CREP Manual
Part III

Peter Dräxler
Rainer Nörenberg
(editors)

Contents

Introduction

Within this third volume of the manual for the CREP system for representation theory of finite dimensional algebras two trends may be observed regarding the development of the system.

The first is an increase in interactive graphical aids for input and display of combinatorial data, especially quivers. This is in particular the case for the `get_alg` interface, which originally a part of a program on Hochschild-Cohomology of algebras by Michael Seibt was adapted to the use with CREP by Dietmar Guhe . Also the elaborate interface of the functions of the `xpre` family by Amrey Krause constructing preprojective components of Auslander-Reiten quivers will not go unnoticed. These functions implement an algorithm formulated by C. M. Ringel [5], using a bound by S. Kasjan and J. A. de la Peña [3].

Possibly more important if less spectacular is a gradual widening of the scope of algebras functions of CREP actually can do work with. The most prominent example here is the `xpre` family, whose basic function accepts rather general input.

Still, development also goes along the more traditional lines of CREP. Here we are especially pleased to be able to present with the `uforms` package by Roman Lychman a well composed assortment of functions for work with unit forms.

Our thanks go to the authors, who have thus contributed to the enrichment of the system.

References

[1] P. Dräxler, R. Nörenberg, *The Crep Manual*, Ergänzungsreihe 96-002, Sonderforschungsbereich 343, Diskrete Strukturen in der Mathematik, Universität Bielefeld, 1996

[2] P. Dräxler, R. Nörenberg, *The Crep Manual, Part II*, Ergänzungsreihe 97-009, Sonderforschungsbereich 343, Diskrete Strukturen in der Mathematik, Universität Bielefeld, 1997

[3] J. A. de la Peña, S. Kasjan, *Constructing the preprojective components of an algebra,* J. Algebra 179 (1996), no. 3, 793–807

[4] R. Nörenberg, *Covering Monomial Algebras,* Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation, ACM Press 1998, 161 – 164

[5] C. M. Ringel, *Constructing All Reachable Modules: An Algorithm,* Bielefeld, 1997

[6] *Quadratic Forms in the Representation Theory of Finite-Dimensional Algebras,* Ergänzungsreihe 96-001, Sonderforschungsbereich 343, Diskrete Strukturen in der Mathematik, Universität Bielefeld, 1996

Crep can be obtained via `ftp` from the server
`ftp.uni-bielefeld.de`
where it can be found in the directory
`pub/math/f-d-alg/crep`

Running Crep will require the Maple system, preferably MapleV release 3 or later.
For installation of some packages of Crep, a Pascal compiler or a C/C++ compiler will be needed. In particular, the `uforms` package will require Pascal, the `get_alg` interface and the

`xpre` family require C++, and the additional functions of the main package will require both Pascal and C for a complete installation.

Questions or remarks on Crep, may be directed at

`fdowner@mathematik.uni-bielefeld.de`

or

Peter Dräxler,
Fakultät für Mathematik,
Universität Bielefeld,
Postfach 100 131,
D-33501 Bielefeld,
e-mail: `draexler@mathematik.uni-bielefeld.de`.

1 Additions to the main package

Amrey Krause (Bielefeld)
Rainer Nörenberg (Bielefeld)

1.1 drw_heralg - draw a hereditary algebra

Calling Sequence:
`drw_heralg(`Q`)`
Parameters:
Q – a quiver given by a quiverdata structure
Synopsis:

- drw_heralg uses Maple plot output to draw the quiver of the hereditary algebra given by the quiverdata structure Q. This will work only for finite-dimensional hereditary algebras, for which the quiver is directed.
- Multiple arrows between a pair of points are represented by a single arrow with the actual number of arrows printed next to it. In case this number is an integer less or equal to 50, it is printed in roman numerals. Otherwise (especially if it is a symbolic expression) the number is printed in brackets.
- Although meant for use with the CREP-package heralg for hereditary algebras, the function drw_heralg uses some parts of the function drwpst of the main package of CREP. Therefore, unlike the functions of the heralg package themselves, the function drw_heralg cannot be used without the main package, which it is a part of.

Examples:

```
> drw_heralg([4,[[1,2,a],[2,3,4],[1,4,200],[4,3,1]]]);
```

SEE ALSO: `heralg, heralg/quiverdata`

1.2 coverz - covering a zero relation algebra

Calling Sequence:
`coverz(`z`)`
`coverz(`z,c`)`
`coverz(`z,c,v`)`
`coverz(`z,c,v,m`)`
Parameters:
z – a zero relation algebra
c,v,m – optional arguments evaluating to names
Synopsis:

- The function `coverz` constructs a finite part of the universal covering of a zero relation algebra, large enough to decide about finite or infinite representation type.
- A zero relation algebra z is given in terms of a quiver and relations by a triple

$$[n, a, r]$$

Here:

- n is the number of vertices of the quiver of z. The vertices are then identified with the numbers $1, \ldots, n$.
- a is a list
$$a = [a_1, \ldots, a_u]$$
containing the arrows of the quiver of z, each arrow a_i being a triple of integers
$$a_i = [s, t, h]$$
where s is the start vertex of the arrow a_i, t its terminal vertex and h a label assigned to it. Different arrows may have the same label.
- r a list
$$r = [r_1, \ldots, r_v]$$
describing the relations imposed in the following way:
Each list entry r_i is a list
$$r_i = [l_1, \ldots, l_w]$$
of labels indicating that any composition $p_1 \ldots p_w$ of arrows with the arrow p_j having label l_j is zero within z.

- The universal covering of a zero relation algebra is given by a (usually infinite) tree with relations. The function returns a finite part of this, in the format of a bound poset algebra (see posetalgebra).

- The finite part of the covering is constructed in the following way:

  The algorithm starts with a spanning tree of the original quiver. To this spanning tree, in a width first fashion, further vertices are added, unless one of the following tree situations is encountered:

  A vertex j will receive no further adjacent vertices if

  1. There is no nonzero path from the vertex j to the initial spanning tree.
  2. The vertex is found to lie on an infinite chain sequence.
  3. The algebra is found not to be finite dimensional.

- Information on the constructed part of the covering is passed via the optional arguments of the function:

  - The optional second argument, if given, will be assigned a list of integers
$$c = [c[1], ..., c[N]]$$
where N is the number of vertices of the part of the covering constructed indicating that the vertex i of the covering lies over the vertex $c[i]$ of the original algebra.
  - The optional third argument, if given, will be assigned a list of integers
$$v = [v[1], ..., v[N]]$$
that for each vertex i contains its state with respect to the algorithm sketched above in encoded form. The explicit meaning of the entries is given by the following table:

$v[i]$	Status of the vertex i
-1	The vertex belongs to the original spanning tree. In this case, all of its neighbours in the universal covering are contained in the part constructed.
-2	All of the neighbours of the vertex in the universal covering are contained in the part constructed.
2	There is no nonzero walk from the vertex in question to the initial spanning tree. There might be neighbours in the universal covering not contained in the constructed part.
3	The vertex lies on an infinite chain. There might be neighbours in the universal covering not contained in the constructed part.
4	Processing this vertex, the original algebra has been found not to be finite dimensional. Processing has been aborted.

- The optional forth argument, if given, will be assigned the maximum of the event codes assigned to the optional third argument v. In particular, a value of 3 means, that the original algebra is not representation finite. A value of 2 means, that the original algebra is representation finite if and only if the constructed part of the covering is. (In fact, the covering is large enough to contain for every vertex of the original algebra a covering vertex, for which the poset of minimal chain sequences (cf rstartsets) can completely be computed.)

Examples:

```
>  z:=[4,[[1,2,1],[2,3,1],[3,4,1],[2,2,2]],[[2,2],[1,1,1]]];
  z := [4, [[1, 2, 1], [2, 3, 1], [3, 4, 1], [2, 2, 2]], [[2, 2], [1, 1, 1]]]

> coverz(z);
[14, [[1], [2, 1, 6], [3, 2], [4, 3], [5, 9, 2], [6, 11, 12], [7, 5], [8, 5],

    [9], [10, 6], [11], [12], [13, 7], [14, 10]],

    [[2, 12], [4, 1], [5, 6], [8, 2], [13, 9], [14, 11]]]

> drwpst(");

> p:=coverz(z,'c','v','m');
p := [14, [[1], [2, 1, 6], [3, 2], [4, 3], [5, 9, 2], [6, 11, 12], [7, 5],

    [8, 5], [9], [10, 6], [11], [12], [13, 7], [14, 10]],

    [[2, 12], [4, 1], [5, 6], [8, 2], [13, 9], [14, 11]]]

> print(c);
                  [1, 2, 3, 4, 2, 2, 3, 2, 1, 3, 1, 2, 4, 4]

> print(v);
            [-1, -1, -1, -1, -2, -2, -2, 2, -2, -2, -2, 2, -2, -2]
```

```
> print(m);
                                        2

> drwpst(p,c);

> drwpst(p,v);

> mchrepfin(p);
                                      false

>#
> z:=[2,[[1,2,1],[1,2,2]]];
                          z := [2, [[1, 2, 1], [1, 2, 2]]]

> p:=coverz(z,'c','v','m');
                   p := [4, [[1], [2, 1, 4], [3, 1], [4]], []]

> m;
                                        3

>#
> z:=[2,[[1,1,1],[1,2,2]],[[1,2]]];
                   z := [2, [[1, 1, 1], [1, 2, 2]], [[1, 2]]]

> p:=coverz(z,'c','v','m');
   p := [6, [[1, 4], [2, 1], [3, 1], [4], [5, 3], [6, 3]], [[2, 4], [6, 1]]]

> m;
                                        4
```

SEE ALSO: posetalgebra, mchrepfin, rstartsets

2 The uforms package

Roman Lychman (Kiev)

The uforms package contains functions dealing with unit forms. It can be used only together with the CREP system. After starting CREP, reading the file `uforms.src` will make the functions of the package available.

2.1 uforms - Unit forms library

Calling Sequence:

`<function>`(*args*)

Synopsis:

- The uforms package provides a set of functions for working with unit forms.
- The available functions are:

uf_init	uf_roots	uf_gabrielov	uf_addvertex
uf_type	uf_sinroots	uf_swelling	uf_editedge
uf_dynkin	uf_critvectors	uf_restriction	
uf_corank			

- For more information on a particular function see `<function>`.

SEE ALSO: `uforms/unit_form`

2.2 uforms/unit_form

Synopsis:

- A unit form q is a map

$$\mathbb{Z}^n \longrightarrow \mathbb{Z}, \quad x \mapsto \sum_{i=1}^{n} x_i^2 + \sum_{i<j} q_{i,j} x_i x_j$$

  where $\mathbb{Z}$ is the ring of integers and $q_{i,j}$ are integers.

- A unit form q is represented by its symmetric $(n \times n)$ matrix Q with coefficients $Q_{i,i} = 2$ for $i = 1 \ldots n$, $Q_{i,j} = Q_{j,i} = q_{i,j}$ for $1 \leq i < j \leq n$.

  It follows that $q(x) = \frac{1}{2} x Q x^T$.

- We can also treat a unit form as a polynomial in n variables.
- There is a well-known bijection between the set of unit forms in n variables and the set of bigraphs on the vertex set 1,..,n (possibly with multiple edges, but without loops) defined as follows: For $i \neq j$ the vertex i is connected with vertex j with $-q_{i,j}$ solid edges, if $q_{i,j} < 0$, and with $q_{i,j}$ dotted edges, if $q_{i,j} > 0$.

  And of course for each bigraph (without loops) with numerated vertices we have a corresponding unit form in an inverse way, setting $a_{i,i} := 2, i = 1 \ldots n$, $a_{j,i} := a_{i,j}, 1 \leq i < j \leq n$.

- For simplicity of description we will in several cases treat a unit form as a corresponding bigraph.

2.3 uf_init - initialize the matrix of a unit form

Calling Sequence:
`uf_init(`v`)`
Parameters:
v – bigraph of the unit form
Synopsis:

- uf_init builds the matrix of the unit form defined by the corresponding bigraph:

$$v = [n, [i_1, j_1, k_1], [i_2, j_2, k_2], \ldots]$$

  where n is the number of vertices, k_l defines the edges between vertices i_l and j_l in the following way: $-k_l$ solid edges provided $k_l < 0$, k_l dotted edges provided $k_l > 0$. All the coefficients n, i_l, j_l, k_l must be integers.

- If you omit k_l then it is supposed that $k_l = -1$.

Examples:

```
> q:=uf_init([5]);

                              [ 2  0  0  0  0 ]
                              [               ]
                              [ 0  2  0  0  0 ]
                              [               ]
                         q := [ 0  0  2  0  0 ]
                              [               ]
                              [ 0  0  0  2  0 ]
                              [               ]
                              [ 0  0  0  0  2 ]

> q:=uf_init([3,[1,2],[2,3,1],[3,1,-2]]);

                              [  2  -1  -2 ]
                              [            ]
                         q := [ -1   2   1 ]
                              [            ]
                              [ -2   1   2 ]
```

SEE ALSO: `uforms/unit_form`

2.4 uf_type - determine the type of a unit form

Calling Sequence:
`uf_type(`q`)`
`uf_type(`q, 'f'`)`
Parameters:
q – matrix of the unit form
'f' – optional parameter which tells the function to perform fast determination of the type of the unit form, without checking if it is positive or nonnegative defined

Synopsis:

- uf_type determines the type of a unit form.

  Available types are:

  - weakly positive,
  - sincere weakly positive,
  - weakly nonnegative,
  - sincere weakly nonnegative,
  - critical.

- Also the function checks if the form is positive or nonnegative defined (see "**Parameters**").

Examples:

```
> q:=uf_init([6,[1,2],[2,3],[3,4],[4,5],[6,3]]);

                          [ 2  -1   0   0   0   0 ]
                          [                       ]
                          [-1   2  -1   0   0   0 ]
                          [                       ]
                          [ 0  -1   2  -1   0  -1 ]
                     q := [                       ]
                          [ 0   0  -1   2  -1   0 ]
                          [                       ]
                          [ 0   0   0  -1   2   0 ]
                          [                       ]
                          [ 0   0  -1   0   0   2 ]

> uf_type(q);
               Positive defined, sincere weakly positive form.

> q:=uf_init([4,[1,2],[2,3],[3,4],[4,1]]);

                          [ 2  -1   0  -1 ]
                          [               ]
                          [-1   2  -1   0 ]
                     q := [               ]
                          [ 0  -1   2  -1 ]
                          [               ]
                          [-1   0  -1   2 ]

> uf_type(q);
                   Nonnegative defined, critical form.
```

SEE ALSO: `uforms/unit_form`

2.5 uf_dynkin - determine the (ext.) Dynkin diagram corresponding to a unit form

Calling Sequence:
uf_dynkin(q)
Parameters:
q – matrix of the unit form
Synopsis:

- uf_dynkin finds the Dynkin or extended Dynkin diagram corresponding to a unit form, by means of Gabrielov transformations.
- The function works only with unit forms which are positive or nonnegative defined.

Examples:

```
> q:=uf_init([4,[1,2],[2,3],[3,4],[4,1]]);

                         [ 2  -1   0  -1 ]
                         [               ]
                         [-1   2  -1   0 ]
                    q := [               ]
                         [ 0  -1   2  -1 ]
                         [               ]
                         [-1   0  -1   2 ]
> uf_dynkin(q);
                           Extended A3

> q:=uf_init([6,[1,2],[2,3],[3,4],[4,5],[5,6],[6,1],[1,3,1],[2,4,1]]);

                    [ 2  -1   1   0   0  -1 ]
                    [                       ]
                    [-1   2  -1   1   0   0 ]
                    [                       ]
                    [ 1  -1   2  -1   0   0 ]
               q := [                       ]
                    [ 0   1  -1   2  -1   0 ]
                    [                       ]
                    [ 0   0   0  -1   2  -1 ]
                    [                       ]
                    [-1   0   0   0  -1   2 ]

> uf_dynkin(q);
                               E6
```

SEE ALSO: uf_type

2.6 uf_corank - determine the corank of a unit form

Calling Sequence:

uf_corank(q)

Parameters:

q – matrix of the unit form

Synopsis:

- uf_corank determines the corank of a unit form using its critical vectors.

Examples:

```
> q:=uf_init([3,[1,2],[2,3]]);

                              [  2  -1   0 ]
                              [            ]
                         q := [ -1   2  -1 ]
                              [            ]
                              [  0  -1   2 ]

> uf_type(q);
                Positive defined, sincere weakly positive form.

> uf_corank(q);
                                   0

> q:=uf_init([6,[1,2],[1,3],[1,4],[1,5],[1,6,2],[6,2],[6,3],[6,4],[6,5]]);

                         [  2  -1  -1  -1  -1   2 ]
                         [                        ]
                         [ -1   2   0   0   0  -1 ]
                         [                        ]
                         [ -1   0   2   0   0  -1 ]
                    q := [                        ]
                         [ -1   0   0   2   0  -1 ]
                         [                        ]
                         [ -1   0   0   0   2  -1 ]
                         [                        ]
                         [  2  -1  -1  -1  -1   2 ]

> uf_critvectors(q);
                 [[0, 1, 1, 1, 1, 2], [2, 1, 1, 1, 1, 0]]

> uf_corank(q);
                                   2
```

SEE ALSO: uf_type, uf_critvectors

2.7 uf_roots - compute real roots for a unit form

Calling Sequence:

uf_roots(q)
uf_roots(q,m)

Parameters:

q – matrix of the unit form
m – optional parameter, maximal value for root entries. Default value is 6.

Synopsis:

- The function computes all real roots (with entries up to m) performing sequences of reflections on base vectors.
- The function returns a list of roots as a vector, in which the entries are roots.

Examples:

```
> q:=uf_init([2,[1,2,-2]]);

                                  [  2  -2 ]
                             q := [        ]
                                  [ -2   2 ]

> uf_roots(q);
   [[1, 0], [0, 1], [1, 2], [2, 1], [3, 2], [2, 3], [3, 4], [4, 3], [5, 4],

       [4, 5], [5, 6], [6, 5]]

> uf_roots(q,2);
                        [[1, 0], [0, 1], [1, 2], [2, 1]]

> r:=uf_roots(q,8);
   [[1, 0], [0, 1], [1, 2], [2, 1], [3, 2], [2, 3], [3, 4], [4, 3], [5, 4],

       [4, 5], [5, 6], [6, 5], [7, 6], [6, 7], [7, 8], [8, 7]]

> r[vectdim(r)];

                                  [8, 7]
```

SEE ALSO: uf_sinroots, uf_critvectors

2.8 uf_sinroots - compute sincere real roots for a unit form

Calling Sequence:
`uf_sinroots(`q`)`
`uf_sinroots(`q,m`)`

Parameters:
q – matrix of the unit form
m – optional parameter, maximal value for root entries. Default value is 6.

Synopsis:

- The function computes all sincere real roots (with entries up to m) performing sequences of reflections on base vectors.
- The function returns a list of roots as a vector, in which the entries are roots.

Examples:

```
> q:=uf_init([6,[1,2],[2,3],[3,4],[4,5],[6,3]]);

                         [ 2  -1   0   0   0   0 ]
                         [                       ]
                         [-1   2  -1   0   0   0 ]
                         [                       ]
                         [ 0  -1   2  -1   0  -1 ]
                    q := [                       ]
                         [ 0   0  -1   2  -1   0 ]
                         [                       ]
                         [ 0   0   0  -1   2   0 ]
                         [                       ]
                         [ 0   0  -1   0   0   2 ]

> uf_sinroots(q);
    [[1, 1, 1, 1, 1, 1], [1, 1, 2, 1, 1, 1], [1, 2, 2, 1, 1, 1],

        [1, 1, 2, 2, 1, 1], [1, 2, 2, 2, 1, 1], [1, 2, 3, 2, 1, 1],

        [1, 2, 3, 2, 1, 2]]
```

SEE ALSO: `uf_roots, uf_critvectors`

2.9 uf_critvectors - compute critical vectors for a unit form

Calling Sequence:
`uf_critvectors(`q`)`

Parameters:
q – matrix of the unit form

Synopsis:

- The function computes all critical vectors of a unit form.
- The function returns a list of roots as a vector, in which the entries are critical vectors.

Examples:

```
> q:=uf_init([2,[1,2,-2]]);

                              [  2  -2 ]
                         q := [        ]
                              [ -2   2 ]

> uf_critvectors(q);
                              [[1, 1]]

>  q:=uf_init([6,[1,2],[1,3],[1,4],[1,5],[1,6,2],[6,2],[6,3],[6,4],[6,5]]);

                        [  2  -1  -1  -1  -1   2 ]
                        [                        ]
                        [ -1   2   0   0   0  -1 ]
                        [                        ]
                        [ -1   0   2   0   0  -1 ]
                   q := [                        ]
                        [ -1   0   0   2   0  -1 ]
                        [                        ]
                        [ -1   0   0   0   2  -1 ]
                        [                        ]
                        [  2  -1  -1  -1  -1   2 ]

> r:=uf_critvectors(q);
               [[0, 1, 1, 1, 1, 2], [2, 1, 1, 1, 1, 0]]

> uf_eval(q,r[1]);
                                0
```

SEE ALSO: `uf_roots`, `uf_sinvectors`, `uf_corank`

2.10 uf_gabrielov - Gabrielov transformation

Calling Sequence:
`uf_gabrielov(`q,i,j`)`
Parameters:
q – matrix of the unit form
i,j – numbers of base elements
Synopsis:

- uf_gabrielov performs the Gabrielov transformation of the unit form q from vertex i to vertex j.

Examples:

```
> q:=uf_init([4,[1,2],[2,3],[2,4]]);
```

```
                    [ 2  -1   0   0 ]
                    [               ]
                    [-1   2  -1  -1 ]
               q := [               ]
                    [ 0  -1   2   0 ]
                    [               ]
                    [ 0  -1   0   2 ]
```

```
> q:=uf_gabrielov(q,1,2);
```

```
                    [ 2   1  -1  -1 ]
                    [               ]
                    [ 1   2  -1  -1 ]
               q := [               ]
                    [-1  -1   2   0 ]
                    [               ]
                    [-1  -1   0   2 ]
```

```
> q:=uf_gabrielov(q,2,1);
```

```
                    [ 2  -1  -1  -1 ]
                    [               ]
                    [-1   2   0   0 ]
               q := [               ]
                    [-1   0   2   0 ]
                    [               ]
                    [-1   0   0   2 ]
```

SEE ALSO: `uf_swelling`

2.11 uf_swelling - swelling of a unit form

Calling Sequence:
`uf_swelling(`q,i`)`
Parameters:
q – matrix of the unit form
i – number of point to swell
Synopsis:

- uf_swelling performs the swelling of the unit form q in point i.

Examples:

```
> q:=uf_init([2,[1,2]]);
```

```
                    [ 2  -1 ]
               q := [       ]
                    [-1   2 ]
```

```
> q:=uf_swelling(q,1);

                                   [  2  -1   2 ]
                                   [            ]
                              q := [ -1   2  -1 ]
                                   [            ]
                                   [  2  -1   2 ]

> q:=uf_swelling(q,2);

                                 [  2  -1   2  -1 ]
                                 [                ]
                                 [ -1   2  -1   2 ]
                            q := [                ]
                                 [  2  -1   2  -1 ]
                                 [                ]
                                 [ -1   2  -1   2 ]
```

SEE ALSO: `uf_gabrielov`

2.12 uf_addvertex - add a vertex to a unit form

Calling Sequence:
`uf_addvertex(`q,c`)`
Parameters:
q – matrix of the unit form
c – edges between the new vertex and other vertices defined as follows:
$c = [i_1, k_1, i_2, k_2, \ldots]$
where $|k_l|$ is the number of edges (solid if $k_l < 0$, dotted if $k_l > 0$) between vertices i_l and $n + 1$.

Synopsis:

- uf_addvertex adds a new vertex to the bigraph of a unit form.

Examples:

```
> q:=uf_init([1]);

                                q := [ 2 ]

> q:=uf_addvertex(q,[1,-1]);

                                  [  2  -1 ]
                             q := [        ]
                                  [ -1   2 ]

> q:=uf_addvertex(q,[1,-1,2,1]);
```

```
             [  2  -1  -1 ]
             [            ]
        q := [ -1   2   1 ]
             [            ]
             [ -1   1   2 ]
```

SEE ALSO: `uf_editedge`

2.13 uf_editedge - edit edges of a unit form

Calling Sequence:
`uf_editedge(`q,i,j,k`)`
Parameters:
q – matrix of the unit form
i,j – vertices
k – new edge between vertices i and j
Synopsis:

- uf_editedge changes edges between two vertices

Examples:

```
> q:=uf_init([3,[1,2],[2,3],[3,1]]);

             [  2  -1  -1 ]
             [            ]
        q := [ -1   2  -1 ]
             [            ]
             [ -1  -1   2 ]

> uf_type(q);
                  Nonnegative defined, critical form.

> q:=uf_editedge(q,1,2,1);

             [  2   1  -1 ]
             [            ]
        q := [  1   2  -1 ]
             [            ]
             [ -1  -1   2 ]

> uf_type(q);
                Positive defined, weakly positive form.
```

SEE ALSO: `uf_addvertex`

2.14 uf_restriction - restriction of a unit form

Calling Sequence:
`uf_restriction(`q,v`)`

Parameters:
q – matrix of the unit form
v – set of vertices to delete

Synopsis:

- uf_restriction computes the restriction of the unit form q to support $\{1,\ldots,n\} \setminus \{v_i | i = 1 \ldots \texttt{vectdim}(v)\}$

Examples:

```
> q:=uf_init([4,[1,2],[2,3],[3,4],[4,1]]);

                              [  2  -1   0  -1 ]
                              [                ]
                              [ -1   2  -1   0 ]
                         q := [                ]
                              [  0  -1   2  -1 ]
                              [                ]
                              [ -1   0  -1   2 ]

> uf_type(q);
                    Nonnegative defined, critical form.

> q:=uf_restriction(q,[1]);

                                [  2  -1   0 ]
                                [            ]
                           q := [ -1   2  -1 ]
                                [            ]
                                [  0  -1   2 ]

> uf_type(q);
              Positive defined, sincere weakly positive form.
```

SEE ALSO: `uf_addvertex, uf_editedge`

2.15 uf_eval - evaluation of a unit form

Calling Sequence:
`uf_eval(`q,v`)`

Parameters:
q – matrix of the unit form
v – a vector

Synopsis:

- uf_eval evaluates the unit form q at the vector v, i.e. it computes the value

$$q(v) = \frac{1}{2} v q v^T$$

  defined (see "**Parameters**").

Examples:

```
> q:=uf_init([6,[1,2],[1,3],[1,4],[1,5],[1,6,2],[6,2],[6,3],[6,4],[6,5]]);

                     [  2   -1   -1   -1   -1    2 ]
                     [                             ]
                     [ -1    2    0    0    0   -1 ]
                     [                             ]
                     [ -1    0    2    0    0   -1 ]
                q := [                             ]
                     [ -1    0    0    2    0   -1 ]
                     [                             ]
                     [ -1    0    0    0    2   -1 ]
                     [                             ]
                     [  2   -1   -1   -1   -1    2 ]

> uf_eval(q,[2,1,1,1,1,0]);

                                0

> uf_eval(q,[1,0,0,0,0,1]);

                                4
```

SEE ALSO: `uforms/unit_form`

3 The get_alg interface

Michael Seibt (Chemnitz)
Dietmar Guhe (Bielefeld)

The functions described in this section provide means of graphical display and graphical input for most types of algebras that currently can be handled by the various packages of the CREP system. These functions will work only together with the main package of CREP.

Once the main package has been loaded, the functions related to the graphical interface *get_alg* are made available by the command `read 'get_alg.src';` (Provided the package has been installed.) It might be more convenient however to add this command to the file '`crep.src`' or when using a Maple startup file to start CREP to add the command to this startup file, in order to have the interface immediately available when starting CREP.

3.1 get_alg - graphical input of an algebra

Calling Sequence:

`get_alg()`
`get_alg(`a`)`

Parameters:

a – an algebra in get_alg intermediary format

Description:

- The function `get_alg` invokes a graphical user interface that allows graphical input of an algebra by means of a quiver with relations. If the function is called with an optional argument, an algebra in intermediary format, the interface displays the corresponding data permitting to change it. Thus in particular the function can be used for graphical display.

- For a description of the handling of the graphical interface see `get_alg/interface`, for information on the intermediary format used by the function see `get_alg/dataformat.`

- The intermediary format used is not understood by most functions of CREP presently. Therefore there are the following functions performing conversions into various data types used by CREP:

  `pst_alg`, converting intermediary format into a bound poset algebra (cf `posetalgebra`),

  `her_alg`, converting intermediary format into a hereditary algebra (cf `heralg/quiverdata`),

  `zr_alg`, converting intermediary format into a zero relation algebra (cf `coverz/dataformat`),

- Conversely there are functions converting algebras of the types mentions above into intermediary format, allowing them to be displayed and modified via the get_alg interface. These are

  `alg_pst`, for bound poset algebras,

  `alg_heralg`, for hereditary algebras,

  `alg_zr`, for zero relation algebras.

- For more information on a specific conversion function see `<function>`.

Examples:

```
> a:=get_alg();

a := table([

    quiver = [4, [[1, 2, 1], [2, 3, 2], [1, 4, 3], [4, 3, 4]]]

    display = [[[4.740260, 2.040816], [2.646104, 4.378479],

    [4.577922, 7.012987], [6.737013, 4.341373]], [

    [3.693182, 3.209647], [3.612013, 5.695733], [5.738636, 3.191095],

    [5.657468, 5.677180]]]

    relations = {[[1, [1, 2]], [-1, [3, 4]]]}

    ])

# This is what the result of a call of get_alg might look like after input
# of a commutative square via the graphical interface invoked.
# Next we impose an additional zero relation:

> a:=get_alg(a);

a := table([

    quiver = [4, [[1, 2, 1], [2, 3, 2], [1, 4, 3], [4, 3, 4]]]

    display = [[[4.740260, 2.040816], [2.646104, 4.378479],

    [4.577922, 7.012987], [6.737013, 4.341373]], [

    [3.693182, 3.209647], [3.612013, 5.695733], [5.738636, 3.191095],

    [5.657468, 5.677180]]]

    relations = {[[1, [1, 2]], [-1, [3, 4]]], [[1, [3, 4]]]}

    ])
```

SEE ALSO: get_alg/interface, get_alg/dataformat, her_alg, pst_alg, zr_alg, alg_heralg, alg_pst, alg_zr, heralg/quiverdata, posetalgebra

3.2 The get_alg - graphical interface

3.2.1 Getting started

A function call get_alg(a) passes the algebra a to an external program, which then displays this algebra in a separate window. If no argument is given no algebra will be shown.

You then can edit, respectively input a quiver Q, i.e. vertices, arrows and relations, corresponding to the algebra.

An algebra here is given by a factor kQ/I, where k is a field, Q is the quiver and I is an ideal generated by the relations you set in the interface. The handling of the get_alg interface is as follows:

3.2.2 Main window

When you execute get_alg you get a window which is divided into five parts: the Taskbar, the Main input area, the Informationbar, the Scrollbars, and the Zoom button.

Taskbar The taskbar is the bar at the very top of the window. It is divided into three pulldown menus, a button to open a window to edit the relations, one *select* button and a *Done* button. There are also some short cuts defined. If you press the alternate key together with an underlined letter (FVERD) you get the same feature as if you click with the left mouse button on this letter.

The pulldown menus By a mouse click, it doesn't matter which button, the pulldown menus are opened. Here you can also navigate with the keyboard by using the cursor up and down keys and the Enter key.

File Here you can design your own algebra libraries. The format of the files the interface saves and loads can not be interpreted with CREP directly. Thus if you want to work in CREP with an algebra you have saved with the interface you first have to load it in the interface by calling get_alg and then press the done button (see also: Main window–>Taskbar–>*The Done* button).

With *New* you can delete the actual quiver and type in a new one.

With *Load* you open a FileSelector window (see also: File Selector). Here you can load a quiver with relations you have previously saved.

With *Save* you open a FileSelector window (see also: File Selector). You can store your quiver and your relations in a file with an arbitrary name but recommended is the extension .alg .

With *Configuration* you open the Configuration window of the interface.

Here you can declare some variables of the interface, like the Maximum Vertex Count, the Maximum Arrow Count, the Double Click Time, the Matrix Size Warning Limit (this is not needed up to now) and the Load Buffer.

If you run into memory problems increase or decrease this values.

With *About* you get the authors, the version, and some other information.

With *Exit* you leave the interface. If you haven't exported your quiver before, nothing is send back to crep (see also: Main window–>Taskbar–>Export).

View Here you can determine how the quiver looks like.

With *Fit to Screen* the quiver is zoomed in such a way that it fits nicely in the Main input area.

With *Zoom In* you can zoom into the quiver.

With *Zoom Out* you can zoom out of the quiver.

With *Redraw* the quiver is redrawn. This is only needed if you think something looks strange.

With the toggle *Show Vertex Numbers* you can decide whether or not the vertex numbers should be displayed.

With the toggle *Show Arrow Numbers* you can decide whether or not the arrow numbers should be displayed.

With the toggle *Draw Arrows as Arcs* you can decide in what your arrows should look like. They are defined by the start and end point and a third point. The toggle defines whether the interface should draw an arc through these points or connect them with lines.

With the toggle *Move Arrows with Vertices* you can decide whether the third defining point of an arrow is also moved or not if the arrow is moved.

Export Here you can export your quiver in the internal format used by the function get_alg for communication with its interface. Afterwards you can leave the get_alg interface with Exit in the File menu (see also: Main window–>Taskbar–>The pulldown menus–>File).

A shortcut for this procedure is provided by the *Done button*.

Relations This button opens and closes the relations window (see Relations window).

The select button With a right or middle mouse click the possible modes are cyclically selected. By a left mouse click you get a menu where you can specify the mode you would like to have. For a description of the modes see also: Main window–>Main input area. The possible modes are *Move Mode, Delete Mode* and, if the relations window is open, *Select Mode*.

The Done button By a left mouse click on the *Done button* you leave the get_alg interface. Now you can go on with your work in CREP. This is the best way to leave the interface.

3.2.3 Main input area

In this area your quiver is displayed. The behaviour of your mouse depends on the setting of the *select button* in the Taskbar (see also: Main window–>Taskbar–>The select button).

Move Mode In this mode the left mouse button adds a vertex to your quiver in the main input area by two mouse clicks.

With the first left mouse click you select the start point of the arrow and with a second left mouse click you specify the end point of the arrow. (see also: Main window–>Informationbar)

With a middle mouse click you add a vertex to the quiver in the main input area.

With a right mouse click you can move arrows and vertices in the following way. If you press the right mouse button near to a vertex and hold the button, you get the vertex and by depressing the button you set the vertex to the new position. The motion of the arrows is defined by the menu point Move Arrows with Vertices but they will never lose their start and end points. (see also: Main window–>Taskbar–>The pulldown menus–>View)

If you press the right mouse button near to an arrow you can define a third point of this arrow where it should go through. If the arrows are drawn as arcs, the arc is defined by the start, the end and this third point. If the arrows are drawn as lines this third point defines an edge of the arrow. This is only needed for a nice look of the quiver.

Both, arrows and vertices, are labeled with increasing numbers. The label of an arrow is set near the third point the arrow passes through.

Delete Mode The only difference to the Move Mode is the behaviour of the right mouse button. With a double click on an arrow or vertex you can delete the arrow or vertex, resp. But a vertex can only be deleted if it is not the start or end point of an arrow and an arrow can only be deleted if it is not part of a relation. (see also: Relations window)

Select Mode This mode is only available if the relations window is open (see also: Relations window).

In this mode you can select a path in the quiver by specifying the arrows of the path with mouse clicks. The left mouse button appends an arrow to the path, the middle mouse button appends in the front of the path and the right mouse button deselects the last arrow or takes back the last action. In this way also arbitrary cycles can be selected. Normally the selected path is printed yellow, but this depends on your installation and computer system, e.g. you might have a monochrome screen... ;-)

Informationbar The informationbar is the bar at the very bottom of the main window. Here is briefly printed how your mouse buttons are operating and on which vertex or arrow the operation would act.

Scrollbars If the quiver does not fit in the Main input area, you can focus on the area you want to edit with the scrollbars in the bottom and the right of the Main window.

Zoom At the bottom on the right there is a small button labeled *Z*. By pressing this button with the left mouse button you zoom in, with the right mouse button you zoom out of the quiver. By pressing the middle mouse button the quiver is zoomed in such a way that it fits on the screen. (see also: Main window–>Taskbar–>View)

3.2.4 Relations window

In this window you can select the generating relations for the ideal I for your algebra. There are five areas which will be described now.

In the first area at the top on the left you can select the start vertex of your path. In the area to the right you can select the end vertex of the path.

At the bottom on the left the selected Path is shown in the *Path Selection & Quiver Statistic* area. First you see the number of the paths then the number of the start vertex, an arrow and the number of the end arrow. Then there is a colon and right beside this you can see the numbers of the arrows of the path. By the first two fields you can decide which paths you would like to see in this area. You can select a path with a left mouse click. The selected path is highlighted in yellow (the color depends on your installation and computer system). It is also drawn yellow in the Main input area (see also: Main window–>Main input area) if the mode is *Select Mode* (see also: Main window–>Taskbar–>The select button) and you can select your path also there. In the next area at the top on the right there are several buttons.

With the *Hide Button* you hide the Relations window.

With the *Refresh button* the shown paths are newly calculated. This is necessary if you have made changes to the quiver while the Relations window is open.

With the *Next Point Pair* button you can go through all possible paths. First all paths with start vertex 1 and end vertex 1 are listed then all paths with start vertex 1 and end vertex 2 and so on.

With the *Previous Point Pair* button you cycle through all possible paths in the opposite direction.

Below these two buttons there is a select button, with which you can decide whether all paths, only circles or no circles should be shown. For the handling of a select button see Main window–>Taskbar–>The select button. But be warned that of course not all cycles can be shown because there all always infinitely many of them. That means if you want to select a cycle which is not shown, you have to use the Main input area (see also: Main window–>Main input area–>Select Mode).

The *New Relation button* adds the selected path to the Relations which are shown at the bottom on the right.

With the *Add to Relation button* you can build up the sum of two or more paths with the same start and end vertices. With a left mouse click you can select the relation to which you want to add another one.

With the *Remove Relation button* you can remove a selected relation.

In the fifth area at the bottom right, where the relations are shown, you can also change the coefficients of the relation, by typing them into the coefficient field and hitting the Enter key.

Finally, with the *Relation button*, you open a menu where you can delete all relations, add them by length, you can take such relations that your algebra gets pseudo Schurian and you can decide whether or not you want to list the empty paths and whether or not you would like to have the scrollbars on the left side.

3.2.5 File selector

Here you can select the file you want to read or where you want to save your file. This window appears in some graphical user interfaces in CREP.

Description: At the top of the File Selector window the current Directory and a pattern is shown. In the area below this all directories and all files of the current directory which match the pattern are shown. If there are changes in the directory you can force an update of the view with the *Rescan button*.

You can either select a file by a left mouse click or directly type in the name of the file at the bottom of the window.

If you have selected a file to load or save press the enter key or the Ready button in the window.

Finally there is a Cancel button which closes the File Selector window and nothing is changed.

You can also navigate through the File Selector window with the arrow keys and the enter key of your keyboard.

3.2.6 Known bugs

In the relations window the add paths by length feature doesn't work correctly.

SEE ALSO: `get_alg`, `get_alg/dataformat`

3.3 get_alg/dataformat - the intermediary format of the function get_alg

Description:

- The graphical interface of the function get_alg allows input of an algebra via a quiver Q and a generating set of relations for a suitable ideal of the quiveralgebra kQ over some field k. The corresponding data is presented using a table -let's call it a- having three entries indexed by the keywords 'quiver', 'relations', and 'display'. Consequently, the component a[quiver] describes the quiver Q, the component a[relations] the generating set of relations. The component a[display] contains data enabling the function to redisplay an algebra when it is passed back function for further treatment

- The format of these components is as follows:

  - a[quiver], describing the quiver of an algebra is either a list

    $$[n]$$

    containing a single entry, the number of vertices of the quiver n in case the quiver contains no arrows, or a pair

    $$[n, l]$$

    where the first entry n again is the number of vertices and the second entry l is a list

    $$l = [l_1, ..., l_v]$$

    containing the arrows of the quiver, each arrow given by a triple

    $$l_j = [s, t, m]$$

    where s is the startvertex, t the terminal vertex and m an integer assigned to the arrow as name. For this purpose, the vertices of the quiver are numbered consecutively from 1 to n. In fact, also the arrows are numbered consecutively starting at 1.

  - a[relations] is a set

    $$\{r_1, ..., r_u\}$$

    of relations generating an ideal I, as mentioned above. Especially, if I is the zero ideal, a[relations] will be the empty set. Otherwise, each relation is given by a list

    $$r_i = [r_{i,1}, ..., r_{i,w(i)}]$$

    consisting of pairs

    $$r_{i,j} = [k_{i,j}, p_{i,j}]$$

    where $k_{i,j}$ is an integral coefficient and $p_{i,j}$ is a path in the quiver Q described by a[quiver]. The relation r_i is then given by

    $$\sum_{j=1,...,w(i)} k_{i,j} p_{i,j} = 0$$

    A path $p_{i,j}$ is given by a list

    $$p_{i,j} = [p_1, ..., p_y]$$

    where the p_l are the numbers associated to arrows in the quiver Q. The path p then is the formal composition of these arrows, from left to right.

– a[display] either consists of an empty list

$$[]$$

in case the quiver Q described by a[quiver] is the empty quiver, or a list with a single entry

$$[l1]$$

which itself is a list, in case the quiver Q is nonempty but contains no arrows, or a pair of lists

$$[l1, l2]$$

if there are arrows. Here, $l1 = [[x_1, y_1], ..., [x_n, y_n]]$ is a list of coordinate pairs for the vertices of the quiver Q, and $l2 = [[p_1, q_1], ..., [p_v, q_v]]$ is a list of coordinate pairs for displaying the arrows of the quiver Q. Display of arrows is done in such a way, that an arrow $p = [s, t, z]$ from s to t passes through the point in the plane given by the coordinates $[p_z, q_z]$.

Examples:

```
> get_alg();

table([

    quiver = [0]

    display = []

    relations = {}

    ])

> get_alg('');

table([

    quiver = [4]

    display = [[[4.123377, 2.597403], [3.149351, 3.803339],

    [5.194805, 3.858998], [4.074676, 4.953618]]]

    relations = {}

    ])

> get_alg('');

table([

    quiver = [4, [[1, 2, 1], [2, 4, 2], [1, 3, 3], [3, 4, 4]]]
```

```
    display = [[[4.123377, 2.597403], [3.149351, 3.803339],

    [5.194805, 3.858998], [4.074676, 4.953618]], [

    [3.636364, 3.200371], [3.612013, 4.378479], [4.659091, 3.228200],

    [4.634741, 4.406308]]]

    relations = {}

    ])

> get_alg(");

table([

    quiver = [4, [[1, 2, 1], [2, 4, 2], [1, 3, 3], [3, 4, 4]]]

    display = [[[4.123377, 2.597403], [3.149351, 3.803339],

    [5.194805, 3.858998], [4.074676, 4.953618]], [

    [3.636364, 3.200371], [3.612013, 4.378479], [4.659091, 3.228200],

    [4.634741, 4.406308]]]

    relations = {[[1, [1, 2]], [-1, [3, 4]]], [[-1, [3, 4]]]}

    ])
```

SEE ALSO: `get_alg`

3.4 pst_alg - convert intermediary format into a (bound) poset algebra

Calling Sequence:

`pst_alg(`a`)`

Parameters:

a – an algebra given by get_alg intermediary format

Synopsis:

- The function `pst_alg` forces an algebra a given in the intermediary format as used by the function `get_alg` into the format of a bound poset algebra.
- The format conversion is done by first interpreting the quiver a[quiver] as the Hasse diagram of a poset. For this the quiver has to be directed. Should this not be the case, the results are bound to be strange.
- Only such relations contained in the data structure of the argument a are taken into

account that consist of a single path. Linear combinations of more than one path are ignored. The single paths then are imposed upon the posetalgebra as zero relations.

- In the data structure for a posetalgebra, commutativity relations are implicit. Thus when using the functions `get_alg` and `pst_alg` together for input of a (bound) poset algebra, no commutativity relations need to be specified.

Examples:

```
> a:=get_alg(): # interactive input via the get_alg interface
> a[quiver];
[8, [[6, 1, 1], [1, 2, 2], [2, 7, 3], [3, 1, 4], [2, 4, 5], [3, 5, 6],

    [5, 4, 7], [8, 5, 8]]]

> a[relations];
                        {[[1, [8, 7]]], [[1, [1, 2, 3]]]}

> p:=pst_alg(a);
p := [8, [[2, 1], [3], [4, 2, 5], [5, 3, 8], [6], [7, 2], [8], [1, 3, 6]],

    [[4, 8], [7, 6]]]
```

SEE ALSO: `get_alg`, `get_alg/dataformat`, `posetalgebra`

3.5 her_alg - convert intermediary format into a hereditary algebra

Calling Sequence:
`her_alg(`a`)`
Parameters:
a – an algebra given by get_alg intermediary format
Synopsis:

- The function `her_alg` extracts from an algebra a given in the intermediary format as used by the function `get_alg` the underlying hereditary algebra. This is done by taking the quiver a[quiver] of the algebra a and converting it to type `heralg/quiverdata`, used by the heralg package of CREP to denote quivers and hereditary algebras.

- Any relations contained in the data structure of the algebra a will consequently by ignored.

Examples:

```
> a:=get_alg(): # interactive input via the get_alg interface
> a[quiver];
[4, [[1, 2, 1], [1, 2, 2], [4, 3, 3], [4, 3, 4], [4, 3, 5], [2, 3, 6],

    [1, 4, 7]]]

> a[relations];
           {[[1, [1, 6]]], [[1, [2, 6]], [1, [7, 3]], [1, [7, 5]]]}
```

```
> q:=her_alg(a);
          q := [4, [[1, 2, 2], [4, 3, 3], [2, 3, 1], [1, 4, 1]]]
```

SEE ALSO: get_alg, get_alg/dataformat, heralg, heralg/quiverdata

3.6 zr_alg - convert intermediary format into a zero relation algebra

Calling Sequence:
zr_alg(a)
Parameters:
a – an algebra given by get_alg intermediary format
Synopsis:

- The function zr_alg forces an algebra a given in the intermediary format as used by the function get_alg into the format of a zero relation algebra.
- The format conversion involves ignoring all relations contained in the data structure of the algebra a which do not consist of a single path but are linear combinations of more than one path.

Examples:

```
> a:=get_alg(): # interactive input via the get_alg interface
> a[quiver];
                    [3, [[2, 1, 1], [1, 3, 2], [1, 1, 3]]]

> a[relations];
                       {[[1, [3, 3, 3]]], [[1, [1, 2]]]}

> c:=zr_alg(a);
       c := [3, [[2, 1, 1], [1, 3, 2], [1, 1, 3]], [[3, 3, 3], [1, 2]]]
```

SEE ALSO: get_alg, get_alg/dataformat, coverz

3.7 alg_pst - conversion of a bound poset algebra into intermediary format

Calling Sequence:
alg_pst(p)
alg_pst(p,'c')
Parameters:
p – a bound poset algebra
'c' – optional
Synopsis:

- The function alg_pst converts a bound poset algebra into the intermediary format used by the CREP-function get_alg.
- As in the data format for bound poset algebras commutativity relations are intrinsic and only zero relations are denoted explicitly, after a call of alg_pst(p) the resulting data structure will contain only the quiver and the zero relations of the bound poset algebra given by p. (Note that from this resulting data the algebra p can be recovered using the function pst_alg.) On the other hand, a call of alg_pst(p, 'c') will also add a complete generating set of commutativity relations. This set however will usually not be minimal.

Examples:

```
> p:= [9, {[9,7],[2,1],[3,1],[4,2,3],[5,3],[6,4,5],[7,6,8]}, {[9,8]}];
p := [9, {[9, 7], [2, 1], [3, 1], [4, 2, 3], [5, 3], [6, 4, 5], [7, 6, 8]},

    {[9, 8]}]

> a:=alg_pst(p):

> a[quiver];
[9, [[1, 2, 1], [1, 3, 2], [2, 4, 3], [3, 4, 4], [3, 5, 5], [4, 6, 6],

    [5, 6, 7], [6, 7, 8], [8, 7, 9], [7, 9, 10]]]

> a[relations];
                              {[[1, [9, 10]]]}

> b:=alg_pst(p,'c'):

> b[quiver];
[9, [[1, 2, 1], [1, 3, 2], [2, 4, 3], [3, 4, 4], [3, 5, 5], [4, 6, 6],

    [5, 6, 7], [6, 7, 8], [8, 7, 9], [7, 9, 10]]]

> b[relations];
{[[1, [9, 10]]], [[1, [1, 3]], [-1, [2, 4]]],

    [[1, [1, 3, 6]], [-1, [2, 5, 7]]], [[1, [4, 6]], [-1, [5, 7]]]}
```

SEE ALSO: get_alg, get_alg/dataformat, posetalgebra, pst_alg

3.8 alg_heralg - conversion of a hereditary algebra into intermediary format

Calling Sequence:
alg_heralg(Q)
Parameters:
Q – a quiver given by a quiverdata structure
Synopsis:

- The function alg_heralg converts a hereditary algebra given by means of the quiverdata structure used by the heralg package into the intermediary format used by the CREP-function get_alg.
- The conversion will work only if the given quiverdata structure contains no symbolic expressions for the multiplicities of arrows, i.e. all multiplicities have to evaluate to integers.
- In particular, the functions alg_heralg and get_alg together can be used to obtain a graphical display of a hereditary algebra without symbolic multiplicities. For displaying hereditary algebras in general see the function drw_heralg.

- Although meant for use with the CREP-package heralg for hereditary algebras, the function alg_heralg uses some internal parts of the main package of CREP. Therefore, unlike the functions of the heralg package themselves, the function alg_heralg cannot be used without the main package.

Examples:

```
> h:=alg_heralg([4,[[1,2,1],[2,3],[1,4,5],[4,3,3]]]);

h := table([

    quiver = [4, [[1, 2, 1], [2, 3, 2], [1, 4, 3], [1, 4, 4], [1, 4, 5],

    [1, 4, 6], [1, 4, 7], [4, 3, 8], [4, 3, 9], [4, 3, 10]]]

    relations = {}

    display = [[[5., 1.], [1., 5.], [5., 9.], [9., 5.]], [

    [3.000000000, 3.000000000], [3.000000000, 7.000000000],

    [7.905096680, 2.094903320], [7.452548340, 2.547451660],

    [7.000000000, 3.000000000], [6.547451660, 3.452548340],

    [6.094903320, 3.905096680], [7.754247233, 7.754247233],

    [7.000000000, 7.000000000], [6.245752767, 6.245752767]]]

    ])

> get_alg(h);
```

SEE ALSO: get_alg, get_alg/dataformat, drw_heralg, heralg, heralg/quiverdata

3.9 alg_zr - conversion of a zero relation algebra into intermediary format

Calling Sequence:
alg_zr(a)
Parameters:
a – a zero relation algebra
Synopsis:

- The function alg_zr converts a zero relation algebra into the intermediary format used by the CREP-function get_alg.
- As the intermediary format does not support labels for arrows such labelling is replaced by a consecutive enumeration of the arrows. During this process, the function also performs a consistency check on the resulting zero relations to ensure, that each of these actually describes a path in the underlying quiver of the algebra.

- In particular, the functions `alg_zr` and `get_alg` together can be used to obtain a graphical display of a zero relation algebra respectively its quiver. It will be noticed, that the points always are arranged in a circle. However, the graphical editing facilities of the function `get_alg` can be used to obtain a more pleasing result.

Examples:

```
> z:=[4,[[2,1,1],[2,1,1],[1,1,3],[2,2,2]
> ,[2,2,2],[3,3,2],[3,4,3],[4,2,1],[4,3,3]],[[2,2],[3,3,3],[2,3]]];
z := [4, [[2, 1, 1], [2, 1, 1], [1, 1, 3], [2, 2, 2], [2, 2, 2], [3, 3, 2],

    [3, 4, 3], [4, 2, 1], [4, 3, 3]], [[2, 2], [3, 3, 3], [2, 3]]]

> a:=alg_zr(z):
> a[quiver];
[4, [[2, 1, 1], [2, 1, 2], [1, 1, 3], [2, 2, 4], [2, 2, 5], [3, 3, 6],

    [3, 4, 7], [4, 2, 8], [4, 3, 9]]]

> a[relations];
{[[1, [6, 7]]], [[1, [4, 4]]], [[1, [4, 5]]], [[1, [5, 4]]], [[1, [5, 5]]],

    [[1, [6, 6]]], [[1, [3, 3, 3]]], [[1, [7, 9, 7]]], [[1, [9, 7, 9]]]}

> get_alg(a):
```

SEE ALSO: get_alg, get_alg/dataformat, coverz/dataformat

4 The xpre function family
Amrey Krause (Bielefeld)

The function xpre looks for and constructs preprojective components of a given algebra. Compared to the function preproj of the main package of crep, which performs a similar task, the function xpre is much more general with respect to the algebras it accepts. In addition it comes with its own graphical interface. For this reason the xpre function family has been made a package of its own, rather than an addition to the main package. However, these functions still will work only together with the main package of CREP.

Once the main package has been loaded, the functions are made available by the command `read 'xpre.src';` (Given, that the package has been installed.) It might be more convenient however to add this command to the file '`crep.src`' or, when using a Maple startup file to start CREP, to add the command to this startup file.

4.1 xpre - look for preprojective components
xpreh
xprep

Calling Sequence:

`xpre(`n`)`
`xpre(`$n,r,p1,m1,\ldots$`)`
`xpreh(`Q`)`
`xprep(`p`)`

Parameters:

n	– an integer
r	– a list of integers (see below)
p1,m1, ...	– dimension vectors and multiplicities (see below)
Q	– a heralg/quiverdata structure
p	– a bound poset algebra

Description:

- The functions of the `xpre` family determine, whether for a given algebra there are preprojective components of the Auslander-Reiten quiver and – in case there are – display the beginning of these.
- In particular, the function `xprep` does so for a bound poset algebra and the function `xpreh` for a hereditary algebra given by a `heralg/quiverdata` structure (which however must not contain symbolic multiplicities of arrows.)
- For the more general function `xpre`, an algebra has to be given in terms of its indecomposable projective modules and their radical summands, as follows:
  - The first argument n of `xpre` is the number of pairwise non-isomorphic indecomposable projective modules of the algebra (which is assumed to be basic).
  - The second argument r is a vector or list of length n containing for each projective the number pairwise non-isomorphic direct summands of its radical. For this purpose, the indecomposable projectives are assumed to be numbered from 1 to n.

- The next argument p1 is supposed to be the dimension vector of the first radical summand of the first indecomposable projective with nonzero radical – let's call this projective module P. The argument is given by a list or vector of length n. The order of the entries of the dimension vector is given by the order of the indecomposable projectives as used for the second argument r.
- The dimension vector of the first radical summand is followed by its multiplicity as radical summand of P.
- Next follows the dimension vector of the second radical summand of P – in case the radical of P decomposes – or the first radical summand of the next indecomposable projective with nonzero radical together with its multiplicity.
- And so on until the radicals of all the indecomposable projectives have been described.
- Thus a call of `xpre` has the form

$$
\begin{array}{ll}
\texttt{xpre(} & n, [r_1, \dots, r_n], \\
 & p_1, m_1, \dots, p_{r_1}, m_{r_1}, \\
 & p_{r_1+1}, m_{r_1+1}, \dots, p_{r_1+r_2}, m_{r_1+r_2} \\
 & \dots \\
 & p_{r_1+\cdots r_{n-1}+1}, \dots, m_{r_1+\cdots+r_n});
\end{array}
$$

- For a semisimple algebra, the short notation `xpre(n)` for `xpre(n,[0,...,0])` also is admissible.
- The functions of the `xpre` family use various external windows to display information, especially the preprojective components encountered, and provide different means for modifying these displays. For a detailed description of these features, please see `xpre/interface`.

Examples:

```
# The following three function calls are equivalent:
> xpre(4,[1,2,0,0],[0,1,1,1],1,[0,0,1,0],1,[0,0,0,1],1);
> xprep([4,[[2,1],[3,2],[4,2]]]);
> xpreh([4,[[1,2],[2,3],[2,4]]]);

# Also, the following two calls are equivalent:
> xprep([4,[[2,1],[3,1],[4,3,2]]]);
> xpre(4,[1,1,1,0],[0,1,1,1],1,[0,0,0,1],1,[0,0,0,1],1);

# Again, two equivalent calls:
> xpreh([3,[[1,2,2],[3,2,1]]]);
> xpre(3,[1,0,1],[0,1,0],2,[0,1,0],1);
```

SEE ALSO: xpre/interface, heralg/quiverdata, posetalgebra

4.2 The graphical interface for the xpre family

4.2.1 Getting started

Executing the command `xpre(a)`, `xpreh(Q)`, or `xprep(p)` starts the calculation of a preprojective component of the algebra given by a, Q, or p respectively, if such a component exists, and, after displaying some information about the calculation done, the Auslander-Reiten-Quiver (dimension vectors and arrows) is drawn in a separate window. You can then edit the appearance of the quiver and the dimension vectors as you like. It is also possible to print the quiver to a text file or a postscript file.

4.2.2 Main Window

The Main Window is divided into the following parts: the *Pulldown Menus*, the *Quit Button*, and the *Main Viewing Area*.

Pulldown Menus

View Menu

- Orbit Graph

  Clicking on *Orbit Graph* displays a window with information about the orbit graph of the AR-Quiver. Basically it is just the incidence matrix of the preprojective component quiver, together with the number of arrows between two orbits. For example, $P(1) : 3(2)$ in the first row of the table means that the τ-orbit of the indecomposable projective module corresponding to point no. 1 has two arrows to the τ-orbit of $P(3)$.

  By clicking on the rows of the table the corresponding orbit is highlighted in the *Viewing Area* of the *Main Window*.

  You can hide the *Orbit Graph* Window by clicking the *Close* Button.

- General Info

  The menu button *General Info* redisplays the messages shown at the start of the program. There are stored some information about the type of the preprojective component, for example which of the indecomposable projective modules have been found.

  You can hide the *General Info* window by clicking the *Close* Button.

Options Menu

- Order Dimension Vector

  Here you can change the appearance of the dimension vectors. When selecting this menu item a window with buttons ordered in a big square pops up. By clicking on one of the buttons with a number on it, you "pick up" the number, *i.e.*, the entry of the dimension vector corresponding to this number, and drop it wherever you want (inside the dialog window of course!) by clicking on an empty button. By pressing *Apply* the changes are applied without closing the dialog, by pressing *Done* the changes are applied and the window is closed. *Cancel* closes the dialog window without any changes, but not if you have already chosen *Apply* before.

- Hide Dimension Vector

  Clicking on this menu item hides the dimension vector and displays bullets instead. This is very convenient when you order the orbits of the AR-Quiver or want to get an overview of the structure of the quiver.

- Number of Modules

  Here a dialog window is displayed with an input field where you can enter an arbitrary number. This is the maximum number of modules to be displayed. Often there are so many modules shown that the display is slowed down considerably. You can speed up the display by typing 100 or 500 into this input field. Moreover, the space the dimension vector occupies on the screen is less after this because the entries are smaller numbers.

  Press *Cancel* to leave without changes or press *Return* or the *OK* button to apply the changes.

- Set Column Width

  This menu item starts a dialog box with a scrollbar which controls the distance between the dimension vectors displayed. As before, by clicking *Apply* you can view the changes without closing the window, and *Done* applies the changes and closes the dialog. By pressing *Cancel* you leave the dialog without any changes.

Export Menu

- Text File

  After clicking on the item *Text File* there pops up a *File Selector* window where you choose a text file to save the data in.

  See also: File_Selector.

  The data is stored as follows: First the *Orbit Graph* is displayed, as in the *Orbit Graph* window, to give the information about the arrows. Then the beginning of the preprojective component is printed, as on the screen, up to the number of modules you chose in the menu item *Number of Modules*.

- Postscript File

  This is the recommended output for *xpre*. This menu item shows a *File Selector* window where you can choose the output file. See also: File_Selector.

  There is an extra button labelled *Settings* in the *File Selector* window. By clicking this button another dialog is displayed where you can choose the paper size and the magnification of the postscript output. By pressing *Done* you hide this dialog and get back to the File Selector window.

  When you have selected a postscript file, the AR-Quiver displayed in the *Viewing Area* of the *Main Window* is written to this file as it is on the screen (with the magnification you selected taken into account). In particular, if you chose to hide the dimension vectors, the postscript output will also display only points and arrows. The pages are ordered the following way: If the quiver is to big to fit on one page or one row of pages, then the output is printed row after row.

  You can view postscript files with Ghostview for example, or ask your system administrator about your local installation.

Quit Button The *Quit Button* closes the *Main Window* and all associated windows of *xpre* and returns control to Maple.

Viewing Area In the *Viewing Area*, which occupies most of the *Main Window*, the calculated AR-Quiver is shown. With the scroll bars at the right and bottom of the window you can scroll to the part of the quiver you wish to see.

It is possible to rearrange the vertical position of the orbits by drag and drop. First click with the middle mouse button on one of the dimension vectors (or points) in the orbit you want to move. It is then highlighted in yellow (or any other color, depending on your system). Now drag it to the desired position by depressing and holding down the right mouse button. Drop it there by releasing the right mouse button.

It is also possible to select an orbit for dragging via the *Orbit Graph Window*. If you select a line in the table displayed there the corresponding orbit in the *Viewing Area* is highlighted at the same time and vice versa. See also *Pulldown Menus->View Menu->Orbit Graph* above.

The horizontal position can only be changed by manipulating the width of the "columns" of the quiver. How to do this is described in detail in the section *Pulldown Menus->Options Menu->Set Column Width* above.

4.2.3 Orbit Graph Window

The Orbit Graph window is described in section *Pulldown Menus->View Menu->Orbit Graph.*

4.2.4 General Info Window

The General Info window is described in section *Pulldown Menus->View Menu->General Info.*

4.2.5 File selector

Here you can select the file you want to read or where you want to save your file. This window appears in some graphical user interfaces in CREP.

Description: At the top of the File Selector window the current Directory and a pattern is shown. In the area below this all directories and all files of the current directory which match the pattern are shown. If there are changes in the directory you can force an update of the view with the *Rescan button.*

You can either select a file by a left mouse click or directly type in the name of the file at the bottom of the window.

If you have selected a file to load or save press the enter key or the Ready button in the window.

Finally there is a Cancel button which closes the File Selector window and nothing is changed.

You can also navigate through the File Selector window with the arrow keys and the enter key of your keyboard.

SEE ALSO: `xpre, xpreh, xprep`