, Id) - &t(e2, e1, Id) + &t(e1we2, Id, Id)
&t(Id, Id, e1we2) + &t(Id, e1, e2) + &t(e1, Id, e2) - &t(Id, e2, e1) + &t(Id, e1we2, Id) + &t(e1, e2, Id)

&t(Id, e1we2, Id) + &t(e1we2, Id, Id)
true

Checking graded co-commutativity:

> g1:=&gco(e1we2+e1we2we3);
g2:=gswitch(g1,1);

true

Note however that acting on different slots of the same tensor gives different answers:

> res1:=&gco(&t(e1,e2),1);
res2:=&gco(&t(e1,e2),2);
printf("res1 - res2 =0 is %s ",evalb(tcollect(res1-res2)=0));

res1 := &t(Id, e1, e2) + &t(e1, Id, e2)
res2 := &t(e1, Id, e2) + &t(e1, e2, Id)

res1 - res2 =0 is false !

If the index is not in the range of the tensor slots, an error occurs, so the user has to account for that.

> &gco(&t(e1,e2),3);

Error, (in Bigebra:-&gco) invalid subscript selector

> printf("Worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine\n",time() - bench);

Worksheet took 3.996000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra:-`&cco`, Bigebra:-`&t`, Bigebra:-drop_t, Bigebra:-`&map`

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-`&gco_pl` - Grassmann-Plücker co-product for hyperplanes

Calling Sequence:

t1 := &gco_pl(t2,i)
t1 := &gco_pl(c1)

Parameters:

• t2 : a tensorpolynom (an element of `type/tensorpolynom`) of rank not less than i in each factor
  i : the slot number (first slot is from the left is 1) on which the co-product acts

• c1 is a Clifford polynom (an element of one of these types: `type/clibasmon`, `type/climon`, `type/clipolynom`)

Output:

• t1 : is a tensorpolynom.

WARNING:
The Grassmann-Plücker co-product takes only one 'factor' (and one parameter), the infix form makes no sense with this function and yields unpredictable nonsense.

Description:

• In analogy with the Grassmann co-product &gco which is dual to the wedge, the Grassmann-Plücker co-product is dual to the meet or vee product &v denoted \(/\). Using a natural pairing and the same construction as for the Grassmann co-product we obtain a co-product which acts on hyperplanes. These hyperplanes are parameterized in Plücker coordinates which is the origin of the name Grassmann-Plücker co-product.

• Note that due to the Plücker coordinatization we need \( \text{dim}_V \) to be set to the dimension of the generating space V.

• The Grassmann-Plücker co-product is expected to behave quite similarly to the Grassamnn product itself. However, due to the Plücker coordinatization a further duality is involved which yields some awkward signs and some unconventional outcomes.

Examples:

```maple
> restart: bench := time(): with(Clifford): with(Bigebra):

Increase verbosity by infolevel[\'function\']=val -- use online help > ?Bigebra[help]

[Some examples of Grassmann-Plücker co-products of Clifford monoms describing hyperplanes:]

> dim_V:=3:
  &gco_pl(e1);
  &gco_pl(&t(e1),1); # the same

Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.

\((e1 \& t e1we2we3) - (e1we2 \& t e1we3) + (e1we3 \& t e1we2) + (e1we2we3 \& t e1)\)

\((e1 \& t e1we2we3) - (e1we2 \& t e1we3) + (e1we3 \& t e1we2) + (e1we2we3 \& t e1)\)

[Note that the volume element behaves now as the unit under the meet (vee-product) and we obtain thus]

> &gco_pl(e1we2we3);

\(e1we2we3 \& t e1we2we3\)

while we find for hyperplanes (i.e. \( \text{dim}_V-1 \) vectors) the usual behaviour:

> &gco_pl(e1we2);
  &gco_pl(e1we3);
  &gco_pl(e2we3);

\((e1we2 \& t e1we2we3) + (e1we2we3 \& t e1we2)\)

\((e1we3 \& t e1we2we3) + (e1we2we3 \& t e1we3)\)

\((e2we3 \& t e1we2we3) + (e1we2we3 \& t e2we3)\)
Checking co-associativity:

```plaintext
> &gco_pl(&gco_pl(&t(e1\text{we}2),1),1);
&gco_pl(&gco_pl(&t(e1\text{we}2),1),2);
evalb(\%-%=0);

\[ \begin{align*}
&\text{t}(e1\text{we}2, e1\text{we}2\text{we}3, e1\text{we}2\text{we}3) + \text{t}(e1\text{we}2\text{we}3, e1\text{we}2, e1\text{we}2\text{we}3) + \text{t}(e1\text{we}2\text{we}3, e1\text{we}2\text{we}3, e1\we 2) \\
&\text{t}(e1\text{we}2, e1\text{we}2\text{we}3, e1\text{we}2\text{we}3) + \text{t}(e1\text{we}2\text{we}3, e1\text{we}2, e1\text{we}2\text{we}3) + \text{t}(e1\text{we}2\text{we}3, e1\text{we}2\text{we}3, e1\we 2)
\end{align*} \]
true
```

Checking ungraded co-commutativity:

```plaintext
> g1:=&gco_pl(e1\text{we}2+e1\text{we}2\text{we}3);
g2:=switch(g1,1);
evalb(\%-%=0);

\[ \begin{align*}
&g1 := (e1\text{we}2 & t e1\text{we}2\text{we}3) + (e1\text{we}2\text{we}3 & t e1\we 2) + (e1\text{we}2\text{we}3 & t e1\text{we}2\text{we}3) \\
g2 := (e1\text{we}2 & t e1\text{we}2\text{we}3) + (e1\text{we}2\text{we}3 & t e1\we 2) + (e1\text{we}2\text{we}3 & t e1\text{we}2\text{we}3)
\end{align*} \]
true
```

To show that the Graßmann-Plücker co-product is indeed the dual of the meet, we compute the loop tangle $\lor\circ \Delta$ which should evaluate to $2$ to the power of the grade of the input element. In dimension $3$ we find:

```plaintext
> dim_V:=3:bas:=cbasis(dim_V);
out1:=[seq(&gco_pl(bas[i]),i=1..nops(bas))]:
u:=proc(x) &map(x,1,`&v`) end:
map(drop_t@u,out1);

\[ \begin{align*}
&bas := [Id, e1, e2, e3, e1\text{we}2, e1\text{we}3, e2\text{we}3, e1\text{we}2\text{we}3] \\
&[8 Id, 4 e1, 4 e2, 4 e3, 2 e1\we 2, 2 e1\text{we}3, 2 e2\text{we}3, 8 e1\text{we}2\text{we}3]
\end{align*} \]
```

Note, that the grade is taken w.r.t. the Plücker coordinatization. In this setting, the volume element is the scalar and has grade zero, bi-vectors (i.e. dim$_V$-1 vectors) have grade one and pick up a factor two etc., while the identity is of Plücker grade $3$ (i.e. $\text{dim}_V - 0$) and gets a factor $2^3 = 8$. To show the same relation for the ordinary Graßmann product, we compute the same loop tangle with Graßmann co-product and wedge:

```plaintext
> bas:=cbasis(dim_V):
out1:=[seq(&gco(bas[i]),i=1..nops(bas))]:
u:=proc(x) &map(x,1,`&w`) end:
map(drop_t@u,out1);

\[ \begin{align*}
&bas := [Id, e1, e2, e3, e1\text{we}2, e1\text{we}3, e2\text{we}3, e1\text{we}2\text{we}3] \\
&[Id, 2 e1, 2 e2, 3 e3, 4 e1\we 2, 4 e1\text{we}3, 4 e2\text{we}3, 8 e1\text{we}2\text{we}3]
\end{align*} \]
```

We may note, that we find not two equivalent versions of the Graßmann-Plücker co-product as e.g. the meet and $\lor\text{v}$ (vee), but that there is a left and right version which differ in an overall sign depending on the parity of the dimension dim$_V$ and the parity of the monomial $x$ on which it is calculated.

We will now proceed to show that this is in full analogy to the convolution unit:

```plaintext
> dim_V:=2:bas:=cbasis(dim_V):
U:=proc(x) linop(x,U) end:
X:=add(_X[i]*bas[i],i=1..2^dim_V);
lhs_eq:=&map(tcollect(mapop(&gco_pl(X),2,U)),1,`&v`);

\[ \begin{align*}
&X := X_1 \text{Id} + X_2 e1 + X_3 e2 + X_4 e1\text{we}2 \\
lhs_eq := X_1 U_{4,4} & t(e2) - X_2 U_{3,4} & t(e2) + X_3 U_{4,1} & t(Id) + X_4 U_{1,1} & t(Id) - X_1 U_{3,1} & t(Id) + X_2 U_{2,4} & t(e1) \\
&+ X_3 U_{4,4} & t(e1) - X_1 U_{3,4} & t(e2) + X_1 U_{3,4} & t(0) + X_1 U_{4,4} & t(Id) + X_1 U_{3,4} & t(0) - X_1 U_{3,3} & t(0) \\
&+ X_1 U_{2,4} & t(e1) - X_2 U_{3,3} & t(Id) + X_1 U_{3,3} & t(e2) + X_1 U_{3,3} & t(0) + X_1 U_{3,4} & t(0) + X_1 U_{1,3} & t(0) \\
&+ X_1 U_{1,3} & t(e1) - X_2 U_{4,3} & t(e1\text{we}2) - X_3 U_{2,3} & t(e1\text{we}2) + X_3 U_{4,1} & t(e1\text{we}2) - X_3 U_{3,1} & t(e2) + X_3 U_{1,4} & t(0) \\
&+ X_1 U_{2,4} & t(0) + X_4 U_{3,4} & t(e1\text{we}2) + X_3 U_{2,4} & t(Id) - X_1 U_{1,2} & t(0) + X_2 U_{1,4} & t(0) + X_2 U_{2,4} & t(0)
\end{align*} \]
```

\[ \text{rhs_eq} := X; \quad \#\text{ i.e. the identity mapping} \]
\[ \text{rhs_eq} := X_1 \text{Id} + X_2 e1 + X_3 e2 + X_4 e1w2 \]
\[ \text{sol} := \text{tsolve1} (\text{tcollect} (\text{drop_t} (\text{lhs_eq} - \text{rhs_eq})), [\text{seq} (\text{seq} (U[i,j], i=1..2^\text{dim}_V), j=1..2^\text{dim}_V)], [\text{seq} (_X[i], i=1..2^\text{dim}_V)]) ; \]
\[ \text{sol} = U_{22} 0 = U_{33} 0 = U_{11} 0 = U_{44} 1 = U_{41} 0 = U_{14} 0 = U_{43} 0 = U_{23} 0 = U_{42} 0 = U_{32} 0 = U_{34} 0 = U_{24} 0, \ldots \]
\[ \text{matU} := \text{linalg[matrix]} (2^\text{dim}_V, 2^\text{dim}_V, (i,j) -> U[i,j]) ; \]
\[ \text{matU} := \text{subs} (\text{sol}[1], \text{evalm} (\text{matU})) ; \]

This outcome shows that we have a unit in this convolution algebra, however, the unit is the projection onto the highest grade element i.e. the lowest Plücker grade, i.e. the volume element.

A further step is to look for the new antipode of this algebra.
\[ \text{S} := \text{proc} (x) \text{ linop} (x, S) \end{proc} ; \]
\[ \text{lhs_eq} := \text{clicollect} (\text{drop_t} (\text{mapop} (\&gco_pl(X, 2, S)), 1, `&v`)) ; \]
\[ \text{lhs_eq} = (X_1 S_{2,1} + X_3 S_{2,4} - X_2 S_{2,2} + X_4 S_{4,4} + X_1 S_{1,3} - X_3 S_{2,3}) e1 \]
\[- (X_1 S_{2,4} - X_3 S_{2,3} + X_4 S_{4,4} - X_2 S_{2,2} - X_4 S_{4,2} + X_3 S_{1,3} - X_1 S_{1,1}) e2 \]
\[ \text{rhs_eq} := \text{subs} (\text{sol}[1], \text{linop} (X, U)) ; \]
\[ \text{rhs_eq} := X_4 e1w2 \]
\[ \text{sol} := \text{tsolve1} (\text{tcollect} (\text{drop_t} (\text{lhs_eq} - \text{rhs_eq})), [\text{seq} (\text{seq} (S[i,j], i=1..2^\text{dim}_V), j=1..2^\text{dim}_V)]) ; \]
\[ \text{sol} = [ S_{2,1} = 0, S_{2,2} = 1, S_{1,3} = 0, S_{1,1} = 1 ] \]
\[ \text{matS} := \text{linalg[matrix]} (2^\text{dim}_V, 2^\text{dim}_V, (i,j) -> S[i,j]) ; \]
\[ \text{matS} := \text{subs} (\text{sol}[1], \text{evalm} (\text{matS})) ; \]

As a further generalization, we are now able to define a \textbf{Clifford-Plücker product}. We will restrict ourselves to the case where we contract a Plücker 1-vector (hyperplane) with an arbitrary polynomial \( X \) as defined above. We turn to \( \text{dim}_V = 3 \), so that hyperplanes are \( e1w2, e1w3 \) and \( e2w3 \).

\[ \text{Gamma[pl]} := \text{proc} (x, u) \]
\[ \text{local} a ; \]
\[ a := &\& (x, &gco_pl(u)) ; \]
\[ \text{contract} (a, 1, \text{scalarpart@LC, B}) + &\& (\& (x, u), 1, `&v`) ; \]
\[ \text{end} ; \]
\[ \text{dim}_V := 3 ; \]
\[ \text{clicollect} (\text{simplify} (\text{drop_t} (\text{Gamma[pl]} (e1w2, e1w2)))) ; \]
\[ \text{clicollect} (\text{simplify} (\text{drop_t} (\text{Gamma[pl]} (e1w2, e1w2we3)))) ; \]
\[ \text{clicollect} (\text{simplify} (\text{drop_t} (\text{Gamma[pl]} (e1w2, e2we3)))) ; \]
\[ \text{clicollect} (\text{simplify} (\text{drop_t} (\text{Gamma[pl]} (e1w2, a*e1w2+b*e2we3+c*e1w3+x*e1+y*e2+z*e3+t*I}
This very brief example shows that the bilinear form of the Clifford-Plücker product is given by the matrix of minors of the original bilinear form $B$ which we have used in the contraction of the hyperplanes with the arbitrary monomial.
Function: Bigebra:-`&map` - maps a product (2 -> 1 map) onto a tensor polynom

Calling Sequence:
\[ t2 := &map(t1,i,pr) \]

Parameters:
- \( t1 \) : a tensor polynomial
- \( i \) : the \( i \)-th tensor slot to act in the pair \((i,i+1)\).
- \( pr \) : a product, i.e. a Clifford polynomial-valued function of two Clifford polynomials, i.e. a 2->1 map.

Output:
- \( t2 \) : the 'product' tensor polynom

Description:

Examples:

```plaintext
> restart:bench:=time():with(Clifford):with(Bigebra):

Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra[help]

Mapping wedge products:
\[
\begin{align*}
&\&\text{&map(\&t(e1,e2)-a*\&t(e3we4,e1)-x*\&t(e1,e1-e2),1,wedge);}
 &\text{drop_t(\%); } \# \text{ convert into a Clifford polynom}
 &\&\text{&map(\&t(e1,e2,e3,e4),1,wedge); } \# \text{ first pair } (1,2)
 &\&\text{&map(\&t(e1,e2,e3,e4),3,wedge); } \# \text{ last pair } (3,4)
 &\&\text{&map(\&t(e1,e2,e3,e4),4,wedge); } \# \Rightarrow \text{ERROR } <=\ 4+1=5 \text{ is not available}
\end{align*}
\]

Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include \&C and \&C[K]. Type ?cliprod for help.

\[
\begin{align*}
&\&\&\&\text{\&t(e1we2)} - a \&t(e1we3we4) - x \&t(0) + x \&t(e1we2)
 &\&\&\&\&\text{e1we2} - a \&t(\text{e3, e4)}
 &\&\&\&\&\text{e1we2}
 &\&\&\&\&\text{\&t(\text{el1, e2, e3, e4})}
\end{align*}
\]

Error, (in Bigebra:-&map) invalid subscript selector

Multiply back again by the Clifford/Graßmann co-product, this gives a loop in tangle notation:

\[
\begin{align*}
&\&\text{&gco(\&t(e1),1);}\&map(\%,1,wedge);
 &\&\text{&gco(\&t(e1),1);}\&map(\%,1,cmul); \# \text{ equivalent}
 &\&\text{&gco(\&t(e1we2),1);}\&map(\%,1,wedge);
 &\&\text{&gco(\&t(e1we2),1);}\&map(\%,1,cmul); \# \text{ different}
\end{align*}
\]

\[
\begin{align*}
&\text{(Id \&t(\text{el}) + (e1 \&t Id)}
 &2 \&t(e1)
 &\text{(Id \&t(\text{el}) + (e1 \&t Id)}
 &2 \&t(e1)
 &\text{(Id \&t(e1we2) + (e1 \&t e2) - (e2 \&t e1) + (e1we2 \&t Id)}
 &4 \&t(e1we2)
 &\text{(Id \&t(e1we2) + (e1 \&t e2) - (e2 \&t e1) + (e1we2 \&t Id)}
 &4 \&t(e1we2) + B_{1,2} \&t(\text{Id}) - B_{2,1} \&t(\text{Id})
\end{align*}
\]

One can show, that wedging back the Graßmann co-product yields 2\^grade of the homogeneous multi-vectors, extended by the multilinearity. The Clifford case works out differently, there one finds 2^dim_V (for non degenerate scalar or co-scalar products). Here, dim_V is the dimension of the vector space \( (V,B) \) that yields \( \text{Cl}(V,B) \). For more information the global variable dim_V, see CLIFFORD_ENV.
\[\text{dim}_V := 2; B := \text{linalg}[\text{matrix}](\text{dim}_V, \text{dim}_V, [a,b,c,d]):\]
\[\text{BI} := \text{linalg}[\text{matrix}](\text{dim}_V, \text{dim}_V, [u,z,t,v]): \text{make BI Id}():\]
\[c1 := \&\text{cco}(&t(e1), 1); \&\text{map}(%, 1, \text{wedge}): \text{drop}_t(%)\; \quad \# \text{as above}\]
\[c2 := \&\text{cco}(&t(e1we2), 1); \&\text{map}(%, 1, \text{cmul}): \text{drop}_t(%)\; \quad \# \text{depends on B}\]
\[c3 := \&\text{cco}(&t(\text{Id} + e1 + e2 + e1we2), 1); \&\text{map}(%, 1, \text{wedge}): \text{drop}_t(%)\; \quad \# \text{depends on BI}\]
\[c4 := \&\text{cco}(&t(e1we2), 1); \&\text{map}(%, 1, \text{cmul}): \text{drop}_t(%)\; \quad \# \text{depends on B and BI}\]

\[
c1 := (\text{Id} \& t e1) - z(e1 \& t e1we2) - v(e2 \& t e1we2) + (e1 \& t \text{Id}) + t(e1we2 \& t e1) + v(e1we2 \& t e2) + 2 e1
\]
\[
c2 := (\text{Id} \& t e1we2) + (e1 \& t e2) - (e2 \& t e1) + (e1we2 \& t \text{Id}) - 4 e1we2 + b \text{Id} - c \text{Id}
\]
\[
c3 := (\text{Id} \& t \text{Id}) + u(e1 \& t e1) + t(e2 \& t e1) + z(e1 \& t e2) + v(e2 \& t e2) + (t z - v u)(e1we2 \& t e1we2) + (\text{Id} \& t e1)
\]
\[
- z(e1 \& t e1we2) - v(e2 \& t e1we2) + (e1 \& t \text{Id}) + t(e1we2 \& t e1) + v(e1we2 \& t e2) + (\text{Id} \& t e2) + u(e1 \& t e1we2)
\]
\[
+ t(e2 \& t e1we2) + (e2 \& t \text{Id}) - u(e1we2 \& t e1) - z(e1we2 \& t e2) + t(e1we2 \& t e1we2) + (\text{Id} \& t e1we2)
\]
\[
+ (e1we2 \& t \text{Id}) - z(e1we2 \& t e1we2) + (e1 \& t e2) - (e2 \& t e1)
\]
\[
z e1we2 + 2 e2 + Id - t e1we2 + 4 e1we2 + 2 e1
\]
\[
c4 := (e1we2 \& t \text{Id}) + t(e1we2 \& t e1we2) + (e1 \& t e2) + (\text{Id} \& t e1we2) - (e2 \& t e1) - z(e1we2 \& t e1we2)
\]
\[
4 e1we2 + t(Id \& c b - Id d a + c e1we2 - b e1we2) + b Id - c Id - z(Id \& c b - Id d a + c e1we2 - b e1we2)
\]

However, we can also map e.g. the meet
\[
> \&\text{map}(&t(e1, e1we2, e1, e2), 2, \text{meet});
\]

\[
> \text{printf("Worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine\n", time() - bench)};
\]

Worksheet took 16.984000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra help page, Bigebra:-meet, Bigebra:-`&v`, CLIFFORD help

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved. Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-`&t` - tensor product

Calling Sequence:
• \( t_0 = \&t(c_1,c_2,...,c_n) \) [or \( c_1 \&t c_2 \&t ... \&t c_n \), not recommended]

Parameters:
• \( c_1, c_2, ..., c_n \) - expressions of `type/clipolynom`

Output:
• \( t_0 \) : is a tensor polynom.

Description:
• At the time of loading, BIGEBRA initializes the tensor product \( \&t \). The operator \( \&t \) is defined using Maple's \texttt{define} facility. Define was patched for two reasons:
  - It had bugs.
  - Scalars had to be changed to be of `type/cliscalar`.
• We recommend to use the prefix form \&t(\(c_1,..,c_n\)) of \&t since the infix form \( c_1 \&t c_2 .. \&t c_n \) can cause problems if no parentheses are used (see below and \texttt{define}).
• The tensor product \&t is only available in its operator (ampersand) form. It is defined as an associative (flat) Maple operator which is multilinear w.r.t. to \texttt{cliscalars}. This allows the user to define Grassmannians, spinor modules, Clifford modules, algebraic varieties etc.
• In the expression \( \&t(c_1,c_2,...,c_n) \), one calls the place 'i', where the parameter 'ci' resides, the i-th slot or place of the tensor product.
• The 'product' rule of a tensor product is simply concatenation of the slots. Associativity allows one to drop parentheses (watch out, parentheses are needed when using the infix form). Associativity is called 'flat' in Maple:
  \( (c_1) \&t (c_2) \&t ... \&t (c_n) = \&t(c_1,c_2,...,c_n) \)
  \( \&t(c_1,...,c_2) \&t \&t(c_3,...,c_4) = \&t(c_1,...,c_2,c_3,...,c_4) \)
• The tensor product is linear in each factor w.r.t. \texttt{cliscalars}.
  \( \&t(c_1,...,a^i+c^i,...,c_n) = a^i \&t(c_1,...,c_i,...,c_n) + b^i \&t(c_1,...,c_i,...,c_n) \)
  where \( c_i,c_i' \) are \texttt{clifford polynoms} and \( a,b \) are \texttt{cliscalars}.
• This tensor product is a prototype in the sense that one can regard it as a graded tensor product also. The difference comes from the functions which are applied to the tensor product, as e.g. the \texttt{switch} and \texttt{graded switch}.
• TO DO: In Maple 6 a new version of \&t will allow to specify the type of scalars, i.e. the domain (ring) the tensor product is built over.

Examples:

```maple
> restart; bench := time(); with(Clifford); with(Bigebra);

Increase verbosity by infolevel[function]:=val -- use online help > ?Bigebra[help]

Infix form (not recommended, see below) and concatenation (i.e. associativity):
> e1 \&t e2;
e1 \&t e2we3 \&t e3we4;
\&t(e1,e2) \&t \&t(e3,e1we2); #\# the middle \&t acts as 'product'.
\&t(e1,\&t(e2,\&t(e1we2,e3)));

  e1 \&t e2
  \&t(e1, e2we3, e3we4)
  \&t(e1, e2, e3, e1we2)
  \&t(e1, e2, e1we2, e3)

The infix form of \&t is peculiar, since it has a BUG !! See this and watch out. It is recommendet not to use the infix form!
> \&t(a*e1,b*e2); # correct

  a b (e1 \&t e2)
```
The reason for this **BUG** is that the infix form gets confused what kind of object it has to treat. `&t` does not know what kind of type it treats and causes an error. Putting in parentheses is necessary. We recommend to use the **prefixform**.

**Linearity** in each slot:

```plaintext
> &t(e1,e2+e3,e1we2);

&t(e1,e2, e1we2) + &t(e1, e3, e1we2)
```

```plaintext
> &t(sin(theta)*e2+cos(theta)*e1we2,e3);

sin(θ) e2 &t e3 + cos(θ) e1we2 &t e3
```

```plaintext
> &t(a*e1+b*e2,c*e3,d*e4,f*e5+g*e6);

acdf &t, , , e1 e3 e4 e5 a c d g &t, , , e1 e3 e4 e6 b c d f &t, , , e2 e3 e4 e5 + b c d g &t(e2, e3, e4, e6)
```

```plaintext
> type(a,cliscalar);

type(exp(x),cliscalar);
type(a*e1,cliscalar);

true
true
false
```

```plaintext
> printf("Worksheet took %f seconds on a 2x 1GHz PIII 1GB RAM machine\n",time()-bench);

Worksheet took 4.197000 seconds on a 2x 1GHz PIII 1GB RAM machine
```

A **cliscalar** is anything which is not of any of these types: `type/clipolynom`, `type/climon`, `type/clibasmon`, or `type/tensorpolynom`.

**See Also:** Biegebra:-&map, Biegebra:-mapop, Biegebra:-mapop2, Biegebra:-contract, Biegebra:-peek, Biegebra:-poke, Biegebra:-switch, Biegebra:-`type/tensorbasmonom`, Biegebra:-`type/tensormonom`, Biegebra:-`type/tensorpolynom`

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.

Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-`&v`   --  the vee (meet) product.
    Bigebra:-meet  --  the meet product.

Calling Sequence:
c3 := &v(c1,c2)  
[or c1 &v c2,  not recommended]  
c3 := meet(c1,c2)  --  synonym.

Parameters:
• c1,c2 -  expressions of `type/clipolynom`

Output:
• c3 -  expression of `type/clipolynom`

Global variables:
• dim_V  -  dimension of the vector space (V,B) that is defined in CLIFFORD as a global variable.

Description:
• The pair of operations wedge (i.e. join) and meet acting on Graßmann multi-vectors make up, together with the duality operator, the Graßmann Cayley algebra. This algebra is of tremendous importance in geometrical applications like robotics, visual perception, camera calibration. However, incidence geometries have their own well developed mathematical theory, see e.g. P. Dembowski, Finite Geometries, Springer Verlag, New York, 1968.

• To avoid confusion we should point out that the notion of a meet is not unique in literature. Let A be a homogeneous decomposable multivector called an extensor. Every such extensor spans a linear subspace of the space over which it was constructed. The span of A is called the support of A, denoted as supp A. Meet and join can be defined in set theoretic terms on the support of extensors. Let A, B denote extensors, one defines:

  A \cup B := \{x \in V \mid x \in \text{supp} A \text{ or } \in \text{supp} B\} \quad \text{i.e. the set theoretic union}
  A \cap B := \{x \in V \mid x \in \text{supp} A \text{ and } \in \text{supp} B\} \quad \text{i.e. the set theoretic intersection}

The operators cup and cap are the same as in set theory. Under these operations every set is an idempotent: A \cup A = A \text{ and } B \cap B = B. Moreover, one finds cup o cup = Id and cap o cap = Id for these operators. Including the set theoretic operation of taking the complement, (A -> |A with A \cup |A = whole space, where we have used, in lack of an over bar, the Graßmann notation of a preceeding bar), this constitutes the structure of an ortho-modular lattice. Boolean logic is based on this construction. The two operations of meet and join are related by de Morgan laws:

|\{ A \cup B\} = (|A) \cap (|B)
|\{ A \cap B\} = (|A) \cup (|B).

In terms of logic we have: cup = and, cap = or,  | = not.

In CLIFFORD and Bigebra packages, the meet and join are defined in the following way:

The wedge product of two extensors A and B is an extensor C which has as support the disjoint union of the supports of A and B. However, extensors having the same support are isomorphic (interchangable). The meet is usually defined using a symmetric correlation in the projective space \(P^\text{dim}(V)\). It needs thus a theorem to show that the meet is independent from its construction. Graßmann defined the meet, which he called regressive product, in [A2], 1862, §5, No. 94 page 61ff. The regressive product was already present in [A1], chapter 3, §125ff. Graßmann edited in 1877 a reprint with annotations where he gave some explanations on his presentation. A careful reading shows that the regressive product was present already in 1844. The Ergänzung is not explicit in [A1], but Graßmann discusses the grade of the complement \(|A\) which he calls there 'Ergänzzahlen' (A1 §133)) using the so called 'Ergänzung' (Graßmann A2, §4. No. 89 page 57), which we defined already above as \(|\), of an extensor A to be \(|A\). In analogy to de Morgan laws (which he most likely did not knew) as:

\(| (A \vee B) := (|A) \text{ and } (|B).\)

Graßmann used no sign for products, having over 16 of them working, many at the same time and their type had to be deduced
by context. He used furthermore no parentheses which makes his writings cumbersome to read. The \( \Lambda \) sign mutated from an (uppercase) Lambda used by Burali-Forti and Marcolongo to be the wedge of Bourbaki.

The usage of the Ergänzung points out clearly that the meet depends on the dimension of the space. We will see below, that this definition of the meet is computationally very ineffective.

Alfred Lotze, (Über eine neue Begründung der regressiven Multiplikation extensiver Größen in einem Hauptgebiet n-ter Stufe, Jahresbericht der DMV, 57:102-110,1955) defined a universal formula for the regressive product of r-factors. He showed that if one considers the n-1 dimensional space as a space of co-vectors then the original wedge product becomes by the same formula the regressive product of the co-vectors, pointing out the fact that a symmetric correlation is needed for this purpose. That is: (n-1)-multi-vectors are not co-vectors, but may be seen as reciprocal vectors. In [4], G.-C. Rota and coworkers gave a definition of the meet in terms of a Peano algebra which is essentially the same construction. However, they used the notion of Hopf algebra which allows to write down the formulas in a comprehensible way.

The Graßmann wedge product has as logical counterpart in the exclusive or xor, the Ergänzung is not w.r.t. the choosen volume form of the space \( V \) the Graßmann algebra is build over. The meaning of the meet follows from his duality relation.

- In Bigebra, the meet and \&v (vee) products are implemented as follows (note the order of factors in the bracket!):

  \[
  \text{meet}(c_1,c_2) := [c_2(1),c_1] c_2(2) \\
  \text{&v}(c_1,c_2) := c_1(1) [c_2,c_1(2)],
  \]

  where the bracket \([ , ]\) is a scalar valued alternating multilinear volume form and the co-products are given in Sweedler notation. It can be shown (and is tested below) that both forms represent the same operation.

- The Hopf algebraic definition of the meet gives us a great deal of computational benefits as we will show below in some benchmarks. However it works exactly as the Graßmann regressive product.

- Graßmann introduced the so called stereometric product, which, being context sensitive, switches between the wedge and the \&v (vee-) product. Using polymorphism this could be implemented, and the user can easily program such a wrapper function. We found it peculiar to implement it using the same notation for basis elements for vectors and co-vectors.

- The meet as defined here is independent of the assigned scalar product \( B \) or the assigend co-scalarproduct \( BI \). In fact it can be shown that the vee-product is \( \text{SL}_n \) invariant. If one is interested in projective geometry, the invariants derived from meet and join are \( \text{GL}_n \) invariants.

- The meet product is related to the notion of a Hopf algebraic integral [3]. As a remarkable fact, in any Clifford Hopf gebra over \( \dim V = 2 \) one is not able to find an non zero integral. The notion of meet has thus to be reconsidered in the deformed case.

Examples:

```
> restart:bench:=time():with(Clifford)::with(Bigebra):

Increase verbosity by infolevel[`function`]-val -- use online help > ?Bigebra[help]
Infix form (not recommended, see help page on &t). Note that we have not assigned a scalar- or co-scalarproduct.

> dim_V:=2:
  e1 &v e2;
  e1 &v elwe2, elwe2 &v e2, elwe2 &v elwe2;

Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.

-Id
  e1, e2, elwe2

Firsts of all let us check that both versions of teh meet compute indeed identical:

> for i from 1 to 4 do
  dim_V:=i:
  bas:=cbasis(dim_V);
  X:=add(_X[i]*bas[i],i=1..2^dim_V):
  Y:=add(_Y[i]*bas[i],i=1..2^dim_V):
```

printf("In dimension %d the equation \`meet(X,Y)=&v(X,Y)` is
%a\n",dim_V,evalb(0=simplify(meet(X,Y)-&v(X,Y))));

od:

In dimension 1 the equation \`meet(X,Y)=&v(X,Y)` is true
In dimension 2 the equation \`meet(X,Y)=&v(X,Y)` is true
In dimension 3 the equation \`meet(X,Y)=&v(X,Y)` is true
In dimension 4 the equation \`meet(X,Y)=&v(X,Y)` is true

The following example will show, that the meet and the join are exterior products on their own right and cannot be distinguished. This makes it unnecessary to use the V (vee) sign for the ordinary wedge product as Rota promoted to stress the analogy with set theoretic operators. We will see that the join of points is the meet of (hyper) planes and the meet of points is the join of (hyper) planes. To demonstrate this, we compute the meet for a Graßmann basis. We check associativity, unit, and show that this product is an exterior product on its own right on reciprocal (sometimes called wrongly dual) vectors (i.e. hyperplanes). The reciprocal meet is then defined to be the meet w.r.t. hyper-planes. Then it is shown that this reciprocal meet is indeed the wedge (join of points) with which we started. [To give a crude reciprocal meet we use Graßmann's Ergänzung, but a combinatorial evaluation is also possible but proved to be too long for this help page.]

We present our demonstration in dimension 3. Define the n-1 (i.e. 2-) vectors A(i). These multi-vectors are the images of a covector basis under dualization, see \[4,3\] and should be called reciprocal vectors. Their definition involves a symmetric correlation. The 'meet' or \`&v` (vee) product of vectors acts as an exterior (or wedge) product on these reciprocal vectors. This is an immediate consequence of categorial duality and is related to the Plücker coordinatization of hyper-planes.

While we show here explicitly how to define a a Meet and Join for hyperplanes, there is an generic Graßmann co-product &gco_pl in the package which could be used with some benefits for the performance, but would probably obscure out aim here.

> dim_V:=3:
> # A(i),A(ij) etc are new basis elements
> # # ==> define the hyperplane basis A(i), A(ij) etc
> A:=proc(x)
> local T;
> T:=table([123=-Id,
> 31=-e2,23=-e1,12=-e3,
> 13= e2,32= e1,21= e3,
> 3=e1we2,2=-e1we3,1=e2we3,
> 0=e1we2we3]);
> RETURN(T[x]);
> end:
> # # w2A is a translation procedure which turns the output
> # into the new A basis of reciprocal vectos (plane vectors)
> # # ==> w2A  (wedge basis to hyperplane basis A(i)
> w2A:=proc(x)
> local bas,y;
> bas:={Id=-'A(123)',
>  e1=-'A(23)', e2= 'A(13)',e3= -'A(12)',
>  elwe2='A(3)',elwe3=-'A(2)',e2we3='A(1)',
>  elwe2we3='A(0)'};
> RETURN(subs(bas,Clifford:-reorder(x)));
> end:
> # # &V (uppercase) is a wrapper function to make the usage of the
> # # A(i) basis more comfortable
> # # ==> &V can act on the hyperplane basis A(i) seen as wedge multivectors
> # # and yields A(jk) hyperplane 2-vectors
> # [same goes for the Meet (formerly meet)]
> `&V`:=proc(x,y)
After these preliminary definitions we can directly show the meet to be the 'wedge product of hyperplanes'. First of all we check some elementary properties of the meet acting on hyperplanes.

> A(0),w2A(A(0)); # The 'scalar' w.r.t. the &v product
  A(1),w2A(A(1));
  A(2),w2A(A(2));
  A(3),w2A(A(3)); # (reciprocal) vectors

  elwe2we3, A(0)
  e2we3, A(1)
  -elwe3, A(2)
  elwe2, A(3)

> &V(A(0),A(1)),&V(A(1),A(0)); # shows A0 to be the identity
  Meet(A(0),A(1)),Meet(A(1),A(0)); # synonym but internally computed differently

  A(1), A(1)
  A(1), A(1)

Now we produce reciprocal bi-vectors (bi-hyperplanes to be precise) A(ij) and the volume element A(123)

> &V(A(1),A(2)),&V(A(2),A(3)),&V(A(3),A(1));  # BI-HYPERPLANES
  &V(A(1),&V(A(2),A(3))),                     # VOLUME ELEMENT EVALUATES TO -ID
  &v(A(1),eval(&V(A(2),A(3))));               # eval is needed here to apply A(23)

  A(12), A(23), -A(13)
  A(123), -Id

There are no higher multi-hyperplanes (reciprocal multi-vectors) and the following expressions evaluate to zero:

> &V(A(1),A(123)),&V(A(12),A(23));

  0, 0

The bracket for co-vectors can be defined using the fact that -Id is the volume in the space of hyperplanes as the projection onto -Id. Hence we can define the reciprocal meet RMeet of reciprocal vectors. This is also a demonstration how to extend the features of the CLIFFORD/Bigebra packages:

> B:=linalg[diag](1$dim_V):     ### internally used for Grassmann Erg"anzung
  `&RMeet`:=proc(x,y)                       ### function co-meet
    local yy,res,lst,var_i,v1,v2;
    option `Copyright (c) Ablamowicz, Fauser 2000/02. All rights reserved.`;
    ## crude version of the Grassmann co-product on the 'multivector plane space'
    yy:=&t(e1we2we3,&gco(eval(cmul(e3we2we1,y))),e1we2we3);
    yy:=&map(tcollect(&map(switch(yy,3),3,cmul)),1,cmul);
    ## if type(yy,tensorbasmonom) or type(yy,tensormonom)then
    lst:=[yy];
    else
      lst:=[op(yy)];
    fi;
    res:=0;
    for var_i in lst do
      v1,v2:=peek(var_i,1);
      res := res - scalarpart(&v(eval(x),v1))*drop_t(op(v2));
To exemplify out claim, let us define the two mutually reciprocal basis sets of points, joined points (i.e., lines) and point space volume and the hyperplanes bi-hyperplanes (i.e., lines) and the volume of the hyperplane multi-vector space -Id.

```maple
> bas := cbasis(3);
bas := [Id, e1, e2, e3, e1we2, e1we3, e2we3, e1we2we3]
bas_A := [A(0), A(1), A(2), A(3), A(12), A(13), A(23), A(123)];
```

For easy comparison, we compute the multiplication table of the RMeet product. This multiplication table is a tensor of rank three.

```maple
> Mul_tab_RMeet := linalg[ matrix ]( 2^dim_V, 2^dim_V, ( i, j ) -> 0 ) :
for i from 1 to 2^dim_V do
    for j from 1 to 2^dim_V do
        Mul_tab_RMeet[i, j] := reorder( &RMeet(bas[i], bas[j]) ) ;
    od: od:
evalm(Mul_tab_RMeet);
```

Our final goal is to show, that the above defined multiplication for RMeet (the meet of hyperplanes) is equivalent to the wedge product of points. We compute therefore the multiplication table for the wedge also:

```maple
> Mul_tab_wedge := linalg[ matrix ]( 2^dim_V, 2^dim_V, ( i, j ) -> 0 ) :
for i from 1 to 2^dim_V do
    for j from 1 to 2^dim_V do
        Mul_tab_wedge[i, j] := &w(bas[i], bas[j]) ;
    od: od:
evalm(Mul_tab_wedge);
```

The final check is to add both matrices which gives zero. This shows that up to a sign (which is irrelevent in projective plane geometry) the products are the same. Or, as operator equation:

\[
\text{RMeet}(x,y) = -\text{wedge}(x,y)
\]

The sign belongs to the fact that in three dimensions we find that the volume element squares to negative identity, which means that we would reach the original wedge after a second turn in our argumentation. However, we resist to demonstrate this explicitly here.
Finally we will provide some **Benchmarks** which shall show how efficient the two alternate definitions of the meet are. One, as adopted recently by Hestenes and followers, is based on the Grassmann's Ergänzung and the other is based on Hopf algebra methods as employed in Bigebra and given by Lotze and Rota.

As a **Benchmark** we compute 100 times a certain meet (this is not a good idea, since some functions may remember its results, e.g. the wedge product from the CLIFFORD package, but it gives nevertheless a feeling what is going on).

The Hopf algebraic case needs:

```plaintext
> s:=time():
for i from 1 to 100 do
    &v(e1we2,e2we3);
od:
printf("This took us %f seconds",time()-s);
&e(e1we2,2we3);
Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.
This took us 45.825000 seconds
```

Since we compute the Clifford product using a very fast Hopf algebraic function cmulRS, this works out faster. However, we could speed up &v by directly employing the Hopf algebraic routines and avoinging wrapper functions as 'peek'. Furthermore we have not computed the inverse of the Ergänzung but introduced simply e3we2we1 which is (e1we2we3)^(-1) in our case.

Now let us go for a non-orthogonal but still symmetric bilinear form (a polar form of a quadratic form or a symmetric correlation) and check what happens there:

```plaintext
> restart:bench:=time():with(Clifford):with(Bigebra):  # reload everything to be fair
> dim_V:=3:B:=linalg[diag](1$dim_V):
> s:=time():
for i from 1 to 100 do
    cmul(e3we2we1,wedge(cmul(e1we2we3,e1we2),cmul(e1we2we3,e2we3)));
od:
printf("This took us %f seconds",time()-s);
cmul(e3we2we1,wedge(cmul(e1we2we3,e1we2),cmul(e1we2we3,e2we3)));
```

Since we compute the Clifford product using a very fast Hopf algebraic function cmulRS, this works out faster. However, we could speed up &v by directly employing the Hopf algebraic routines and avoinging wrapper functions as \'peek\'. Furthermore we have not computed the inverse of the Ergänzung but introduced simply e3we2we1 which is (e1we2we3)^(-1) in our case.
Let us use an arbitrary bilinear form in 3 dimensions:

```
> B := linalg[matrix](dim_V, dim_V, (i, j) -> b[i, j]);
```

The Hopf algebraic meet remains to be

```
> &v(elwe2,e2we3);
```

while the 'meet' computed using the Ergänzung does not even yield a homogeneous multi-vector, but a Clifford polynomial:

```
> clicollect(simplify(cmul(e3we2we1, wedge(cmul(e1we2we3, e1we2), cmul(e1we2we3, e2we3)))));
```

Moreover, we can go beyond the possibilities of the Ergänzungs method since we can compute the meet in the presence of a non-symmetric bilinear form (which cannot be derived from a quadratic form by polarization) using Hopf algebra methods. Our meet works **independently of the assigned bilinear form** while the Eränzungs method needs an orthogonal non-degenerate bilinear form (which is the polar form of the symmetric correlation, i.e. a quadratic form).
\[ + b_{2,1} b_{3,1} b_{1,1}^{2} + 2 b_{2,2} b_{3,1} b_{1,1} b_{1,3} - b_{2,1} b_{1,1} b_{1,2} + b_{2,2} b_{1,1}^{2} b_{1,1} + b_{2,1} b_{1,1} b_{1,3} + b_{2,2} b_{1,1} b_{1,3} + b_{2,2} b_{1,1} b_{1,3} + b_{2,2} b_{1,1} b_{1,3} \]

See Also: `&map`, `Bigebra:-peeK`, `Bigebra:poKe`, `Bigebra:switCh`
Function: Bigebra:-bracket - the bracket of Peano space (i.e. invariant theory)

Calling Sequence:
sc := bracket(c1,c2,c3,...)

Parameters:
• ci  : Clifford polynomials

Output:
• sc  : a cliscalar

Global variables:
• dim_V  : the dimension of the Peano space

Description:

Examples:
> restart:bench:=time():with(Clifford):with(Bigebra):
Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]
Set an arbitrary bilinear form to show that the bracket does not depend on this setting:
> dim_V:=4:B:=linalg[&t(matrix)](dim_V,dim_V,(i,j)->b[i,j]);
\[
\begin{bmatrix}
  b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\
  b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\
  b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\
  b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4}
\end{bmatrix}
\]
> bracket(&t(e1,e2,e3,e4)),bracket(&t(e1,e2),&t(e3,e4)),
bracket(e1,&t(e2,e3),e4),bracket(e1,e2,e3,e4);
bracket(e1),bracket(e2we3),bracket(e2we3we4);bracket(e1,e2);
Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type
?cliprod for help.

> &t(e1we3we4,e2+e3,e2+e3,e4);
contract(%[,1,bracket);
\[
&(e1we3we4, e2, e2, e4) + &(e1we3we4, e2, e3, e4) + &(e1we3we4, e3, e2, e4) + &(e1we3we4, e3, e3, e4)
(e2 &t e4) + (e3 &t e4)
\]
> printf("Worksheet took %f seconds on a 2x 1GHz PIII 1MB RAM machine\n",time()-bench);
Worksheet took 5.178000 seconds on a 2x 1GHz PIII 1MB RAM machine

See Also: Bigebra:-contract, Bigebra:-`&v`, Bigebra help page, Bigebra:-pairing, Bigebra:-EV

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2003/BF/RA.
**Function:** Bigebra:-contract - contraction of adjacent slots of a tensor, lowers the tensor rank by two.

**Calling Sequence:**
\[ t2 := \text{contract}(t1,i,f) \]

**Parameters:**
- \( t1 \) : a tensor polynomial
- \( i \) : the \( i \)-th tensorslot to act on the pair \((i,i+1)\).
- \( f \) : a scalar valued function of two Clifford polynomials; \( f : \Lambda V \times \Lambda V \rightarrow k \).

**Output:**
- \( t2 \) : the contracted tensor polynomial.

**Description:**
- The contraction is needed to contract tensor polynomials by arbitrary 2->0 mappings \( f \). Depending on the type of the mapping \( f \), the contraction can be a contraction in the sense of tensor calculus, i.e. like a sum over co- and contravariant indices, or like a (co-) scalar product acting on two indices of the same valence (co- or contravariant). For more information see [3]. The contraction can be seen as a cup tangle, i.e. 2->0 but on yet unoriented tangles.
- Possible scalar-valued functions of the Bigebra package are, \text{bracket@wedge}, or \text{bracket} with two arguments, \text{pairing}, \text{EV}, \text{scalarpart@cmul}, etc...
- A more detailed help page is planned in the future.

**Examples:**

```maple
> restart; bench := time(): with(Clifford); with(Bigebra):
Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]
Set the dimension and a symmetric bilinear form
> dim_V := 4: B := linalg[matrix](dim_V, dim_V, (i,j)->abs(i-5/j));

\[
B := \begin{bmatrix}
4 & 3 & 2 & 1 \\
2 & 3 & 4 \\
3 & 1 & 3 & 4 \\
2 & 4 & 7 \\
3 & 7 & 11 \\
1 & 2 & 3 & 4
\end{bmatrix}
\]

> contract(&t(e1,e2),1,scalarpart@cmul);
> contract(&t(e1,e2,e3),2,scalarpart@cmul);
> contract(&t(a*e1,e2-b*e4,e3),1,scalarpart@cmul);
> contract(&t(e1),2,scalarpart@cmul);  # ERROR -> only rank one tensor
Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.
> contract(&t(e1,e2,e3we4),1,bracket@wedge);
> contract(&t(e1,e2we3we4,y*e1-e3),2,bracket@wedge);
> contract(&t(e1we2,e3we4)-&t(a*e1we3,e2we4-b*e4,e3),1,bracket@wedge);
> contract(&t(e1,e2,e3,e4),2,bracket@wedge);
```
contract(&t(e1,e2,e3,e4),5,pairing);  # ==> ERROR <= no 5th and 5th tensor slot

1
-γ &t(e1)
1 + a &t(e3)
0

Error, (in Bigebra:-peek) improper op or subscript selector

> contract(&t(e1we2,e3we4),1,pairing);
  contract(&t(e1,e2we3we4,y*e1-e3),2,pairing);
  contract(&t(e1we2,e3we4)-&t(a*e1we3,e2we4-b*e4,e3),1,pairing);
  contract(&t(e1,e2,e3,e4),2,pairing);

-5
12
0
-5 5
12 + a &t(e3)
1/3(e1 &t e4)

> contract(&t(e1we2+a*e3we4,e3we4-y*e1we2),1,EV);
  contract(&t(e1,e2we3we4,y*e1-e3),2,scalarpart@meet);
  contract(&t(e1we2,e3we4)-&t(a*e1we3,e2we4-b*e4,e3),1,scalarpart@LC);
  contract(&t(e1,Id,Id,e4),2,scalarpart@wedge);

-γ + a
γ &t(e1)
-5 5
12 + a &t(e3)
e1 &t e4

> printf("Worksheet to %f seconds to compute on a 2x 1GHz PIII 1GB RAM
  machine",time()-bench);

Worksheet to 7.862000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra help page, Bigebra:-meet, Bigebra:-`&v`, Bigebra:-pairing, Bigebra:-EV, CLIFFORD help page

(c) Copyright December-16-1999 --2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
**Function:** Define:-define and Define:-definemore -- (partially) patched version of Maple 6's define.

**Calling Sequence:**

`define(op, prop1, prop2, ...)`

**Parameters:**

- `op` : the name of an operator (or function) which is defined by 'define', e.g. `&t`, i.e. the BIGEBRA tensor product.
- `prop_i` : properties like 'multilinear', 'flat' (which we have tested) and functional programming.
- `new` : an option domain may be specified to use k-linearity over a ring k `domain=type`

**Output:**

- none.

**Warning:**

- multilinearity works only in the associative (Maple's flat option) case.

**Description:**

- Maple's define has bugs so it cannot be used for our purpose. For this reason, Bigebra installs a patch to 'define' which allows to use the 'flat' (associativity) and 'multilinear' properties. The name of such an 'defined' operator is usually composed by an &((amphersand) followed by some name, i.e. &t for the tensor product, but see the tensor product help page for some peculiarities of the &((amphersand) operators, which may cause errors too if used in its infix form.
- A second reason for patching define is that the multilinear option handles constants not in an appropriate way. The multilinear option considers beside numbers (integers, floats, ...) any other algebraic expression as 'non constant', i.e. keeps it inside of the userdefined &((amphersand) function/operator. Since we want to consider e.g. modules, this behavior is not convenient. In fact, a tensor product defined by define (after having fixed the errors) is only a Z-tensorproduct or an R-tensorproduct, if floats are used. Interger, real or complex variables have to be added to the 'constants' list painfully. This was inconvenient.
- The user can define with our 'define' new associative multilinear operators over a (common!) domain which he specifies by overwriting the type cliscalar of the CLIFFORD package. Any element which has type 'cliscalar' is then considered as a constant and put in front or the multinear function. In this way, modules, varieties, polynomial rings in several variables (the option orderless was not tested, but is quite trivial an should work) etc. can be 'defined'. That is one defines a tensor product over the 'cliscalars'.
- In further upcomming versions of BIGEBRA (for Maple 6) the user can supply an arbitrary domain. From Maple 7 on, local types are possible and BIGEBRA will allow then to have separate domains for any user defined &((amphersand) operator.

**Examples:**

```maple
> restart:
libname:=libname[2..-1]; # make sure we are not useing the patched define from <Cliffordlib> library
libname := "C:\Maple6/lib"

BUG in multilinear option:

Define &r as a multilinear function

> define(`&r`,multilinear);

Check some cases which work. Note that 'false' and 'true' compute like constants.

> &r(-e1);                  # but see flat,multilinear below
&r(2*e1,3*e2+5*e3);
&r(a*e1,e2-e3,e4+e5);     # note a is not handled here
constants:=constants,a;
&r(a*e1,e2-e3,e4+e5);     # add a to the constants
&r(a*e1,e2-e3,e4+e5);     # now a is put in front
&r(false*e1,true*e2);     # this 'works' since false,true are 'constants'

−&r(e1)
```
\begin{align*}
6(e_1 \& e_2) + 10(e_1 \& e_3) \\
\& (a_1 e_2, e_4) + \& (a_1 e_3, e_5) - \& (a_1 e_1, e_4) - \& (a_1 e_2, e_5) \\
\text{constants} := \text{false, } \gamma, \infty, \text{true, Catalan, FAIL, } \pi, a \\
\& (a_1 e_2, e_4) + \& (a_1 e_3, e_5) - \& (a_1 e_1, e_4) - \& (a_1 e_2, e_5)
\end{align*}

\begin{verbatim}
> e := exp(1); evalf(e);
\& (e*e1, 2*e1);
\end{verbatim}

\begin{verbatim}
e := e
2.718281828
2 \& (e_1 &\& e_1)
\end{verbatim}

But see this (loops until out of memory or to many levels of recursion):
\begin{verbatim}
> \& (2.5*e1);  # bug loops infinitely
Error, (in \&r) too many levels of recursion
\end{verbatim}

Let's see what happens if we use negative elements
\begin{verbatim}
> \& (-e1, -e_2, e_3);  # \&r(e1, e_2, e_3) expected
\& (-e1, -e_2, e_3)
> \& (-e1, -e_2);  # \&r(e1, e_2) expected
\& (-e_1) \& (-e_2)
> \& (\& (-e1));  # \&-r(\&r(e1)) expected
\& (-r(\&r(e1)))
\end{verbatim}

This renders the multilinear function to be useless, even the treatment of constants is poor, since any multilinear function has to be defined over some ring which has to be specified, but cannot be so in Maple's define.

Let us check the flat option:
\begin{verbatim}
> restart:
libname := libname[2..-1];  # make sure we are not using the patched define from
<Cliffordlib> library
define(`\&r`, flat);
\end{verbatim}

\begin{verbatim}
> \& (\&r(e_1, e_2), e_3);
\& (\&r(e_1, e_2), \& (e_2, e_3));
\& (e_1, e_2, e_3)
\& (e_1, e_2, e_2, e_3)
\end{verbatim}

\begin{verbatim}
> \& (-e1, \&r(e2));
\& (\& (e1));
\& (-e_1) \& r e_2
\& (\& (e1))
\end{verbatim}

Old Maple 5 bug partially fixed:
\begin{verbatim}
> \& (\& (-e1));
\& (-e_1)
\end{verbatim}

\begin{verbatim}
> restart:
libname := libname[2..-1];  # make sure we are not using the patched define from
<Cliffordlib> library
define(`\&r`, flat, multilinear);
\& (\& (-r(e1)), \& (-r(e2)));  # bug, does not work out anything
\end{verbatim}

\begin{verbatim}
> (\& (-r(e1))) \& (\& (-r(e2)))
\end{verbatim}

A good deal of Maple 5 bugs are resolved in Maple 6 we check for:
\begin{verbatim}
> \& (\& (e1, -e2));  # expected -(e1 \& e2)
-(e1 &\& e2)
> \& (\& (-e1));  # expected -&(e1)
\end{verbatim}
Now we load the Bigebra package which uses a patched 'define':

```maple
> restart: bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel['function']=val -- use online help > ?Bigebra[help]
> define("&r",multilinear);
> &r(2.5*e1);  # works now
&r(e1);
> &r(-e1);  # OK now
2.5 &r(e1)
> &r(&r(-e1));
> &r(2.5*(e1)); # works now
2.5 &r(e1)
&r(2.5 &r(-e1)); # OK now
> &r(e1);
2.5 &r(e1)
> &r(2.5 &r(e1));
> &r(-e1);
2.5 &r(e1)
> &r(e1);
> &r(e1);
```

Note, that flat, multilinear works, but we still cannot define nonassociative operators! This will probably be fixed in a new version of Bigebra. However, the tensor product (already defined in Bigebra) is associative i.e. flat and multilinear hence it works out correctly:

```maple
> &t(2.5*e1);
> &t(-e1,-e2);  # (-e1) &r (-e2) formerly
> &t(&t(-e1));  # formerly Error, too many levels of recursion
> &t(-&t(e1),-&t(e2));  # formerly Error, too many levels of recursion
```

No additional tensor slots are created now:

```maple
> &t(&t(e1,-e2));  # formerly -&t(1,e1,e2)
> &t(&t(-e1));   # formerly -&t(1,1,e1)
> &t(-e1);   # formerly -(1 &t e1)
```

This shows examples, but can not prove `&t` to be a flat multilinear operator. However, Bigebra's tensor product was tested in numerous huge worksheets (running for days using over 700MB RAM) and produced mathematically reasonable and afterwards (manually) provable output, so we are convinced to have a safe device.

User defined types:

The user may use the type cliscalar to define tensor products i.e. associative (i.e. flat) multilinear functions himself. For this purpose, he has to re-define the type 'cliscalar' to check for the ring he wants to use. Below we define tensor products T1 and T2 over the polynomial ring in x and the Laurent polynomials in the variables x,y,z. It is also possible to redefine a global variable _scalartypes from CLIFFORD. See Cliff5[CLIFFORD_ENV] for more information.

Example 1: Unprotect the type cliscalar which was protected by CLIFFORD. Define a new operator &T1 to be associative and multilinear. Re-define the scalars to be polynomials over the integers in the variable x, i.e. belong to Z[[x]].

```maple
> unprotect('type/cliscalar');  # type/cliscalar was protected by CLIFFORD
'type/mydomain':=proc(expr) type(expr,polynom(integer,x)) end:
define('&T1', flat, multilinear, domain='mydomain');
```
Check some types to be in our domain (Z[[x]] = `cliscalar`):

> type(2*x+3*x^2+5,mydomain);  # true, since in Z[[x]]
true

> type(x/2,mydomain);          # false, no fractions allowed
false

> type(2*y^2+3*y-1,mydomain);  # false, polynomial variable is not x
false

A second example, using algebraic functions as domain, i.e. define a variety:

> unprotect(`type/cliscalar`):  # was done already above

`type/mypolydom`:=proc(expr) type(expr, radalgfun(rational,[x,y,z])) end:

> define(`&T2`,flat,multilinear,domain='mypolydom');

> type(x/(1-x*y),mypolydom); # true, belons to the ring k[[x,y,x^(-1),y^(-1)]]
true

> type(sin(x),mypolydom);    # false, transcendental
false

> type(a^2,mypolydom);       # false, belons to the polynomial ring k[[a]]
false

> type(y^3,mypolydom);       # true, belons to the polynomial ring k[[y]]
true

Some examples of tensor products:

> collect(&T1(3*x^2-x+5,x-x^4),`&T1`);

$$
(3x^3 - 3x^6 - x^2 + x^5 + 5x - 5x^4)(1 &T1 1)
$$

$$
(-x^2 + 5x)(1 &T1 y) + 3x^3(y &T1 y) - 3x^6(y &T1 1) + (x^3 - 5x^4)(1 &T1 1)
$$

Example 2: Symmetric tensor products:

> unprotect(`type/cliscalar`):  # was done already above

`type/myinteger`:=proc(expr) type(expr, integer) end:

> if assigned(`&S`) then unassign(`&S`) fi:

define(`&S`,flat,multilinear,orderless,domain='myinteger');

> &S(x1,x2), &S(x2,x1);

$$
(x_1 + x_2, 2x_2 + x_1 + x_3);
$$

$$
3(x_1 &S x_2) + 2(x_2 &S x_2) + (x_1 &S x_1) + (x_1 &S x_3) + (x_2 &S x_3)
$$

Now we define a (crude) symmetric product on these monomials:

> `&s`:=proc()
local lst, st, res, i, fun;
    if nargs=0 then RETURN(0) fi;
    if nargs=1 then RETURN(`'&s'`(args)); fi;
    ### WARNING: note that `I` is no longer of type `^`
    fun:=proc(a1) local k; if type(a1,`^`) then seq(op(1,a1),k=1..op(2,a1)) else a1 fi
    end:
    lst:=sort(map(fun,[args]),address);
    st:=[op(lst)];
    res:=[];
    for i from 1 to nops(st) do
        res:=res,[op(i,st)];
    od;
    RETURN(res);
end:

> &S(x1,x2,&S(x1,x2));

$$
\frac{x &T2(1, \sin(x) b, a^2)}{1 - x y} + \frac{x y^3 &T2(1, \sin(x) b, b)}{1 - x y}
$$
This allows us to multiply symmetric functions by applying the multiplication to monoms:

Let $s_{123} = x_1 + x_2 + x_3$ be a symmetric polynomial we compute $s_{123} * s_{123}$:

```maple
> eval(subs(`&S`=`&s`,&S(x1+x2+x3,x1+x2+x3)));
```

Or compute $s_1^2 * s_{123}$

```maple
> eval(subs(`&S`=`&s`,&S(x1^2,x1+x2+x3)));
```

The results are well known from the theory of symmetric functions:

$s_{123} * s_{123} = s_1^2 + s_2^2 + s_3^2 + 2 * s_{12} + 2 * s_{13} + 2 * s_{23}$

$s_1^2 * s_{123} = s_1^3 + s_{(1^2)2} + s_{(1^2)3}$

As a further device these results should be expanded in a canonical basis of symmetric polynomials, which can be done by the user!

**NOTE:** This device has to be used with great care, since it was not well tested! Certain definitions of types may interfere with the abilities of `define`.

```maple
> printf("Worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);
```

Worksheet took 2.853000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

(c) Copyright December-16-1999, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.

Last modified: December 26, 2001/BF/RA.
Function: Bigebra:-drop_t - drops the tensor symbol &t from tensors of rank one

Calling Sequence:

c1 := drop_t(p1)

Parameters:

• p1 : is a tensorpolynom, generally of rank one (but drop_t works on general elements)

Output:

• c1 : a Clifford polynomial or an expression sequence if the tensor is of rank > 1

Description:

• For computational reasons it was convenient to stay with expressions like &t(Id) or &t(e1we2), even if those elements are Clifford polynomials. To get rid of these tensor symbol &t, use drop_t.

• Drop_t is a helper function, that was useful in some worksheets, it may be replaced in later versions of Bigebra.

• Drop_t was especially useful during solving equations in Clifford polynomials if one searches for elements fulfilling some tangle equation having one outgoing line (n->1 maps). If equations have to be solved for operators acting inside of the tangle on arbitrary elements going through the tangle, one should use tsolve1.

Examples:

> restart: bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra[help]

Drop_t on homogenous tensors:

> drop_t(&t(e1we2));
drop_t(&t(Id));
drop_t(&t(e1)+&t(e2)+&t(e3));

\[ e1we2 \]
\[ Id \]
\[ e1 + e2 + e3 \]

> drop_t(a*&t(e1));
drop_t(&t(a*e2));
drop_t(x*&t(a*e1we2+b*e2-c*e1we2we3)-y*&t(e4));

\[ a e1 \]
\[ a e2 \]
\[ x(a e1we2 + b + e2 - c e1we2we3) - y e4 \]

Abuse of drop_t:

> drop_t(&t(a*e2we3,e1we4));
drop_t(&t(a*e1+b*e2,e3));

\[ a e2we3 \]
\[ a e1 + b e2 \]

> printf("Worksheet to %f seconds on a 2x 1GHz PIII 1GB RAM machine",time()-bench);

Worksheet to 2.724000 seconds on a 2x 1GHz PIII 1GB RAM machine

[>

NOTE that drop_t acts like a projection on the first tensor slot and that the other slots are lost, no error message is delivered to the user, watch out. If you need access to other tensor slots, see Bigebra:-peek.

See Also: Bigebra:-&t, Bigebra:-type/tensorpolynom, Bigebra:-peek

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
**Function:** Bigebra:-EV - the evaluation map

**Calling Sequence:**

```
sc := EV(c1,c2)
```

**Parameters:**

- `c1, c2` are Clifford polynomials

**Output:**

- `sc` is a scalar.

**Description:**

- The evaluation maps, `EV : \( \mathbb{V} \times \mathbb{V} \rightarrow k \)`, or `EV : \( \mathbb{V} \times \mathbb{V} \rightarrow k \)`, is defined via the action of a dual element, called also linear form, on a multi-vector polynomial. The dual element is a co-multi-vector polynomial. However, since we have not yet supplied different bases in CLIFFORD/Bigebra, co-elements are represented also as Clifford polynomials. The user has to trace which elements in which slot of a tensor are considered to be co-elements!

- The evaluation is free of any bilinear form \( B \). The **cannonical** co-basis is given by the Grassmann basis \( \mathbb{V} \).

- We do not distinguish between the action of co-elements on elements and the action of elements on co-elements.

**Examples:**

```maple
restart; bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel[`function`=val -- use online help > ?Bigebra[help]
The evaluation on homogeneous decomposable elements of the same grade:
> EV(Id,Id);
          1
          0, 0, 1, 0, 1
          1
 1

NOTE that differently named symbolic indices are considered to be different! Bigebra is not yet well tested and designed to handle symbolic indices.

On homogeneous decomposable elements of different grades, `EV` yields:
> EV(0,Id),EV(e1we2,0),EV(Id,ei);
          0, 0, 0
          0, 0

Evaluating inhomogeneous elements:
> EV(a*Id+b*e1,Id+e3);
          a
          b
          a + 4 \sin(x)

Use contract to map the evaluation onto adjacent tensor slots.
> contract(&t(e1,e2we3+e3we1,e3we1,e2),2,EV);

\[ e_1 \otimes e_2 \]

> printf("worksheet tok \%f seconds on a 2x 1GHz PIII 1GB RAM machine\n",time()-bench);

worksheet tok 3.836000 seconds on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra:-`\&t`, Bigebra:-`\text{type/tensorpolynom}`, Bigebra:-contract, Bigebra:-pairing, Bigebra help page

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-gantipode - the antipode map for Graßmann Hopf gebra

Calling Sequence:
  t1 := gantipode(t1,i)
  c1 := gantipode(c1)

Parameters:
  • t2 : a tensor polynom
  • c2 : a Clifford polynom

Output:
  • t1 : a tensor polynom
  • c1 : a Clifford polynom

Description:
  • The Graßmann antipode is the convolutive inverse of the identity map. It fulfills the antipode axioms (e.g. Sweedler)
    \[ S(x_(1)) x_(2) = \eta \circ \epsilon(x) = x_(1) S(x_(2)). \]

  • The Graßmann antipode is closely related to the grade involution of the Graßmann algebra, see examples below. This involution
    constitutes a Z_2 grading which is also present in Clifford algebras.

  • The Graßmann antipode can be obtained by direct computation (e.g. using tsolev1) from the unital Graßmann bi-convolution.

Examples:

```plaintext
> restart: bench := time(): with(Clifford): with(Bigebra):

Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]

On a Graßmann basis we compute the antipode map, which shows it to be equivalent to the grade involution, this is not an
accident but can be proved:

> dim_V := 4: bas := cbasis(dim_V);
Sbas := map(gantipode, bas);
Cbas := map(gradeinv, bas);
printf("It is `true` that the two lists Sbas and Cbas containe the same
elements", evalb({0} = {op(zip((i,j)->i-j, Sbas, Cbas))})
);

bas := [
  Id, e1, e2, e3, e4, e1we2, e1we3, e1we4, e2we3, e2we4, e3we4, e1we2we3, e1we2we4, e1we3we4, e2we3we4, e1we2we3we4
]

Cplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type
?cliprod for help.

Sbas := [Id, -e1, -e2, -e3, -e4, e1we2, e1we3, e1we4, e2we3, e2we4, e3we4, -e1we2we3, -e1we2we4, -e1we3we4,
  -e2we3we4, e1we2we3we4]

Cbas := [Id, -e1, -e2, -e3, -e4, e1we2, e1we3, e1we4, e2we3, e2we4, e3we4, -e1we2we3, -e1we2we4, -e1we3we4,
  -e2we3we4, e1we2we3we4]

It is `true` that the two lists Sbas and Cbas containe the same elements

Now we give some examples where the Graßmann antipode acts on slots of tensor products:

> gantipode(&t(e1, e2we3, e4), 2);
gantipode(&t(e1, e2we3, e4), 3);

  \&t(e1, e2we3, e4)
  -&t(e1, e2we3, e4)

On special elements we find:

> gantipode(0), gantipode(&t(e1, 0, e3), 2);

0, 0
```
On inhomogenous elements we find:

```maple
> gantipode(&t(a*Id-b*e1-e1we2,d*e2we3we4,Id+e2we3-4*sin(x)*e1we2),1);
```

```maple
a d &t( Id, e2we3we4, Id ) + a d &t( Id, e2we3we4, e2we3 ) − 4 a d sin(x) &t( Id, e2we3we4, e1we2 )
```

```maple
+ b d &t( e1, e2we3we4, Id ) + b d &t( e1, e2we3we4, e2we3 ) − 4 b d sin(x) &t( e1, e2we3we4, e1we2 )
```

```maple
− d &t( e1we2, e2we3we4, Id ) − d &t( e1we2, e2we3we4, e2we3 ) + 4 d sin(x) &t( e1we2, e2we3we4, e1we2 )
```

```maple
> printf("Worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);
```

Worksheet took 4.927000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra:-`&t`, Bigebra:-`type/tensorpolynom`, Bigebra help page

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-gco_unit - Grassmann co-unit

Calling Sequence:

\[ p_2 := \text{gco_unit}(p_1, i) \]

Parameters:

- \( p_1 \): a tensor polynomial
- \( i \): the \( i \)-th slot to act on

Output:

- \( p_2 \): a tensor polynomial

Description:

- The Grassmann co-unit is essentially the projection onto the identity component in the \( i \)-th slot of a tensor polynomial. It acts like the augmentation linear form.
- It can be shown that Grassmann algebra is an augmented connected algebra, which has important consequences in topology, but also for computational means, see [3].
- The co-unit, denoted by \( \varepsilon \), can be constructed by categorial duality from the axioms of a unit, and fulfills these relations:

\[ (\varepsilon \otimes \text{Id}) \Delta = \text{Id} = (\text{Id} \otimes \varepsilon) \Delta \]

which are the dual relations of a algebra unit. \( \Delta \) is the Grassmann co-product and \( \text{Id} \) the identity mapping.
- Note that the co-unit lowers the rank of a tensor by one and maps Clifford polynomials onto scalars.

Examples:

```maple
> restart: bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra[help]
The co-unit acts on tensors:
> dim_V := 2: bas := cbasis(dim_V):
map(gco_unit,bas);
gco_unit(&t(Id,e1),1);
gco_unit(&t(Id,e1),2);
gco_unit(&t(0,e1),1);
:= bas ,, 
```

```
Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.

[Id, 0, 0, 0]
&t(e1)
0
0
```

Check that it is a co-unit in \( \text{dim} := 3 \):

```maple
> dim_V := 3:
basis := cbasis(dim_V):
x := add(_x[i]*bas[i], i=1..2^dim_V);
LHS := gco_unit(tcollect(&gco(x)), 1);
RHS := gco_unit(tcollect(&gco(x)), 2);
```

This is an arbitrary element, we have to compute the lhs and rhs of the defining equation (*) and have to compare the output with the identity mapping (i.e. the arbitrary element). We have to use tcollect in an intermediate step to expand the output of &gco.
\[ + x_1 \& t(\text{Id}) \]
\[ \text{RHS} := _x e \& t(e1we3) + _x e \& t(e2) + _x e \& t(e1) + _x e \& t(e1we2we3) + _x e \& t(e2we3) + _x e \& t(e1we2) + _x e \& t(e3) \]
\[ + _x e \& t(\text{Id}) \]

Now we check if the equations (*) are true:

```latex
> printf("The left equality sign of (*) is %a", evalb(X-drop_t(LHS)=0));
> printf("The right equality sign of (*) is %s", evalb(X-drop_t(RHS)=0));
```

The left equality sign of (*) is true
The right equality sign of (*) is true

```latex
> printf("Worksheet took %f seconds to compute on a 2x 1GHz 1GB RAM machine",time()-bench);
```
Worksheet took 4.956000 seconds to compute on a 2x 1GHz 1GB RAM machine

See Also: Bigebra:-`&t`, Bigebra:-tcollect, Bigebra:-contract, Bigebra help page

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-gswitch - graded switch of tensor slots

Calling Sequence:
\[ p1 = \text{gswitch}(p2, i) \]

Parameters:
- \( p2 \) : a tensorpolynom which is of not less than rank \( i \) in each factor
- \( i \) : the slot number (first slot from the left is 1) of the pair \( (i, i+1) \) on which the switch acts

Output:
- \( p1 \) : a tensorpolynom

Global variables:
- \(_\text{CLIENV}[\_\text{QDEF}\_\text{PREFACCTOR}]\)

Description:
- Given a tensor polynomial the graded switch switches two adjacent slots in a tensor product. In switching the factors, it takes account of the sign of the permutation. Denote the grade of a homogenous Grassmann element \( A \) by \(|A|\). The graded switch of two homogenous elements is related to the (ungraded) switch as follows:

\[
\tau'(A \&t B) = (-1)^{|A||B|} \tau(A \&t B) = (-1)^{|A||B|} (B \&t A).
\]

The action is extended by linearity to arbitrary inhomogenous elements.
- The graded switch is the natural switch for the Grassmann Hopf algebra. If this switch is used in the crossed products, the co-product becomes an algebra homomorphism while the wedge product becomes a co-algebra homomorphism.
- The switch of an antipodal convolution algebra can be derived [3,7]. It happens to be the graded switch in the case of the Grassmann Hopf algebra.
- The graded switch makes the Grassmann co-algebra graded co-commutative.

Examples:

```plaintext
> restart; bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]
Graded switch/swap of tensor factors:
> &t(e1,e2);
gswitch(%,1);
Claplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.
\( e1 \&t e2 \)
\( -(e2 \&t e1) \)

> &t(Id,Id);
gswitch(%,1);
\( Id \&t Id \)
\( Id \&t Id \)

> &t(e1,e1we2);
gswitch(%,1);
\( e1 \&t e1we2 \)
\( e1we2 \&t e1 \)

> p1 := &t(e1,e2);
gsw := &map(gswitch(p1,1),1,wedge);
aswaa := gantipode(&map(gantipode(gantipode(p1,1),2),1,wedge),1);
\[ p1 := e1 \&t e2 \]
\[ gsw := &t(e1we2) \]
```
The graded switch and the antipode of a Grassmann Hopf algebra are related as follows:

\[ \text{awaa} := \&\text{t}(e1we2we3) \text{awaa} \]

### In dimension 2 the equation \( gsw = \text{awaa} \) is true

\[ \text{awaa} := \&\text{t}(e1we2we3) \text{awaa} \]

### In dimension 3 the equation \( gsw = \text{awaa} \) is true

\[ \text{awaa} := \&\text{t}(e1we2we3) \text{awaa} \]

### In dimension 4 the equation \( gsw = \text{awaa} \) is true

\[ \text{awaa} := \&\text{t}(e1we2we3) \text{awaa} \]
since only in this setting one has identities like $e_1 \& e_2 = e_1 \& w e_2$ etc., i.e. the Clifford and Grassmann bases coincide (since Grassmann reversion one would have to set $B$ to be a diagonal bilinear form (their values do not matter and could even be zero),

> &t(e_1,a*e_2+b*e_2\&w e_3,e_1\&w e_4-sin(x)*e_5);

The graded switch is involutive:

> reversion(reversion(e_1\&w e_2));           ## reversion is involutive !!

Error, (in Bigebra:-gswitch) invalid subscript selector

In dimension 4 the equation `gswaa=aaw` is `true`

However, be aware that the reversion is the Clifford reversion which introduces possibly additional terms! If one thinks about a Grassmann reversion one would have to set $B$ to be a diagonal bilinear form (their values do not matter and could even be zero), since only in this setting one has identities like $e_1 & c e_2 = e_1 & w e_2$ etc. i.e. the Clifford and Grassmann bases coincide (since only off diagonal $B[i,j]$ terms occur in reorderings).
> printf("worksheet to %f seconds on a 2x 1Ghz PIII 1GB RAM machine",time()-bench);

worksheet to 82.169000 seconds on a 2x 1Ghz PIII 1GB RAM machine

See Also: Bigebra:-&t, Bigebra:-switch, Bigebra:-&gco

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002 /BF/RA.
Function: \texttt{Bigebra:-linop} - define a linear operator on $\wedge V$ using a matrix
\texttt{Bigebra:-linop2} - define a linear operator on $\wedge V \times \wedge V$ using a matrix

**Calling Sequence:**
\[ c2 := \text{linop}(c1, \text{name}) \]

**Parameters:**
- $c1$ : a Clifford polynomial
- $\text{name}$ : the kernel name of matrix entries e.g. $R \leftrightarrow (R[i,j])$

**Output:**
- $c2$ : a Clifford polynomial.

**Description:**
- The \texttt{linop} command can be used to define linear operators, i.e. elements of $\text{End} \, \wedge V$, using matrices. The basis assumed is the standard Graßmann basis of \texttt{CLIFFORD}.
- The main purpose of these two functions is to handle operators which need a long computation time, e.g. the antipode of a Clifford Hopf algebra or the switch (needs \texttt{linop2}) of a Clifford Hopf algebra.
- To be able to map an operator one has to define a wrapper function, see below.
- The \texttt{linop2} command is similar to \texttt{linop}, but it acts on the space $\wedge V \times \wedge V$, i.e. it creates operators from $\text{End} \, (\wedge V \times \wedge V)$.

**Examples:**
\begin{verbatim}
> restart; bench := time(): with(Clifford): with(Bigebra):
dim_V := 3:
Increase verbosity by infolevel[’function’]=val -- use online help > ?Bigebra[help]
Let the name of the operator be R, its matrix elements be R[i,j], we have:
> linop(Id,R);
linop(e1,R);
linop(e1we2,R);
Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type
?cliprod for help.
\[
\begin{array}{l}
R_{1,1} = 1d + R_{2,1}x + R_{3,1}x2 + R_{4,1}x3 + R_{5,1}xelwe2 + R_{6,1}xelwe3 + R_{7,1}xelwe3 + R_{8,1}xelwe2we3 \\
R_{1,2} = 1d + R_{2,2}x + R_{3,2}x2 + R_{4,2}x3 + R_{5,2}xelwe2 + R_{6,2}xelwe3 + R_{7,2}xelwe3 + R_{8,2}xelwe2we3 \\
R_{1,5} = 1d + R_{2,5}x + R_{3,5}x2 + R_{4,5}x3 + R_{5,5}xelwe2 + R_{6,5}xelwe3 + R_{7,5}xelwe3 + R_{8,5}xelwe2we3 \\
\end{array}
\]
Define an operator (function) R which can be mapped on tensor slots. Then map it on some examples
\begin{verbatim}
> R := proc(x) linop(x, R) end:
mapop(&t(e1,e2we3), 1, R);
mapop(&t(e1,e2we3), 2, R);
R_{1,1} = 1d + R_{2,1}x + R_{3,1}x2 + R_{4,1}x3 + R_{5,1}xelwe2 + R_{6,1}xelwe3 + R_{7,1}xelwe3 + R_{8,1}xelwe2we3 \\
R_{1,2} = 1d + R_{2,2}x + R_{3,2}x2 + R_{4,2}x3 + R_{5,2}xelwe2 + R_{6,2}xelwe3 + R_{7,2}xelwe3 + R_{8,2}xelwe2we3 \\
R_{1,5} = 1d + R_{2,5}x + R_{3,5}x2 + R_{4,5}x3 + R_{5,5}xelwe2 + R_{6,5}xelwe3 + R_{7,5}xelwe3 + R_{8,5}xelwe2we3 \\
\end{verbatim}
Define the antipode of the Graßmann algebra over $V$ having $\text{dim}_V=2$ via a matrix and apply it to a $2^2$ basis:
\begin{verbatim}
> dim_V := 2:
for i from 1 to 4 do for j from 1 to 4 do
if i<>j then S[i,j] := 0 else if i=2 or i=3 then S[i,j] := -1 else S[i,j] := 1 fi fi
od:od:
matS := linalg[matrix](4,4,(i,j)->S[i,j]);
operS := proc(x) linop(x, S) end:
bas := cbasis(2);
\end{verbatim}
map(eval@operS,bas);
map(gantipode,bas);

$$matS := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

bas := [Id, e1, e2, e1we2]
[Id, -e1, -e2, e1we2]
[Id, -e1, -e2, e1we2]

Such an operator is mapped onto tensor slots using mapop, some examples are:
> mapop(&t(e1,e1we2,e2),2,operS);

$$\&t(e1, e1we2, e2)$$

> mapop(&t(e1,e1we2,e2),3,operS);

$$\&t(e1, e1we2, e2)$$

Linop2 is the counterpart for operators acting on $$V^\wedge \&t V^\wedge$$, in our case the vector space dimension is 2 (dim_V=2) and dim $$V^\wedge = 2^2 = 4$$ so the product is a 16 times 16 matrix:
> bas:=cbasis(dim_V):
GSW:=op2mat2(gswitch,1);
`V^2_bas` := [seq(seq(&t(bas[i],bas[j]),i=1..2^dim_V),j=1..2^dim_V)];
`V^2_GSW_bas` := convert(evalm(GSW &* `V^2_bas`),list);
`V^2_gs_bas` := map(gswitch,`V^2_bas`,1);
printf("Are the two lists equal \?  %a
\n",op({seq(is(`V^2_GSW_bas`[i]=`V^2_gs_bas`[i]),i=1..4^dim_V)}));

> bas:=cbasis(dim_V):
GSW:=op2mat2(gswitch,1);
`V^2_bas` := [seq(seq(&t(bas[i],bas[j]),i=1..2^dim_V),j=1..2^dim_V)];
`V^2_GSW_bas` := convert(evalm(GSW &* `V^2_bas`),list);
`V^2_gs_bas` := map(gswitch,`V^2_bas`,1);
printf("Are the two lists equal \?  %a
\n",op({seq(is(`V^2_GSW_bas`[i]=`V^2_gs_bas`[i]),i=1..4^dim_V)}));

> bas:=cbasis(dim_V):
GSW:=op2mat2(gswitch,1);
`V^2_bas` := [seq(seq(&t(bas[i],bas[j]),i=1..2^dim_V),j=1..2^dim_V)];
`V^2_GSW_bas` := convert(evalm(GSW &* `V^2_bas`),list);
`V^2_gs_bas` := map(gswitch,`V^2_bas`,1);
printf("Are the two lists equal \?  %a
\n",op({seq(is(`V^2_GSW_bas`[i]=`V^2_gs_bas`[i]),i=1..4^dim_V)}));

> bas:=cbasis(dim_V):
GSW:=op2mat2(gswitch,1);
`V^2_bas` := [seq(seq(&t(bas[i],bas[j]),i=1..2^dim_V),j=1..2^dim_V)];
`V^2_GSW_bas` := convert(evalm(GSW &* `V^2_bas`),list);
`V^2_gs_bas` := map(gswitch,`V^2_bas`,1);
printf("Are the two lists equal \?  %a
\n",op({seq(is(`V^2_GSW_bas`[i]=`V^2(gs_bas`[i]),i=1..4^dim_V)}));

This is the matrix representation of the graded switch of the Grassmann Hopf algebra, as can be checked by comparing the two lists of basis vectors. We define, using this matrix, the operator 'gs' and compare some elements with the BIGEBRA built in procedure gswitch.
> gs:=proc(x) linop2(x,GSW) end:
gs(&t(e1,e2)),gswitch(&t(e1,e2),1);
gs(&t(e1we2,e2)),gswitch(&t(e1we2,e2),1);
Using mapop2, we can map 2->2 tensor operators onto two adjacent tensor slots i,i+1 of a tensor.

```plaintext
> mapop2(&t(e1,e2,e3,e4),1,gs);
	-(&t(e2,e1));
> mapop2(&t(e1,e1we2,e2,e1we2),1,gs);
	-(&t(e1,e2,e1we2,e1we2))
> mapop2(&t(e1,e1we2,e2,e1we2),2,gs);
> mapop2(&t(e1,e1we2,e2,e1we2),3,gs);
```

```plaintext
> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);
```

The worksheet took 11.529000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

**NOTE:** If the entries of the tensor polynom are out of the bound of the matrix, this function may go into an **endless loop**! E.g. mapop2(&t(e5,e6),1,gs); in our example, since dim_V was 2.

**See Also:** Bigebra:-mapop, Bigebra:-mapop2, Bigebra:-EV, Bigebra:-pairing, Bigebra:-op2mat, Bigebra:-list2mat2, Bigebra help page

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
**Function:** Bigebra:-lists2mat - derive a matrix of an operator whose action is given as two lists of source and target elements:
\[ \text{op: } x \rightarrow \text{op}(x) = y \]

Bigebra:-lists2mat2 - derive a matrix of an operator from source and target element in \( V^x V^ \rightarrow V^x V^ \)

**Calling Sequence:**
\[
m1 := \text{lists2mat}(\text{list1}, \text{list2})
m1 := \text{lists2mat2}(\text{list1}, \text{list2})
\]

**Parameters:**
- \( \text{list1, list2} \) : two lists of elements related by the action of an operator \( \text{list2}[i] := \text{operator}(\text{list}[i]) \) for all \( i \).

**Output:**
- \( m1 \) : a \( 2^{\dim_V} \times 2^{\dim_V} \) matrix (a \( 4^{\dim_V} \times 4^{\dim_V} \) matrix)

**Global parameters:**
- \( \dim_V \)

**Description:**
- The lists2mat command is useful to derive matrix forms of linear operators. It can be used together with linop (linop2) to move from an operator description to matrix form and back. The derived matrices are regarded as elements of \( \text{End} \ V^x (\text{End} \ V^ \& t V^) \). The basis used is assumed to be the standard Grassmann basis of \text{Clifford} or the following sum:\( \sum_{i,j=1}^{2^{\dim_V}} b[i],b[j] \), where the \( b[i] \) are Grassmann bases of \( V^ \).
- The main purpose of these two functions is to get a matrix form of heavily used operators needed in long computation time, e.g. the antipode of a Clifford Hopf algebra or the switch (may be derived from lists2mat2). The functionality of linalg is then available to those matrices.

**Examples:**
```maple
> restart:
bench := time():
with(Clifford):
with(Bigebra):
dim_V := 2:
Increase verbosity by infolevel[\text{\`function\}]=val -- use online help > ?Bigebra[help]

Let the name of the operator be \( R \), its matrix elements be \( R[i,j] \), we have:
```
> abas:=map(gantipode,sbas,1);
  matS:=lists2mat(sbas,abas);
  map(linop,sbas,matS);
  map(gantipode,sbas,1);

      abas := [Id, -e1, -e2, e1we2]
      matS :=

        [ 1  0  0  0 ]
        [ 0 -1  0  0 ]
        [ 0  0 -1  0 ]
        [ 0  0  0  1 ]
      [Id, -e1, -e2, e1we2]
      [Id, -e1, -e2, e1we2]

We checked using linop that this is the same operator as defined abstratly. Hence our indexing is compatible.

lists2mat2 is the counterpart for operators acting on \( V^\wedge \times V^\wedge \), in our case the vector space dimension is 2 (\( \dim_V=2 \)) and \( V^\wedge = 2^2 = 4 \) so the output is a 16 times 16 matrix:

We check the function by an examples and will see that it is compatible in indexing with linop:

> `V^2_sbas`:=map(gswitch,`V^2_sbas`,1);                  # act using the operator
GSW:=lists2mat2(`V^2_sbas`,`V^2_tbas`);

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

For more examples see \texttt{op2mat}. Which can easily be converted to the lists2mat case.

> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);

The worksheet took 11.037000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

NOTE: If the entries of the tensor polynomial are out of the bound of the matrix, this function may go into an \texttt{endless loop}! E.g. mapop2(&t(e5,e6),1,gs); in our example, since \( \dim_V \) was 2.

See Also: \texttt{Bigebra:-mapop}, \texttt{Bigebra:-mapop2}, \texttt{Bigebra:-EV}, \texttt{Bigebra:-pairing}, \texttt{Bigebra:-linop}, \texttt{Bigebra:-op2mat}, \texttt{Bigebra:-help}

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2001/BF/RA.
Function: Bigebra:-make_BI_Id - initialization of the Clifford co-product

Calling Sequence:

• make_BI_Id();

Parameters:

• none.

Global variables:

• BI_Id - set by make_BI_Id
• dim_V - dimension of B, as described in CLIFFORD (see CLIFFORD_ENV)
• BI - dim_V x dim_V matrix of the Clifford co-scalarproudct w.r.t. a co-basis

Description:

• Like the Clifford product, a Clifford co-product needs a bilinear form to be defined on the base space of the Grassmann algebra. In the case of the co-product this form is tied to co-one-vectors, so it is called co-scalarproudct. Since we deal with finite dimensional spaces, the dimension of the co-vector space is dim_V as for the vector space used by CLIFFORD. Hence we use the global variable dim_V, which has to be assigned (caution: CLIFFORD sets this variable to 9 by default which may result in a very long initialization). The matrix of the Clifford co-product is stored in BI. BI can be assigned freely, without any restrictions or relations to the matrix of the Clifford scalarproduct B. BI can be singular or nonsymmetric or even zero, in which case the Clifford co-product reduces to the Grassmann co-product.

• The make_BI_Id() function is needed to initialize the Clifford co-product which is calculated using the Rota-sausage tangle. In Sweedler notation this reads:

\[ \Delta_{(c)}(x) = (\wedge &t \wedge)(Id &t BI_Id &t Id) \Delta(x) \]

where \( \Delta_{(c)} \) is the Clifford co-product, \( \Delta \) is the Grassmann co-product, BI_Id is a two tensor, BI_Id_(1) &t BI_Id_(2) in Sweedler notation. The function make_BI_Id sets this global variable BI_Id which is essentially the cap tangle in the Rota-sausage tangle. Note, that unlike in the Grassmann case, the Clifford co-product of the identity is not the tensorproduct of two identity elements. That is, the unit of the Clifford product is not Clifford co-algebra homomorphism. Clifford algebras are not connected, see Milnor and Moore.

• BI_Id represents the 'cup' tangle in the co-cliffordization.

• BI_Id is not a tensor polynom but a data structure to speed up internal computations for the clifford co-product. In later versions of BIGEBRA this variable may be incorporated directly into the Clifford co-product and will be dropped in the package.

Examples:

```maple
restart; bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel["function"]=val -- use online help > ?Bigebra[help]
dim_V := 2;
BI := linalg[matrix](dim_V, dim_V, [a, b, c, d]): # co-scalarproudct
BI :=
[ a   b ]
[ c   d ]

> assigned(BI_Id);
false

The Clifford co-product does not yet work

> &cco(\&t(e1,e2),1);
Cliplus has been loaded. Definitions for type/clim and type/clipolynom now include \&C and \&C[K]. Type ?cliprod for help.

\[ BI_Id_1 \times BI_Id_2 \times BI_Id_3 \times (Id, e1, e2) + BI_Id_1 \times BI_Id_2 \times BI_Id_3 \times \&t(e1, Id, e2) \]

> make_BI_Id();

\( (Id \&t Id) + a (e1 \&t e1) + b (e2 \&t e1) + b (e1 \&t e2) + d (e2 \&t e2) + (c b - d a) (e1we2 \&t e1we2) \)

> assigned(BI_Id);
true
```
We can compose from $\text{BI}_\text{Id}$ the cup tangle as a tensor of rank 2. Therefore one has to know, that the datastructure of $\text{BI}_\text{Id}$ is a list of lists, where the inner list contains a tripple of a [prefactor, first tensor slot, second tensor slot]. Adding this up yields:

```maple
> BI_Id;
cup:=add(BI_Id[i][1]*&t(BI_Id[i][2],BI_Id[i][3]),i=1..nops(BI_Id));
```

Let us check that $\text{BI}_\text{Id}$ is the Clifford co-product of $\text{Id}$:

```maple
> &cco(Id);
```

Further examples are:

```maple
 > &cco(e1);
 > &cco(e1we2);
 > bas:=cbasis(dim_V):
  &cco(add(a[i]*bas[i],i=1..2^dim_V));
```

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-mapop  - maps a linear operator from End $V^\wedge$ onto tensor slots
Bigebra:-mapop2  - maps a linear operator from End $(V^\wedge \&t V^\wedge)$ onto two tensor slots

Calling Sequence:
\[ t2 := \text{mapop}(t1,i,\text{End1}) \]
\[ t2 := \text{mapop2}(t1,i,\text{End2}) \]

Parameters:
- \( t1 \): a tensor polynom
- \( i \): the \( i \)-th tensor slot
- \( \text{End1} \): a (linear) operator / an endomorphism $\text{End} \wedge V$
- \( \text{End2} \): a (linear) operator / an endomorphisms $\text{End} \wedge V \&t \wedge V$

Output:
- \( t2 \): is a tensor polynom

Description:
- The mapop device allows to define operators and map them onto a certain place in a tensor polynom. Linear operators can be defined using the linop device. Arbitrary not necessarily linear operators can be defined as functions of one or two tensor slots, however, due to the multilinearity of $\&t$ they will be nearly linear.

Examples:
```
> restart: bench := time(): with(Clifford): with(Bigebra):

> intdim := 3:
> Increase verbosity by infolevel[`function`] = val -- use online help > ?Bigebra[help]
We define a linear operator, connected to the integral of a Graßmann Hopf gebra:

> integral := proc(a1) Id*coeff(a1, &w(seq(cat(e,i), i=1..intdim))) end:
Now we can apply the integral to any tensor we want:

> mapop(&t(e1,e1,e1we2we3),2,integral);
> mapop(&t(e1,e1,e1we2we3),3,integral);

> mapop2(&t(e1,e2+e3,e1+e2,e3),2,same_neighbours);
```

The mapop functionality is effectively combined with the linop and linop2 functions which define linear operators in terms of matrix elements w.r.t. the standard Graßmann basis.
matS :=
\[
\begin{bmatrix}
2 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 2 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 2 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 2 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 2
\end{bmatrix}
\]

S := proc(x) linop(x, matS) end:

\[\]
\[
= S := \text{proc}(x) \ \text{linop}(x, \text{matS}) \ \text{end:}
\]

\[
> \text{tcollect(mapop} &\text{(}\&t(\text{Id}, a*el+b*e2+c*e3, elwe2), 2, S))
\]

\[
= (2 + a + b) \ &t(\text{Id, el, elwe2}) + (a + 2 + b + c) \ &t(\text{Id, e2, elwe2}) + (b + 2 + c) \ &t(\text{Id, e3, elwe2}) + a \ &t(\text{Id, Id, elwe2})
\]

\[
+ c \ &t(\text{Id, elwe2, elwe2})
\]

\[
> \text{printf("Worksheet to } %f \text{ seconds to compute on a 2x 1GHz PIII 1GB RAM machine", time()-bench)};
\]

\[
\text{Worksheet to 4.577000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine}
\]

\[
\]

\[
\text{For a corresponding example using linop2 and mapop2 see the help page for linop.}
\]

\[
\text{See Also: Bigebra:-linop, Bigebra:-tcollect, Bigebra:-contract, Bigebra help page}
\]

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.

Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-`&v` -- the vee (meet) product.
Bigebra:-meet -- the meet product.

Calling Sequence:
\[c3 := \&v(c1,c2)\] [or \(c1 \&v c2\), not recommended]
\[c3 := \text{meet}(c1,c2)\] -- synonym.

Parameters:
- \(c1,c2\) - expressions of `type/clipolynom`

Output:
- \(c3\) - expression of `type/clipolynom`

Global variables:
- \(\text{dim}_V\) - dimension of the vector space \((V,B)\) that is defined in CLIFFORD as a global variable.

Description:
- The pair of operations \(\text{wedge (i.e. join)}\) and \(\text{meet}\) acting on Grassmann multi-vectors make up, together with the duality operator, the Grassmann Cayley algebra. This algebra is of tremendous importance in geometrical applications like robotics, visual perception, camera calibration. However, incidence geometries have their own well developed mathematical theory, see e.g. P. Dembowski, Finite Geometries, Springer Verlag, New York, 1968.

- To avoid confusion we should point out that the notion of a meet is not unique in literature. Let \(A\) be a homogeneous decomposable multivector called an extensor. Every such extensor spans a linear subspace of the space over which it was constructed. The span of \(A\) is called the support of \(A\), denoted as \(\text{supp}\ A\). Meet and join can be defined in set theoretic terms on the support of extensors. Let \(A, B\) denote extensors, one defines:

\[A \cup B := \{x \in V | x \in \text{supp}\ A \text{ or } \text{supp}\ B\}\] i.e. the set theoretic union
\[A \cap B := \{x \in V | x \in \text{supp}\ A \text{ and } \text{supp}\ B\}\] i.e. the set theoretic intersection

The operators \(\cup\) and \(\cap\) are the same as in set theory. Under these operations every set is an idempotent: \(A \cup A = A\) and \(B \cap B = B\). Moreover, one finds \(\cup \circ \cup = \text{Id}\) and \(\cap \circ \cap = \text{Id}\) for these operators. Including the set theoretic operation of taking the complement, \((A -> |A\text{ with }|A \cup |A = \text{whole space, where we have used, in lack of an over bar, the Grassmann notation of a preceeding bar}), this constitutes the structure of an ortho-modular lattice. Boolean logic is based on this construction. The two operations of meet and join are related by de Morgan laws:

\[| (A \cup B) = (|A) \cap (|B)\]
\[| (A \cap B) = (|A) \cup (|B)\]

In terms of logic we have: \(\cup = \text{and, } \cap = \text{or, } | = \text{not}\).

In CLIFFORD and Bigebra packages, the meet and join are defined in the following way:

The wedge product of two extensors \(A\) and \(B\) is an extensor \(C\) which has as support the disjoint union of the supports of \(A\) and \(B\). However, extensors having the same support are isomorphic (interchangable). We define the join to be this wedge operation.

We define the join to be this wedge operation. We define the meet to be this wedge operation. We define the join to be this wedge operation. We define the meet to be this wedge operation. We define the meet to be this wedge operation. We define the meet to be this wedge operation.

The ergänzung is not explicit in [A1], but Graßmann discusses the grade of the complement [A which he calls there 'Ergänzzahlen' (A1 §133)] using the so called 'Ergänzung' (Graßmann A2, §4, No. 89 page 57), which we defined already above as \(\text{acial}\) of an extensor \(A\) to be \(|A|. In analogy to de Morgan laws (which he most likely did not know) as:

\[| (A \cup B) := (|A) \land (|B)\]

Graßmann used no sign for products, having over 16 of them working, many at the same time and their type had to be deduced
by context. He used furthermore no parentheses which makes his writings cumbersome to read. The \( \Lambda \) sign mutated from an (uppercase) Lambda used by Burali-Forti and Marcolongo to be the wedge of Bourbaki.

The usage of the Ergänzung points out clearly that the meet depends on the dimension of the space. We will see below, that this definition of the meet is computationally very ineffective.

Alfred Lotze, (Über eine neue Begründung der regressiven Multiplikation extensiver Größen in einem Hauptgebiet n-ter Stufe, Jahresbericht der DMV, 57:102-110,1955) defined a universal formula for the regressive product of r-factors. He showed that if one considers the n-1 dimensional space as a space of co-vectors then the original wedge product becomes by the same formula the regressive product of the co-vectors, pointing out the fact that a symmetric correlation is needed for this purpose. That is: (n-1)-multi-vectors are not co-vectors, but may be seen as reciprocal vectors. In [4], G.-C. Rota and coworkers gave a definition of the meet in terms of a Peano algebra which is essentially the same construction. However, they used the notion of Hopf algebra which allows to write down the formulas in a comprehensible way.

The Graßmann wedge product has as logical counterpart in the exclusive or \( \text{xor} \), the Ergänzung is not w.r.t. the chosen volume form of the space \( V \) the Graßmann algebra is build over. The meaning of the meet follows from his duality relation.

- In Bigebra, the meet and \( \&v \) (vee) products are implemented as follows (note the order of factors in the bracket!):

\[
\text{meet}(c_1,c_2) := [c_2,1,c_1] c_2,2
\]
\[
\&v(c_1,c_2) := c_1,1 [c_2,c_1,2],
\]

where the bracket \([ , ]\) is a scalar valued alternating multilinear volume form and the co-products are given in Sweedler notation. It can be shown (and is tested below) that both forms represent the same operation.

- The Hopf algebraic definition of the meet gives us a great deal of computational benefits as we will show below in some benchmarks. However it works exactly as the Graßmann regressive product.

- Graßmann introduced the so called stereometric product, which, being context sensitive, switches between the wedge and the \( \&v \) (vee-) product. Using polymorphism this could be implemented, and the user can easily program such a wrapper function. We found it peculiar to implement it using the same notation for basis elements for vectors and co-vectors.

- The meet as defined here is independent of the assigned scalar product \( B \) or the assigend co-scalarproduct \( BI \). In fact it can be shown that the vee-product is \( \text{SL}_n \) invariant. If one is interested in projective geometry, the invariants derived from meet and join are \( \text{GL}_n \) invariants.

- The meet product is related to the notion of a Hopf algebraic integral [3]. As a remarkable fact, in any Clifford Hopf gebra over \( \text{dim} \ V = 2 \) one is not able to find an non zero integral. The notion of meet has thus to be reconsidered in the deformed case.

**Examples:**

```maple
restart; bench := time(): with(Clifford):: with(Bigebra):

# Increase verbosity by infolevel['function']=val -- use online help > ?Bigebra[help]
# Infix form (not recommended, see help page on &t). Note that we have not assigned a scalar- or co-scalarproduct.
> dim_V:=2:
   e1 &v e2;
   e1 &v elwe2, elwe2 &v e2, elwe2 &v elwe2;

Clilplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.

        -Id
      e1, e2, elwe2

Firsts of all let us check that both versions of teh meet compute indeed identical:
> for i from 1 to 4 do
    dim_V:=i:
    bas:=cbasis(dim_V);
    X:=add(_X[i]*bas[i],i=1..2^dim_V):
    Y:=add(_Y[i]*bas[i],i=1..2^dim_V):
```
printf("In dimension %d the equation \`meet(X,Y)=\&v(X,Y)` is
\%a\n",dim_V,evalb(0=simplify(meet(X,Y)-\&v(X,Y))));

In dimension 1 the equation `meet(X,Y)=\&v(X,Y)` is true
In dimension 2 the equation `meet(X,Y)=\&v(X,Y)` is true
In dimension 3 the equation `meet(X,Y)=\&v(X,Y)` is true
In dimension 4 the equation `meet(X,Y)=\&v(X,Y)` is true

The following example will show, that the meet and the join are exterior products on their own right and cannot be distinguished.
This makes it unnecessary to use the \lor (vee) sign for the ordinary wedge product as Rota promoted to stress the analogy with set theoretic operators. We will see that the join of points is the meet of (hyper) planes and the meet of points is the join of (hyper) planes. To demonstrate this, we compute the meet for a Graßmann basis. We check associativity, unit, and show that this product is an exterior product on its own right on reciprocal (sometimes called wrongly dual) vectors (i.e. hyperplanes). The reciprocal meet is then defined to be the meet w.r.t. hyper-planes. Then it is shown that this reciprocal meet is indeed the wedge (join of points) with which we started. [To give a crude reciprocal meet we use Graßmann's Ergänzung, but a combinatorial evaluation is also possible but proved to be too long for this help page.]

We present our demonstration in dimension 3. Define the n-1 (i.e. 2-) vectors \(A(i)\). These multi-vectors are the images of a covector basis under dualization, see [4,3] and should be called reciprocal vectors. Their definition involves a symmetric correlation. The 'meet' or \&v (vee) product of vectors acts as an exterior (or wedge) product on these reciprocal vectors. This is an immediate consequence of categorial duality and is related to the Plücker coordinatization of hyper-planes.

While we show here explicitly how to define a a Meet and Join for hyperplanes, there is a generic Graßmann co-product \&gco_pl in the package which could be used with some benefits for the performance, but would probably obscure out aim here.

> dim_V:=3:

```
# A(i),A(ij) etc are new basis elements
# # ==> define the hyperplane basis A(i), A(ij) etc
A:=proc(x)
  local T;
  T:=table([123=-Id,
            31=-e2, 23=-e1, 12=-e3,
            13= e2, 32= e1, 21= e3,
            3=e1we2, 2=-e1we3, 1=e2we3,
            0=e1we2we3]);
  RETURN(T[x]);
end:
#
# w2A is a translation procedure which turns the output
# into the new A basis of reciprocal vectos (plane vectors)
# # ==> w2A  (wedge basis to hyperplane basis A(i)
w2A:=proc(x)
  local bas,y;
  bas:={Id=\'A(123)\',
        e1=\'A(23)\', e2= \'A(13)\', e3= \'A(12)\',
        elwe2=\'A(3)\',elwe3=\'-A(2)\',e2we3=\'A(1)\',
        elwe2we3=\'A(0)\');
  RETURN(subs(bas,Clifford:-reorder(x)));
end:
#
# \&V (uppercase) is a wrapper function to make the usage of the
# A(i) basis more comfortable
#
# ==> \&V can act on the hyperplane basis A(i) seen as wedge multivectors
#    and yields A(jk) hyperplane 2-vectors
#    [same goes for the Meet (formerly meet)]
`\&V`:=proc(x,y)
After these preliminary definitions we can directly show the meet to be the 'wedge product of hyperplanes'. First of all we check some elementary properties of the meet acting on hyperplanes.

```
> A(0), w2A(A(0)); # The 'scalar' w.r.t. the &v product
A(1), w2A(A(1));
A(2), w2A(A(2));
A(3), w2A(A(3)); # (reciprocal) vectors
```

```
e1we2we3(A0)
e2we3(A1)
-e1we3(A2)
e1we2(A3)
```

```
> &V(A(0), A(1)), &V(A(1), A(0)); # shows A0 to be the identity
Meet(A(0), A(1)), Meet(A(1), A(0)); # synonym but internally computed differently
```

Now we produce reciprocal bi-vectors (bi-hyperplanes to be precise) A(ij) and the volume element A(123)

```
> &V(A(1), A(2)), &V(A(2), A(3)), &V(A(3), A(1)); # BI-HYPERPLANES
&V(A(1), &V(A(2), A(3))), # VOLUME ELEMENT EVALUATES TO -ID
&v(A(1), eval(&V(A(2), A(3)))); # eval is needed here to apply A(23)
```

```
A(12), A(23), -A(13)
A(123), -1Id
```

There are no higher multi-hyperplanes (reciprocal multi-vectors) and the following expressions evaluate to zero:

```
> &V(A(1), A(123)), &V(A(12), A(23));
```

```
0, 0
```

The bracket for co-vectors can be defined using the fact that -1Id is the volume in the space of hyperplanes as the projection onto -1Id. Hence we can define the reciprocal meet RMeet of reciprocal vectors. This is also a demonstration how to extend the features of the CLIFFORD/Bigebra packages:

```
> B := linalg[diag](1$dim_V);     ### internally used for Grassmann Erg"anzung
`&RMeet` := proc(x, y)                       ### function co-meet
local yy, res, lst, var_i, v1, v2;
option `Copyright (c) Ablamowicz, Fauser 2000/02. All rights reserved.`;
## crude version of the Grassmann co-product on the 'multivector plane space'
yy := &t(e1we2we3, &gco(eval(cmul(e3we2we1, y))), e1we2we3);
yy := &map(tcollect(&map(switch(yy, 3), 3, cmul)), 1, cmul);
##
if type(yy, tensorbasmonom) or type(yy, tensormonom) then
    lst := [yy];
else
    lst := [op(yy)];
fi;
res := 0;
for var_i in lst do
    v1, v2 := peek(var_i, 1);
    res := res - scalarpart(&v(eval(x), v1))*drop_t(op(v2));
end;
```
To exemplify our claim, let us define the two mutually reciprocal basis sets of points, joined points (i.e., lines) and point space volume and the hyperplanes bi-hyperplanes (i.e., lines) and the volume of the hyperplane multi-vector space $-1d$.

```plaintext
> bas:=cbasis(3);
basis_A:=[A(0),A(1),A(2),A(3),A(12),A(13),A(23),A(123)];
```

```
bas := [Id, e1, e2, e3, e1we2, e1we3, e2we3, e1we2we3]
basis_A := [e1we2we3, e2we3, -e1we3, elwe2, -e3, -e1, -Id]
```

For easy comparison, we compute the multiplication table of the RMeet product. This multiplication table is a tensor of rank three. To be able to display this tensor as rank two array, we put the resulting multivectors (in Grassmann basis) into the array. The numerical matrices $m_{ij}^k$ are then obtained by setting one basis element to 1 and all other to zero (i.e. by acting with the dual multivectors on this scheme.)

```plaintext
> Mul_tab_RMeet:=linalg[{'matrix'}](2^dim_V,2^dim_V,(i,j)-> 0):
for i from 1 to 2^dim_V do
for j from 1 to 2^dim_V do
Mul_tab_RMeet[i,j]:=reorder(&RMeet(bas[i],bas[j]));
end:
end:
```

```
\[
\begin{bmatrix}
-Id & -1 & -2 & -3 & -1we2 & -1we3 & -2we3 & -1we2we3 \\
-1 & 0 & -1we2 & -1we3 & 0 & 0 & -1we2we3 & 0 \\
-2 & 1we2 & 0 & -2we3 & 0 & 1we2we3 & 0 & 0 \\
-3 & 1we3 & 2we3 & 0 & -1we2we3 & 0 & 0 & 0 \\
-1we2 & 0 & 0 & -1we2we3 & 0 & 0 & 0 & 0 \\
-1we3 & 0 & 1we2we3 & 0 & 0 & 0 & 0 & 0 \\
-e2we3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-e1we2we3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
```

Our final goal is to show, that the above defined multiplication for RMeet (the meet of hyperplanes) is equivalent to the wedge product of points. We compute therefore the multiplication table for the wedge also:

```plaintext
> Mul_tab_wedge:=linalg['matrix'](2^dim_V,2^dim_V,(i,j)-> 0):
for i from 1 to 2^dim_V do
for j from 1 to 2^dim_V do
Mul_tab_wedge[i,j]:=&w(bas[i],bas[j]);
end:
end:
```

```
\[
\begin{bmatrix}
Id & e1 & e2 & e3 & e1we2 & e1we3 & e2we3 & e1we2we3 \\
e1 & 0 & e1we2 & e1we3 & 0 & 0 & e1we2we3 & 0 \\
e2 & -e1we2 & 0 & e2we3 & 0 & -e1we2we3 & 0 & 0 \\
e3 & -e1we3 & -e2we3 & 0 & e1we2we3 & 0 & 0 & 0 \\
e1we2 & 0 & 0 & e1we2we3 & 0 & 0 & 0 & 0 \\
e1we3 & 0 & -e1we2we3 & 0 & 0 & 0 & 0 & 0 \\
e2we3 & e1we2we3 & 0 & 0 & 0 & 0 & 0 & 0 \\
e1we2we3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
```

The final check is to add both matrices which gives zero. This shows that up to a sign (which is irrelevant in projective plane geometry) the products are the same. Or, as operator equation:

\[
\text{RMeet}(x,y) = -\text{wedge}(x,y)
\]

The sign belongs to the fact that in three dimensions we find that the volume element squares to negative identity, which means that we would reach the original wedge after a second turn in our argumentation. However, we resist to demonstrate this explicitly here.
Finally we will provide some BENCHMARKS which shall show how efficient the two alternate definitions of the meet are. One, as adopted recently by Hestenes and followers, is based on the Grassmann's Ergänzung and the other is based on Hopf algebra methods as employed in Bigebra and given by Lotze and Rota.

As a Benchmark we compute 100 times a certain meet (this is not a good idea, since some functions may remember its results, e.g. the wedge product from the CLIFFORD package, but it gives nevertheless a feeling what is going on).

The Hopf algebraic case needs:

```plaintext
> s:=time():
  for i from 1 to 100 do
    &v(e1we2,e2we3);
  od:
  printf("This took us %f seconds",time()-s);
&v(e1we2,e2we3);
```

This took us 45.825000 seconds

Since we compute the Clifford product using a very fast Hopf algebraic function cmulRS, this works out faster. However, we could speed up &v by directly employing the Hopf algebraic routines and avoinging wrapper functions as `peek`. Furthermore we have not computed the inverse of the Ergänzung but introduced simply e3we2we1 which is (e1we2we3)^(-1) in our case.

Now let us go for a non-orthogonal but still symmetric bilinear form (a polar form of a quadratic form or a symmetric correlation) and check what happens there:

```plaintext
> restart:bench:=time():with(Clifford):with(Bigebra):  # reload everything to be fair
  dim_V:=3:B:=linalg[diag](1$dim_V):
  s:=time():
  for i from 1 to 100 do
    cmul(e3we2we1,wedge(cmul(e1we2we3,e1we2),cmul(e1we2we3,e2we3)));
  od:
  printf("This took us %f seconds",time()-s);
  cmul(e3we2we1,wedge(cmul(e1we2we3,e1we2),cmul(e1we2we3,e2we3)));
```

This took us 17.705000 seconds

Since we compute the Clifford product using a very fast Hopf algebraic function cmulRS, this works out faster. However, we could speed up &v by directly employing the Hopf algebraic routines and avoinging wrapper functions as 'peek'. Furthermore we have not computed the inverse of the Ergänzung but introduced simply e3we2we1 which is (e1we2we3)^(-1) in our case.
Let us use an arbitrary bilinear form in 3 dimensions:

\[
B := \begin{bmatrix}
g_{1,1} & g_{1,2} & g_{1,3} \\
g_{1,2} & g_{2,2} & g_{2,3} \\
g_{1,3} & g_{2,3} & g_{3,3}
\end{bmatrix}
\]

We reload once more the package to be fair and compute the meet using the Ergänzung:

\[
\text{cli~collect(cmul(e3we2we1,wedge(cmul(e1we2we3,e1we2),cmul(e1we2we3,e2we3))))};
\]

This shows already a difference (approx. a factor 6, which varies from computation to computation due to garbage collection overhead) which would further increase if the dimension were higher. Thus, the computational efficiency of the meet has been demonstrated.

Moreover, we can go beyond the possibilities of the Ergänzungs method since we can compute the meet in the presence of a non-symmetric bilinear form (which cannot be derived from a quadratic form by polarization) using Hopf algebra methods. Our meet works independently of the assigned bilinear form while the Ergänzungs method needs an orthogonal non-degenerate bilinear form (which is the polar form of the symmetric correlation, i.e. a quadratic form).

Let us use an arbitrary bilinear form in 3 dimensions:

\[
B := \begin{bmatrix}
b_{1,1} & b_{1,2} & b_{1,3} \\
b_{2,1} & b_{2,2} & b_{2,3} \\
b_{3,1} & b_{3,2} & b_{3,3}
\end{bmatrix}
\]

The Hopf algebraic meet remains to be

\[
\&v(e1we2,e2we3);
\]

while the 'meet' computed using the Ergänzung does not even yield a homogeneous multi-vector, but a Clifford polynomial:

\[
\text{cli~collect(simplify(cmul(e3we2we1,wedge(cmul(e1we2we3,e1we2),cmul(e1we2we3,e2we3))))});
\]
See Also: Bigebra:-`&map`, Bigebra-peek, Bigebra-poke, Bigebra-switch
### Function: Bigebra:-op2mat
- derive a matrix of an operator acting in $V^\wedge$ which is possibly unfaithful and not irreducible

### Calling Sequence:
```
m1:= op2mat(fkt)
m1:= op2mat2(fkt)
```

### Parameters:
- `fkt`: an linear operator acting on $V^\wedge$ (or $V^\wedge \&t V^\wedge$)

### Output:
- `m1`: a $2^\dim_V \times 2^\dim_V$ matrix (a $4^\dim_V \times 4^\dim_V$ matrix)

### Global parameters:
- `dim_V`

### Description:
- The `op2mat` command is useful to derive matrix forms of linear operators. It can be used to gather with `linop` (linop2) to move from an operator description to matrix form and back. The derived matrices are regarded as elements of $\text{End } V^\wedge$ ($\text{End } V^\wedge \&t V^\wedge$). The basis used is assumed to be the standard Graßmann basis of `Clifford` or the following sum $\sum_{i,j=1}^{2^\dim_V} \&t(b[i],b[j])$, where the `b[i]` are Grassmann bases of $V^\wedge$.
- The main purpose of these two functions is to get a matrix form of heavily used operators needed in long computation time, e.g., the antipode of a Clifford Hopfgebra or the switch (may be derived from `op2mat2`). The functionality of linalg is then available to those matrices.

### Examples:
```
> restart: bench:=time(): with(Clifford): with(Bigebra):
  
dim_V:=2:
Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]

Let the name of the operator be `R`, its matrix elements be `R[i,j]`, we have:
```
> linop(Id,R);
  linop(e1,R);
  linop(e1we2,R);
```

```
Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type ?cliprod for help.
```
```
\[ R_{1,1} \text{Id} + R_{2,1} e1 + R_{3,1} e2 + R_{4,1} e1we2 \]
```
```
\[ R_{1,2} \text{Id} + R_{2,2} e1 + R_{3,2} e2 + R_{4,2} e1we2 \]
```
```
\[ R_{1,4} \text{Id} + R_{2,4} e1 + R_{3,4} e2 + R_{4,4} e1we2 \]
```
```
Get the matrix form of `linop(x,R)`. First we define an operator `R` from which we derive the matrix form, but we show in a second line that parameters may be passed to `op2mat` which allows us to use `linop` directly:
```
> R:=proc(x) linop(x,R) end: # define a proper operator
  
op2mat(R);
  #
op2mat(linop,R);
  #
```
```
Derive the matrix of the Grassmann antipode, in \(\text{dim}_V=2\) we get a 4x4 matrix:

\[
\text{bas}:=\text{cbasis}(\text{dim}_V):
\]

\[
\text{matS}:=\text{op2mat(antipode,1)};
\]

\[
\text{map(linop,bas,matS)};
\]

\[
\text{map(antipode,bas,1)};
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

[\text{Id,} -e_1, -e_2, e_1 e_2]  
[\text{Id,} -e_1, -e_2, e_1 e_2]

We checked using linop that this is the same operator as defined abstractly. Hence our indexing is compatible.

A few further examples are:

\[
\text{conv_unit}:=\text{proc(x) Id*scalarpart(x) end};
\]

\[
\text{conv_unit} \rightarrow \text{op2mat(conv_unit)};
\]

\[
\text{gr_loop}:=\text{proc(x) drop_t(\text{map(\&gco(x),1,wedge)}) end};
\]

\[
\text{Grassmann loop} \rightarrow \text{op2mat(gr_loop)};
\]

\[
\text{X}:=\text{add(x[i]*bas[i],i=1..2^\text{dim}_V)};
\]

\[
\text{scalar_right_conv}:=\text{proc(x) wedge(x,X) end};
\]

\[
\text{scalar right conv} \rightarrow \text{op2mat(scalar_right_conv)};
\]

\[
\text{scalar_left_conv}:=\text{proc(x) wedge(X,x) end};
\]

\[
\text{scalar left conv} \rightarrow \text{op2mat(scalar_left_conv)};
\]

\[
\text{scalar_coright_conv}:=\text{proc(x) drop_t(\text{contract(\&t(\&gco(x),X),2,EV)) end};
\]

\[
\text{scalar coright conv} \rightarrow \text{op2mat(scalar_coright_conv)};
\]

\[
\text{scalar_coleft_conv}:=\text{proc(x) drop_t(\text{contract(\&t(X,\&gco(x)),1,EV)) end};
\]

\[
\text{scalar coleft conv} \rightarrow \text{op2mat(scalar_coleft_conv)};
\]
Are the two lists $V^2_GSW_{bas}$ and $V^2_gs_{bas2}$ equal? true

Are the two lists $V^2_GSW_{bas}$ and $V^2_gs_{bas1}$ equal? true

\begin{verbatim}
> GSW:=op2mat2(gswitch,1);
'V^2_2_bas':=[seq(seq(`&t`(bas[i],bas[j]),i=1..2^dim_V),j=1..2^dim_V)];
'V^2_GSW_bas':=convert(evalm(GSW * 'V^2_2_bas'),list); # compose using linalg
'V^2_2_gs_bas1':=map(gswitch,'V^2_2_bas',1); # act using the operator
'V^2_2_gs_bas2':=map(linop2, 'V^2_2_bas`,GSW); # act using linop2 and the matrix GSW

print("Are the two lists $V^2_GSW_{bas}$ and $V^2_gs_{bas1}$ equal? \%a \n",
      op({seq(is('V^2_GSW_bas`[i]=`V^2_2_gs_bas1`[i]),i=1..4^dim_V)}));
print("Are the two lists $V^2_GSW_{bas}$ and $V^2_gs_{bas2}$ equal? \%a \n",
      op({seq(is('V^2_GSW_bas`[i]=`V^2_2_gs_bas2`[i]),i=1..4^dim_V)}));

GSW :=
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

V^2_2_bas := [Id &t Id, el &t Id, e2 &t Id, elwe2 &t Id, Id &t el, el &t el, e2 &t el, elwe2 &t el, Id &t e2, el &t e2, e2 &t e2, elwe2 &t e2, Id &t elwe2, e2 &t elwe2, elwe2 &t elwe2]
V^2_GSW_bas := [Id &t Id, Id &t el, Id &t e2, Id &t elwe2, Id &t elwe2, Id &t el, el &t el, el &t e2, el &t elwe2, el &t elwe2, -(el &t el), -(el &t e2), el &t elwe2, e2 &t Id, -(e2 &t el), -(e2 &t e2), e2 &t elwe2, elwe2 &t Id, elwe2 &t el, elwe2 &t e2, elwe2 &t elwe2]
V^2_2_gs_bas1 := [Id &t Id, Id &t el, Id &t e2, Id &t elwe2, el &t el, el &t e2, el &t elwe2, e2 &t Id, -(e2 &t el), -(e2 &t e2), e2 &t elwe2, elwe2 &t Id, elwe2 &t el, elwe2 &t e2, elwe2 &t elwe2]
V^2_2_gs_bas2 := [Id &t Id, Id &t el, Id &t e2, Id &t elwe2, el &t el, el &t e2, el &t elwe2, e2 &t Id, -(e2 &t el), -(e2 &t e2), e2 &t elwe2, elwe2 &t Id, elwe2 &t el, elwe2 &t e2, elwe2 &t elwe2]

Are the two lists $V^2_GSW_{bas}$ and $V^2_2_gs_{bas1}$ equal? true
Are the two lists $V^2_GSW_{bas}$ and $V^2_2_gs_{bas2}$ equal? true
\end{verbatim}

Let us give a further example:

\begin{verbatim}
> BI:=linalg[map](2^dim_V, 2^dim_V, (i, j)->C[i, j]):

\end{verbatim}
\[ + B_{2,1} C_{2,1} (e2 \& t e1) + B_{2,1} C_{1,1} (e1 \& t e1), \&t(0), \&t(0), B_{1,2} C_{1,2} (e1 \& t e2) + B_{1,2} C_{2,2} (e2 \& t e2) + B_{1,2} (Id \& t Id) + B_{1,2} (C_{2,1} C_{1,2} - C_{2,2} C_{1,1}) (elwe2 \& t elwe2) + B_{1,2} C_{2,1} (e1 \& t e1) + B_{1,2} C_{1,1} (el \& t e2) + B_{2,2} C_{2,2} (e2 \& t e2) + B_{2,2} (Id \& t Id) + B_{2,2} (C_{2,1} C_{1,2} - C_{2,2} C_{1,1}) (elwe2 \& t elwe2) + B_{2,2} C_{1,1} (e1 \& t e1) + B_{2,2} C_{1,1} (el \& t e2), \&t(0), \&t(0), \&t(0), \&t(0), (B_{2,1} B_{1,2} - B_{2,2} B_{1,1}) C_{1,2} (el \& t e2) + (B_{2,1} B_{1,2} - B_{2,2} B_{1,1}) C_{1,2} (e1 \& t e2) + (B_{2,1} B_{1,2} - B_{2,2} B_{1,1}) C_{1,1} (el \& t e1) + (B_{2,1} B_{1,2} - B_{2,2} B_{1,1}) C_{1,1} (el \& t e2) \]

> op2mat2(BC);
> [1, 0, 0, 0, 0, C_{1,1}, C_{2,1}, 0, 0, C_{1,2}, C_{2,2}, 0, 0, 0, 0, C_{2,1} C_{1,2} - C_{2,2} C_{1,1}]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [B_{1,1}, 0, 0, 0, 0, B_{1,1} C_{1,1}, B_{1,1} C_{2,1}, 0, 0, B_{1,1} C_{1,2}, C_{1,1}, 0, 0, 0, 0, B_{1,1} (C_{2,1} C_{1,2} - C_{2,2} C_{1,1})]  
  [B_{2,1}, 0, 0, 0, 0, B_{2,1} C_{1,1}, B_{2,1} C_{2,1}, 0, 0, B_{2,1} C_{1,2}, C_{1,1}, 0, 0, 0, 0, B_{2,1} (C_{2,1} C_{1,2} - C_{2,2} C_{1,1})]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [B_{1,2}, 0, 0, 0, 0, B_{1,2} C_{1,1}, B_{1,2} C_{2,1}, 0, 0, B_{1,2} C_{1,2}, C_{1,1}, 0, 0, 0, 0, B_{1,2} (C_{2,1} C_{1,2} - C_{2,2} C_{1,1})]  
  [B_{2,2}, 0, 0, 0, 0, B_{2,2} C_{1,1}, B_{2,2} C_{2,1}, 0, 0, B_{2,2} C_{1,2}, C_{1,1}, 0, 0, 0, 0, B_{2,2} (C_{2,1} C_{1,2} - C_{2,2} C_{1,1})]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  [B_{1,1}, B_{1,2} - B_{2,2} B_{1,1}, 0, 0, 0, 0, (B_{1,1} B_{1,2} - B_{2,2} B_{1,1}) C_{1,1}, (B_{1,1} B_{1,2} - B_{2,2} B_{1,1}) C_{1,2}, 0, 0, 0, 0, (B_{1,1} B_{1,2} - B_{2,2} B_{1,1}) (C_{2,1} C_{1,2} - C_{2,2} C_{1,1})]  

As a last example we give switch and the time the antiptode acting on the switch:

> aasw:=proc(x) switch(tcollect(gantipode(gantipode(x,1),2)),1) end:
  op2mat2(gswitch,1);
  op2mat2(aasw);
  convert(evalm(\%-\%),set);  ## these operators are not identical!

\[ \begin{bmatrix}
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  \end{bmatrix} \]
\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\{-2, 0, 2\}

\[
> \text{printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine", time()-bench);}
\]

The worksheet took 140.985000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

\[
> \text{NOTE: If the entries of the tensor polynom are out of the bound of the matrix, this function may go into an \textsf{endless loop}! E.g. mapop2(\&t(e5,e6),1,gs); in our example, since \text{dim}_V \text{ was 2.}}
\]

\[
> \text{See Also: Bigebra:-mapop, Bigebra:-mapop2, Bigebra:-EV, Bigebra:-pairing, Bigebra:-linop, Bigebra:-list2mat2, Bigebra:-help}
\]

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
**Function:** Bigebra:-pairing - computes pairing w.r.t. an exponentially generated bilinear form $B^\wedge$.

**Calling Sequence:**

- $sc := \text{pairing}(c1,c2,name)$

**Parameters:**

- $c1,c2$ : are Clifford polynoms.
- $name$ : optional name to be used as kernel symbol for the pairing.

**Output:**

- $sc$ : is a scalar.

**Description:**

- The pairing is most often used together with $\text{contract}$ function on tensors. However, it acts generically on two Clifford polynomials.
- The pairing acts w.r.t. the bilinear form $B^\wedge$ which is a global variable used in $\text{CLIFFORD}$.
- Note that the relation between vectors and co-vectors is not fixed. One is therefore free to choose any nondegenerate (or even degenerate) bilinear form to establish this connection. If the usual canonical duality is used, this should be achieved by using $\text{EV}$ rather than the pairing.
- A pairing is called $\textit{exponentially generated}$ if it can be written as exterior exponential of a pairing of the generating vector space: $B^\wedge = \exp(B)$.

**Examples:**

```maple
> restart: bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra[help]
The pairing on homogeneous decomposable elements of the same grade, we use the optional parameter A and Z:
> pairing(Id,Id);
pairing(e1,e2),pairing(e2,e1),pairing(ei,ej);
pairing(elwe2,elwe2),pairing(elwe3,elwe2,A);
pairing(elwe2we3,elwe2we3,Z);
Cliplus has been loaded. Definitions for type/climon and type/clipolynom now include &C and &C[K]. Type
?cliprod for help.

1

$B_{1,2}B_{2,1}B_{i,j}$

$B_{2,1}B_{1,2} - B_{2,2}B_{1,1}, A_{1,1}A_{1,2} - A_{1,2}A_{1,1}$

$Z_{3,1}Z_{2,2}Z_{1,3} - Z_{3,1}Z_{2,3}Z_{1,2} - Z_{3,2}Z_{2,1}Z_{1,3} + Z_{3,2}Z_{2,3}Z_{1,1} + Z_{3,3}Z_{2,1}Z_{1,2} - Z_{3,3}Z_{2,2}Z_{1,1}$

The pairing on homogeneous decomposable elements of different grades:
> pairing(Id,el);
pairing(elwe2,e2),pairing(e2,elwe2);

0

0, 0

The pairing on inhomogeneous elements:
> pairing(a*Id-b*e1-elwe2+d*elwe3we4,Id+elwe3-4*sin(x)*elwe2);

$a + 4 \sin(x) B_{2,1}B_{1,2} - 4 \sin(x) B_{2,2}B_{1,1} - B_{2,2}B_{1,3} + B_{2,3}B_{1,2}$

Use contract to map the pairing onto adjacent tensor slots.
> contract(&t(e1,e2we3,e3we1,e2),2,pairing);

$(B_{3,3}B_{3,1} - B_{3,1}B_{3,3})(el &t e2)$

> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);
The worksheet took 6.379000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine
>
Function: Bigebra:-peek - pick elements from tensor slots.

Calling Sequence:
- \([a,b | S] := \text{peek}(p_1,i)\)

Parameters:
- \(p_1\) : a tensorpolynom which is of rank not less than \(i\) in each factor
- \(i\) : the slot number (first slot is from the left is 1) of the pair \((i,i+1)\) on which the switch acts

Output:
- IF \(p_1\) was homogeneous:
  - \(a\) : the entry of the \(i\)-th slot
  - \(b\) : the remaining tensor
- IF \(p_1\) was an inhomogeneous element:
  - \(S\) : a sequence of pairs \(S[i] = a,b\) of elements as in the homogeneous case.

Description:
- Given a tensor monom or tensor polynomial peek selects the element of the \(i\)-th slot of the tensor product. This function is for internal use, but can be used to form user supplied functions on tensors.
- The output of peek depends on the type of the tensor being processed. On homogeneous tensors, i.e. tensor monoms, peek simply returns a pair (expression sequence) composed of the element in the \(i\)-th tensor slot and a tensor composed from the \(i-1\) slots. On tensorpolynoms peek acts on every term of the sum as described above, and returns a sequence of lists of type clipolynom, tensorpolynom.
- Scalar prefactors are returned with the extracted element \(a\).

Examples:
```plaintext
> restart: bench := time(): with(Clifford): with(Bigebra):
Increase verbosity by infolevel[`function`]=val -- use online help > ?Bigebra[help]

Peek on homogeneous tensors:
> peek(&t(e1,e2),1);

\[ e1, &t(e2) \]

> peek(&t(e1,e2),2);

\[ e2, &t(e1) \]

> peek(&t(e1,e2,e3,e4,e5,e6),4);

\[ e4, &t(e1, e2, e3, e5, e6) \]

Having scalar prefactors:
> peek(&t(a*e1we2,b*e3),1);

\[ a b e1we2, &t(e3) \]

> peek(&t(a*e1we2,b*e3),2);

\[ a b e1we2, &t(e1we2) \]

Peek on inhomogeneous elements:
> peek(&t(a*e1+b*e2we3,c*e2we3+d+e1we4),1);

\[ a c e1, &t(e2we3) \]

> peek(&t(a*e1+b*e2we3,c*e2we3+d+e1we4),2);

\[ a d e1, &t(e1) \]

> [a e1, &t(e1we4)], [a d e1, &t(1)], [a e1, &t(e1we4)], [b c e2we3, &t(e2we3)], [b d e2we3, &t(1)],

\[ b e2we3, &t(e1we4)] \]

> [a c e2we3, &t(e1)], [a d, &t(e1)], [a e1we4, &t(e1)], [b c e2we3, &t(e2we3)], [b d, &t(e2we3)],

\[ b e1we4, &t(e2we3)] \]

> peek(&t(e1,e2)+&t(e3,e4),1);

\[ e1, &t(e2) \]

\[ e3, &t(e4) \]

If the slot \(i\) is not available in a tensor, peek fails!
> peek(&t(e1,e2),3);
```
> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);
> The worksheet took 4.025000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra:-&t, Bigebra:-`type/tensorpolynom`, Bigebra:-poke

(c) Copyright December-16-1999 -- 2003, by Rafał Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002, 2001/BF/RA.
Function: Bigebra:-poke - puts elements into tensor slots.

Calling Sequence:

\[ t1 := \text{poke}(p1,c1,i) \]

Parameters:

- \( p1 \) : a tensorpolynom of rank not less than \( i \) in each factor
- \( c1 \) : a Clifford polynomial
- \( i \) : the slot number (first slot is from the left is 1) where to insert the element, i.e. the tail of elements from the \( i \)th one onwards in moved by 1 to the right and the new element is placed at the \( i \)th slot.

Output:

- \( t1 \) : a tensor

Description:

- Given a tensor monom or tensor polynomial poke puts a Clifford monom or polynomial into the \( i \)-th slot of the tensor product.
  
  This function is for internal use, but can be used to form user supplied functions on tensors.
  
  E.g.:
  
  \[ \text{poke}(&\text{t}(p1,\ldots,pi,\ldots,pn),c,i) = &\text{t}(p1,\ldots,p(i-1),c,pi,\ldots pn) \]
  
  \[ \text{poke}(&\text{t}(p1,\ldots,pn),c,(n+1)) = &\text{t}(p1,\ldots,pn,c) \quad \text{i.e. append } c. \]

- Poke raises the rank of a tensor by one.

Examples:

```
> restart; bench := time(); with(Clifford); with(Bigebra);
Increase verbosity by infolevel[`function`] = val -- use online help > ?Bigebra[help]
Poke into homogeneous tensors:
> poke(&t(e1,e2),e4,1);
\[ &\text{t}(e4, e1, e2) \]

> poke(&t(e1,e2),e4,2);
\[ &\text{t}(e1, e4, e2) \]

> poke(&t(e1,e2),e4,3);  # i.e. behind the last slot !
\[ a^2 x \quad &\text{t}(e4, e1we2, e3) \]
\[ a b x \quad &\text{t}(e1we2, e4, e3) \]
\[ a b x \quad &\text{t}(e1we2, e3, e4) \]

Having scalar prefactors in \( p1 \) and \( c1 \):
> poke(&t(a*e1we2,a*e3),x*e4,1);
\[ a^2 x \quad &\text{t}(e4, e1we2, e3) \]

> poke(&t(a*e1we2,b*e3),x*e4,2);
\[ a b x \quad &\text{t}(e1we2, e4, e3) \]

> poke(&t(a*e1we2,b*e3),x*e4,3);  # i.e. behind the last slot !
\[ a^2 x \quad &\text{t}(e4, e1we2, e3) \]

Poke inhomogeneous Clifford elements into homogeneous tensors:
> poke(&t(a*e2we3,e1we4),x*e1+y*e2,1);
\[ a (x \quad &\text{t}(e1, e2we3, e1we4) + y \quad &\text{t}(e2, e2we3, e1we4)) \]
\[ a (x \quad &\text{t}(e2we3, e1, e1we4) + y \quad &\text{t}(e2we3, e2, e1we4)) \]
\[ a (x \quad &\text{t}(e2we3, e1we4, e1) + y \quad &\text{t}(e2we3, e1we4, e2)) \]

Poke inhomogeneous Clifford elements into inhomogeneous tensors:
> poke(&t(e1,e2)+&t(e3,e4),e5+e6,2);
\[ &\text{t}(e1, e5, e2) + &\text{t}(e1, e6, e2) + &\text{t}(e3, e5, e4) + &\text{t}(e3, e6, e4) \]

If the number of slots is \( i \), you can poke to \( i+1 \) (append a slot) but not to \( i+2 \) which causes an error!
> poke(&t(e2we3,e1we4),e5,3);  # OK, appends
poke(&t(e2we3,e1we4),e5,4); # error

&t(e2we3,e1we4,e5)

Error, (in Bigebra:-poke) invalid subscript selector

> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);
The worksheet took 3.955000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra:-`&t`, Bigebra:-`type/tensorpolynom`, Bigebra:-peek

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-remove_eq - helper function to remove tautologies.

Calling Sequence:
• s2 := map(remove_eq,s2)

Parameters:
• s1 : set of equations.

Output:
• s2 : set of equations free of tautologies

Description:
• Remove_eq(uations) is used e.g. in tsolve1 and may be useful to the user in solving large sets of equations
• Remove_eq is usually mapped to a set of equations to remove tautologies. This is helpful to figure out those equalities which are conditions for some variables.

Examples:
> restart:bench:=time():with(Clifford):with(Bigebra):
Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra[help]

Reducing a set of equations:
> s1:={e1=e1,x[1]=x[2],x[2]=x[2],a=e1*b};
s2:={seq(seq(op([x[i]=x[i+(j mod 2)]]),i=1..5),j=1..10)};

\[
\begin{align*}
\text{s1} & := \{ e1 = e1, x_1 = x_2, x_2 = x_2, a = e1 \times b \} \\
\text{s2} & := \{ x_2 = x_4, x_3 = x_5, x_1 = x_2, x_2 = x_3, x_3 = x_4, x_4 = x_5, x_5 = x_6, x_1 = x_1, x_3 = x_3 \}
\end{align*}
\]

> map(remove_eq,s1);
\[
\{ x_1 = x_2, a = e1 \times b \}
\]

> map(remove_eq,s2);
\[
\{ x_1 = x_2, x_2 = x_3, x_3 = x_4, x_4 = x_5, x_5 = x_6 \}
\]

> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);

The worksheet took 2.303000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra:-help, Bigebra:`tsolve`

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-switch - switch of tensor slots

Calling Sequence:

• p1 = switch(p2,i)

Parameters:

• p2 : a tensorpolynom of rank in each factor not less than i
• i : the slot number (first slot is from the left is 1) of the pair (i,i+1) on which the switch acts.

Output:

• p1 : a tensorpolynom

Description:

• Given a tensor polynomial the switch swaps two adjacent slots in a tensor product. No other action is performed.
• Note that switch generates not signs like gswitch.

Examples:

> restart:bench:=time():with(Clifford):with(Bigebra):
Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra[help]

Switch/swap tensor factors:

> &t(e1,e2);

\[ e1 \&t e2 \]

\[ e2 \&t e1 \]

> &t(e1,a*e2+b*e2we3,e1we4-sin(x)*e5);

\[ a \&t(e1, e2, e1we4) - a \sin(x) \&t(e1, e2, e5) + b \&t(e1, e2we3, e1we4) - b \sin(x) \&t(e1, e2we3, e5) \]

\[ a \&t(e2, e1, e1we4) - a \sin(x) \&t(e2, e1, e5) + b \&t(e2we3, e1, e1we4) - b \sin(x) \&t(e2we3, e1, e5) \]

\[ a \&t(e1, e1we4, e2) - a \sin(x) \&t(e1, e5, e2) + b \&t(e1, e1we4, e2we3) - b \sin(x) \&t(e1, e5, e2we3) \]

If the index is not in the range of the tensor slots, an error occurs, so the user has to account for that.

> switch(&t(e1,e2),3);

Error, (in Bigebra:-switch) invalid subscript selector

> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine",time()-bench);

The worksheet took 3.644000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra:-&t', Bigebra:-gswitch

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:−tcollect - collects coefficients of tensor polynomial

Calling Sequence:

• \( p1 := \text{tcollect}(p2) \)

Parameters:

• \( p2 : \) a tensor polynomial

Output:

• \( p1 : \) a tensor polynomia.

Description:

• The function \( \text{tcollect} \) is used to collect coefficients of \( \text{tensor polynomials} \). \textit{This function is sometimes needed to feed output into other functions of Bigebra.} Moreover it allows for a better comparison between tensor polynomials.

• In later versions of BIGEBRA some functions will automatically \( \text{tcollect} \) their output for convenience. However in this version the user is called to do this for performance reasons.

Examples:

\begin{verbatim}
> restart:bench:=time():with(Clifford):with(Bigebra):
Increase verbosity by infolevel[\text{\textquoteleft}\text{function}\text{\textquoteleft}]]=val -- use online help > ?Bigebra[help]

Examples:

\>

\>

\>

\>

\>

\text{tcollect} \left( a \left( e_1 \wedge e_2 \right) + b \left( e_1 \wedge e_2 \right) \right);
\text{tcollect} \left( x \left( e_1 \wedge e_2 \wedge e_3 \right) + y \left( e_1 \wedge e_2 \wedge e_3 \wedge e_4 \right) \right);
\( a \left( 1 + b \right) \wedge \left( e_1 \wedge e_2 \right) \)
\( (x + y) \left( e_1 \wedge e_2 \wedge e_3 \wedge e_4 \right) \)

\text{tcollect} \text{simply returns \( \text{Clifford polynomials} \) without \( \text{tcollecting} \) them!}

\>

\>

\>

\>

\>

\>

\>

\text{tcollect} \left( e_1 + e_2 \wedge e_3 - 4 + \sin\left( x \right) \left( e_1 \wedge e_2 + e_1 \wedge e_2 \right) \right);

\>

\>

\>

\>

\>

\>

\>

\text{printf\text{\textquoteleft}\text{\textquoteleft}\textit{The worksheet took \%f seconds to compute on a 2x 1GHz PIII 1GB RAM machine}\text{\textquoteleft}\text{\textquoteleft}}} \left( \text{time()} - \text{bench} \right);

\text{The worksheet took 2.423000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine}

\>

\>

\>

\>

\>

\>

\>

\text{See Also: \texttt{Bigebra:−\&t}}, \texttt{Bigebra:−\textquoteleft type\textquoteright\textquoteleft tensorpolynom\textquoteright\texttt{}}, \texttt{Bigebra:−help}

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: `type/tensorbasmonom`  
`type/tensormonom`  
`type/tensorpolynom`  -- new types for tensors

Calling Sequence:

- `b1 = type(p, tensorbasmonom)`
- `b1 = type(p, tensormonom)`
- `b1 = type(p, tensorpolynom)`

Parameters:

- `p` - an algebraic expression of type 'anything'.

Output:

- A boolean value 'true' or 'false'

Description:

- Elements of the tensor algebra share this type, see tensor product.
- The procedure returns 'true' or 'false' depending whether its argument is or is not of one of the types 'tensorbasmonom', 'tensormonom' or 'tensorpolynom'.
- The types are inclusive, i.e. a 'tensorbasmonom' is also a 'tensormonom' which happens to be also a 'tensorpolynom'.
- Types are designed for mostly internal use.
- Note: During initialization of the Bigebra package these types are defined and have been placed in the top-level name space, no long form available/needed/usefull.

Examples:

```maple
> restart:bench:=time():with(Clifford):with(Bigebra):
Increase verbosity by infolevel[function]=val -- use online help > ?Bigebra[help]
> Basmonom1:=&t(e1,e2,e3);
Basmonom2:=&t(e1,e1we2);
Monom1:=exp(I*phi)*&t(e1,e2we3,e4);
Monom2:=-a*&t(e3we4,e1,e2);
Polynom:=&t(Monom1+Monom2+Basmonom2);

Basmonom1 := &t(e1, e2, e3)
Basmonom2 := e1 &t e1we2
Monom1 := e^{(1)} &t(e1, e2we3, e4)
Monom2 := -a &t(e3we4, e1, e2)
Polynom := e^{(1)} &t(e1, e2we3, e4) - a &t(e3we4, e1, e2) + (e1 &t e1we2)

> type(Basmonom1,tensorbasmonom),
type(Basmonom1,tensormonom),
type(Basmonom1,tensorpolynom);
true, true, true

> type(Basmonom2,tensorbasmonom),
type(Basmonom2,tensormonom),
type(Basmonom2,tensorpolynom);
true, true, true

> type(Monom2,tensorbasmonom),
type(Monom2,tensormonom),
type(Monom2,tensorpolynom);

false, true, true

> type(Polynom,tensorbasmonom),
type(Polynom,tensormonom),
```
type(Polynom, tensorpolynom);
false, false, true

However, be careful with the infix form of & (ampersand) operators, see define, tensor product.

\[ \text{type}(a \cdot e_2 \& t b \cdot e_3, \text{tensorbasmonom}), \]
\[ \text{type}(a \cdot e_2 \& t b \cdot e_3, \text{tensormonom}); \]  ## NOTE: infix is buggy, use parentheses!!
\[ \text{eval}(a \cdot e_2 \& t b \cdot e_3); \]  ## second tensor slot is not properly treated, see e3!!
\[ \text{eval}(a \cdot e_1 \& t (b \cdot e_3)), \& t (a \cdot e_2, b \cdot e_3); \]  ## works out correctly.
false, true

\[ ab (e_2 \& t 1) e_3 \]
a \& t b (e_2 \& t e_3), a \& t b (e_2 \& t e_3)

\[ \text{add}(a[i] \& t (\text{seq}(e||j, j=1..i)), i=1..4); \]
type(%, tensorpolynom);
\[ a_1 \& t (e_1) + a_2 \& t (e_2) + a_3 \& t (e_1, e_2, e_3) + a_4 \& t (e_1, e_2, e_3, e_4) \]
true

\[ \text{remove}(\text{type}, a \& t (e_1, e_2), \text{tensorbasmonom}); \]
a

\[ \text{select}(\text{type}, a \& t (e_1, e_2), \text{tensorbasmonom}); \]
e_1 \& t e_2

But be careful about this:

\[ \text{select}(\text{type}, a \& t (e_1, e_2) + b \& t (e_3, e_4), \text{tensorbasmonom}); \]
0

\[ \text{printf}("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine", \text{time}()-\text{bench}); \]
The worksheet took 4.476000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

See Also: Bigebra::'&t', Bigebra::define, Bigebra::help

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
**Function:** Bigebra:-tsolve1 - solves \( n \rightarrow 1 \) tangle equations for endomorphisms

**Calling Sequence:**
- \( \text{lst} := \text{tsolve1}(\text{eq}, \text{vars}, \text{param}) \)

**Parameters:**
- \( \text{eq} \) : an expression which is subjected to be zero
- \( \text{vars} \) : variables to be solved for
- \( \text{param} \) : parameters occurring in \( \text{eq} \) but not to be solved for

**Output:**
- \( \text{lst} \) : a list of solution sets (if any exists, otherwise an empty list)

**Description:**
- The \( n \rightarrow 1 \) tangle solver `tsolve1` is an extension of `clisolve` which solves Clifford polynomial equations to \( n \rightarrow 1 \) tangle equations. \( n \rightarrow 1 \) tangles have \( n \) inputs and one output. But we do not seek for solutions in the input/output space \( \Lambda/V \) but in End \( \Lambda/V \), therefore the name 'tsolve1'.
- A detailed analysis of the problem to solve in \( n \rightarrow 1 \) mappings for endomorphisms shows, that the parameters play the role of 'co-vectors' to get a sufficient amount of equations.
- The tsolve1 facility is most effectively used with `mapop` and `linop` to seek for endomorphisms fulfilling certain \( n \rightarrow 1 \) tangle relations.
- If one is interested in convolution algebras \( n (=1) \rightarrow 1 \) tangle equations are the generic case.
- Be careful to think properly about the variables and parameters!

**Examples:**

```maple
> restart; bench := time():
with(Clifford): with(Bigebra):
dim_V := 2:
Increase verbosity by infolevel[`function`] = val -- use online help > ?Bigebra[help]

Example 1: We will show, how to find the convolutive unit of a Grassmann Hopf algebra using tsolve1. The tangle equation is

\[(*) \quad \text{wedge} (F \&t U) \Delta(x) = F(x) = \text{wedge} (U \&t F) \Delta(x)\]

for arbitrary \( F \). The problem is to find the operator \( U \). We compute in \( \text{dimV}= 2, \text{dim} \Lambda/V = 2^2=4 \), hence \( U, F \) can be represented by 4 times 4 matrices.

> matU := linalg[ matrix](2^dim_V, 2^dim_V, (i,j) -> U[i,j]);
> matF := linalg[ matrix](2^dim_V, 2^dim_V, (i,j) -> F[i,j]);
> U := proc(x) linop(x, U) end:
> F := proc(x) linop(x, F) end:
> bas := cbasis(dim_V):
> X := add(_x[i] * bas[i], i = 1 .. 2^dim_V):
```

![matrix equations and calculations]

\[matU := \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} & U_{1,4} \\
U_{2,1} & U_{2,2} & U_{2,3} & U_{2,4} \\
U_{3,1} & U_{3,2} & U_{3,3} & U_{3,4} \\
U_{4,1} & U_{4,2} & U_{4,3} & U_{4,4}\end{bmatrix}\]

\[matF := \begin{bmatrix} F_{1,1} & F_{1,2} & F_{1,3} & F_{1,4} \\
F_{2,1} & F_{2,2} & F_{2,3} & F_{2,4} \\
F_{3,1} & F_{3,2} & F_{3,3} & F_{3,4} \\
F_{4,1} & F_{4,2} & F_{4,3} & F_{4,4}\end{bmatrix}\]

\[X := _x_1 \text{Id} + _x_2 e1 + _x_3 e2 + _x_4 e1we2\]
where $U$ is the convolutive unit as computed above and we have to solve for $S$.

**Example 2:** We compute the antipode of a Graßmann Hopf algebra over $\Lambda V$, dim $V$=2. This allows to ask for the antipode of this algebra.

The second equality is true.

**Example 1:** We compute the antipode of a Graßmann Hopf algebra over $\Lambda V$, dim $V$=2 (continuing example 1). The antipode axioms read

\[
(**) \quad \text{wedge} (S \& \text{Id}) \Delta (x) = U(x) = \text{wedge}(\text{Id} \& S) \Delta (x)
\]

where $U$ is the convolutive unit as computed above and we have to solve for $S$.  

[This is the end of the document content.]
First we define and compute the LHS and middle term, but suppress the output for brevity:

```maple
> matS:=linalg[&matrix](4,4,(i,j)->S[i,j]):
S:=proc(x) linop(x,S) end:
exp4:=clicollect(drop_t(&map(tcollect(mapop(tcollect(&gco(X),1,S)),1,wedge))):
exp5:=linop(X,matU):
```

# Note that U = \( \eta \epsilon \) in Hopf algebraic terms, i.e. display_id@scalarpart in terms of CLIFFORD
```maple
exp5
```

And solve for the antipode S in terms of its matrix representation matS:

```maple
> sol2:=tsolve1(exp4-exp5,[seq(seq(S[i,j],i=1..4),j=1..4)],[seq(_x[i],i=1..4))):
matS=subs(sol2[1],evalm(matS):
```

```maple
sol2
```

```maple
```

Note that the Graßmann Hopf antpode is exactly the grade involution on \( \wedge V \) (if \( \text{dim} V = 2 \), but this can be proved algebraically for arbitrary \( \text{dim} V \)):

```maple
> map(gradeinv,bas);
```

```maple
subs(sol2[1],map(S,bas));
```

Once more we should test that the second equality in (***) is fulfilled, which might be done by the reader!

**Example 3:**

We want to exemplify the tsolve1 facility to prove that in a Clifford Hopf gebra (over \( \text{dim} V = 2 \)) there exists no right/left integrals!

The definition of an integral \( h \) in a Hopf algebra is as follows:

```maple
(***) \quad (\text{Id} \& t h) \Delta (x) = \eta h (x)
```

where \( \eta \) is the algebra unit. Note that \( \Delta \) is the Clifford co-product here. Which we have to initialize using the co-scalarproduct BI.

```maple
> BI:=linalg[&matrix](2,2,[u,z,t,v]);
unprotect(`type/clipolynom`):
dim_V;make_BI_Id():
:= BI
```

First, we define \( h \) as a 'co-vector' (\textit{NOTE}: we use the same symbols for the co-vector basis elements!). Then we compute now the LHS on (***) as follows:

```maple
> h:=add(_h[i]*bas[i],i=1..4);
```

```maple
exp6:=clicollect(simplify(drop_t(contract(&t(&cco(X,1),h),2,EV))));
```

```maple
exp6 + + + _x_2 _h_1 _x_4 _h_3 _x_4 _h_4 _x_1 _h_2
exp6 + + + _x_3 _h_2 + _x_3 _h_1 - _x_2 _v _h_4 - _x_4 _h_2 + _x_1 _t _h_2 + _x_1 _v _h_3 _e_2
exp6 + + + _x_3 _h_2 + _x_3 _h_1 - _x_2 _u _h_2 + _x_1 _z _h_3 - _x_2 _z _h_3 + _x_1 _u _h_2)
```

The RHS of (**) is computed as:

```maple
> exp7:=displayid(contract(&t(X,h),1,EV));
```

```maple
exp7 + + + _x_1 _h_1 + _x_2 _h_2 + _x_3 _h_3 + _x_4 _h_4)
```

And we can solve for \( h \) using tsolve1 (Note that we have to add the parameters of the co-scalarproduct BI which are present in
expr6 and expr7.)

> sol3 := tsolve1(exp6-exp7, [seq(_h[i], i=1..4)], [seq(_x[i], i=1..4), u, z, t, v]);

\[
\begin{align*}
_\text{sol3} & := \{ _{-h_4} = 0, _{-h_3} = 0, _{-h_2} = 0, _{-h_1} = 0 \} \\
\end{align*}
\]

Hence this shows, that there are no non zero integrals in a Clifford convolution for arbitrary co-scalar product. However, we can ask, if there are co-scalarpods which allow an integral to exits. In fact we know that a Graßmann Hopf algebra has a non zero left and right integral. To aswer this question, we have to put parameters of the co-scalarpod into the variables and not the parameters of tsolve1:

> sol4 := tsolve1(exp6-exp7, [seq(_h[i], i=1..4), u, z, t, v], [seq(_x[i], i=1..4)]);

\[
\begin{align*}
\text{select_sol} := () \rightarrow \text{if } 1 = \text{nops}((\text{select}(\text{has}, \text{map}((\text{evalb}, [\text{op(sol4[1])}]))), \text{true})) \text{ then } \\
\text{new_sol} := \text{select_sol}() \\
\text{sol4 := } \{ _{-h_4} = 0, _{-h_3} = 0, _{-h_2} = 0, _{-h_1} = 0, \text{t} = \text{t}, u = u, v = v, z = z \}, \{ _{-h_3} = 0, _{-h_2} = 0, _{-h_1} = 0, u = u, v = v, z = z, \text{t} = 0, _{-h_4} = _{-h_4} \} \\
\text{new_sol} := \{ _{-h_3} = 0, _{-h_2} = 0, _{-h_1} = 0, u = u, v = v, z = z, \text{t} = 0, _{-h_4} = _{-h_4} \} \\
\end{align*}
\]

One of these solutions (since Maple arranges solutions at random every time the worksheet is executed we had to pic the right one) is that of sol3, but we found a second, called new_sol, which provides a non-trivial integral, however for a non zero integral to exist the co-scalarpod, assigned to BI, has to vanish identically!

If we set \( h(x) = \text{int}_H[x] \), its action is given in the next expression, the co-scalarpod vanishes.

> int_H[x] := contract(&t(X, subs(new_sol, h)), 1, EV);

\[
\begin{align*}
\text{int}_H \ := \ _{-x_4} \_h_4 \\
\text{int}_B \ := \ \begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\end{align*}
\]

Note that this is closely related to the bracket which we needed to define the meet \((&v \text{ vee product})\). However the integral is a linear form (multi-co-vector) and obtains this result in a much clearer way. Hence the integral \( h(X) \) of \( X \) can be computed as follows:

> bracket(X) * bracket(h);

\[
\begin{align*}
_{-x_4} \_h_4 \\
\end{align*}
\]

> printf("The worksheet took %f seconds to compute on a 2x 1GHz PIII 1GB RAM machine", time()-bench);

The worksheet took 56.591000 seconds to compute on a 2x 1GHz PIII 1GB RAM machine

> If the integral is normalized to 1, i.e. \( _h[4] = 1 \), then the bracket is exactly the value of the integral. In a projective setting the normalization is not needed and only \( _h[4] \leftrightarrow 0 \) has to be asserted.

See Also: Bigebra:-linop, Bigebra:-tcollect, Bigebra:-contract, Bigebra:-\&map, Bigebra:-EV, Bigebra:-\&gco, Bigebra:-\&cco, Bigebra help page, Clifford intro

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002/BF/RA.
Function: Bigebra:-VERSION - prints informations about Bigebra and the version

Calling Sequence:
VERSION()

Parameters:
• none.

Output:
• none.

Description:
• VERSION() displays informations about the Bigebra package

Examples:
> restart:with(Bigebra):
> Increase verbosity by infolevel['function']=val -- use online help > ?Bigebra[help]
VERSION is a function hence parentheses are needed after the name!
> VERSION();
°°Bi-Gebra Package VERSION 1.01 for Clifford version 6°°
by Rafal Ablamowicz($) and Bertfried Fauser(*)
(c) Dec-16-99 / Oct-22-2002
Available from http://math.tntech.edu/rafal/

($) Department of Mathematics, Box 5054
Tennessee Technological University
Cookeville, TN 38505, U.S.A.
Email: rablamowicz@tntech.edu
URL: http://math.tntech.edu/rafal/

(*) Universit"at Konstanz
Fachbereich Physik, Fach M678
78457 Konstanz, Germany
Email: Bertfried.Fauser@uni-konstanz.de
URL: http://clifford.physik.uni-konstanz.de/~fauser/

Online help available with:
> ?Bigebra
or use 'help' menue and search for topics

Copyright (c) Rafal Ablamowicz, Bertfried Fauser 1999-2003.
All rights reserved. See also > ?Bigebra[help]

BUG-REPORTS to Bertfried Fauser

See Also: Bigebra helppage

(c) Copyright December-16-1999 -- 2003, by Rafal Ablamowicz & Bertfried Fauser, all rights reserved.
Last modified: November 3, 2002 /BF/RA.