hecke

Calculating decomposition matrices of Hecke algebras

1.5.3

1 September 2019

Dmitriy Traytel

Dmitriy Traytel
Email: traytel@in.tum.de
Homepage: http://home.in.tum.de/~traytel/hecke/

Copyright

Acknowledgements

Hecke is a port of the GAP 3 package Specht 2.4 to GAP 4. Specht 2.4 was written by Andrew Mathas, who allowed Dmitriy Traytel to use his source code as the basis for hecke.

Contents

Chapter 1

Decomposition numbers of Hecke algebras of type A

1.1 Description

Hecke is a port of the GAP 3-package Specht 2.4 to GAP 4.

This package contains functions for computing the decomposition matrices for Iwahori-Hecke algebras of the symmetric groups. As the (modular) representation theory of these algebras closely resembles that of the (modular) representation theory of the symmetric groups (indeed, the latter is a special case of the former) many of the combinatorial tools from the representation theory of the symmetric group are included in the package.

These programs grew out of the attempts by Gordon James and Andrew Mathas [JM96] to understand the decomposition matrices of Hecke algebras of type A when $q = -1$. The package is now much more general and its highlights include:

1. Hecke provides a means of working in the Grothendieck ring of a Hecke algebra H using the three natural bases corresponding to the Specht modules, projective indecomposable modules, and simple modules.

2. For Hecke algebras defined over fields of characteristic zero, the algorithm of Lascoux, Leclerc, and Thibon [LLT96] for computing decomposition numbers and "crystallized decomposition matrices" has been implemented. In principle, this gives all of the decomposition matrices of Hecke algebras defined over fields of characteristic zero.

3. Hecke provides a way of inducing and restricting modules. In addition, it is possible to "induce" decomposition matrices; this function is quite effective in calculating the decomposition matrices of Hecke algebras for small n.

4. The q-analogue of Schaper's theorem [JM97] is included, as is Kleshchev's [Kle96] algorithm of calculating the Mullineux map. Both are used extensively when inducing decomposition matrices.

5. Hecke can be used to compute the decomposition numbers of q-Schur algebras (and the general linear groups), although there is less direct support for these algebras. The decomposition matrices for the q-Schur algebras defined over fields of characteristic zero for $n < 11$ and all e are included in Hecke.

6. The Littlewood-Richard rule, its inverse, and functions for many of the standard operations on partitions (such as calculating cores, quotients, and adding and removing hooks), are included.

7. The decomposition matrices for the symmetric groups S_n are included for $n < 15$ and for all primes.

1.2 The modular representation theory of Hecke algebras

The "modular" representation theory of the Iwahori-Hecke algebras of type A was pioneered by Dipper and James [DJ86] [DJ87]; here the theory is briefly outlined, referring the reader to the references for details.

Given a commutative integral domain R and a non-zero unit q in R, let $H = H_{R,q}$ be the Hecke algebra of the symmetric group S_n on n symbols defined over R and with parameter q. For each partition μ of n, Dipper and James defined a *Specht module* $S(\mu)$. Let *rad* $S(\mu)$ be the radical of $S(\mu)$ and define $D(\mu) = S(\mu)/rad\ S(\mu)$. When R is a field, $D(\mu)$ is either zero or absolutely irreducible. Henceforth, we will always assume that R is a field.

Given a non-negative integer i, let $[i]_q = 1+q+\ldots+q^{i-1}$. Define e to be the smallest non-negative integer such that $[e]_q = 0$; if no such integer exists, we set e equal to 0. Many of the functions in this package depend upon e; the integer e is the Hecke algebras analogue of the characteristic of the field in the modular representation theory of finite groups.

A partition $\mu = (\mu_1, \mu_2, \ldots)$ is *e-singular* if there exists an integer i such that $\mu_i = \mu_{i+1} = \cdots = \mu_{i+e-1} > 0$; otherwise, μ is *e-regular*. Dipper and James [DJ86] showed that $D(\nu) \neq 0$ if and only if ν is e-regular and that the $D(\nu)$ give a complete set of non-isomorphic irreducible H-modules as ν runs over the e-regular partitions of n. Further, $S(\mu)$ and $S(\nu)$ belong to the same block if and only if μ and ν have the same e-core [DJ87][JM97]. Note that these results depend only on e and not directly on R or q.

Given two partitions μ and ν, where ν is e -regular, let $d_{\mu\nu}$ be the composition multiplicity of $D(\nu)$ in $S(\nu)$. The matrix $D = (d_{\mu\nu})$ is the *decomposition matrix* of H. When the rows and columns are ordered in a way compatible with dominance, D is lower unitriangular.

The indecomposable H-modules $P(\nu)$ are indexed by e -regular partitions ν. By general arguments, $P(\nu)$ has the same composition factors as $\sum_\mu d_{\mu\nu} S(\mu)$; so these linear combinations of modules become identified in the Grothendieck ring of H. Similarly, $D(\nu) = \sum_\mu d^{-1}_{\nu\mu} S(\mu)$ in the Grothendieck ring. These observations are the basis for many of the computations in Hecke.

1.3 Two small examples

Because of the algorithm of [LLT96], in principle, all of decomposition matrices for all Hecke algebras defined over fields of characteristic zero are known and available using Hecke. The algorithm is recursive; however, it is quite quick and, as with a car, you need never look at the engine:

Example

```
gap> H:=Specht(4);   # e=4, 'R' a field of characteristic 0
<Hecke algebra with e = 4>
gap> RInducedModule(MakePIM(H,12,2));
<direct sum of 5 P-modules>
gap> Display(last);
P(13,2) + P(12,3) + P(12,2,1) + P(10,3,2) + P(9,6)
```

The [LLT96] algorithm was applied 24 times during this calculation.

For Hecke algebras defined over fields of positive characteristic the major tool provided by Hecke, apart from the decomposition matrices contained in the libraries, is a way of "inducing" decomposition matrices. This makes it fairly easy to calculate the associated decomposition matrices for "small" n. For example, the Hecke libraries contain the decomposition matrices for the symmetric groups S_n over fields of characteristic 3 for $n < 15$. These matrices were calculated by Hecke using the following commands:

Example

```
gap> H:=Specht(3,3);   # e=3, 'R' field of characteristic 3
<Hecke algebra with e = 3>
gap> d:=DecompositionMatrix(H,5);  # known for n<2e
<7x5 decomposition matrix>
gap> Display(last);
5    | 1
4,1  | . 1
3,2  | . 1 1
3,1^2| . . . 1
2^2,1| 1 . . . 1
2,1^3| . . . . 1
1^5  | . . 1 . .
gap> for n in [6..14] do
>       d:=InducedDecompositionMatrix(d); SaveDecompositionMatrix(d);
>    od;
# Inducing..
# Inducing..
# Inducing...
# Inducing...
# Inducing...
# Inducing....
```

The function `InducedDecompositionMatrix` contains almost every trick for computing decomposition matrices (except using the spin groups).

Hecke can also be used to calculate the decomposition numbers of the q-Schur algebras; although, as yet, here no additional routines for calculating the projective indecomposables indexed by e-singular partitions. Such routines may be included in a future release, together with the (conjectural) algorithm [LT96] for computing the decomposition matrices of the q-Schur algebras over fields of characteristic zero.

1.4 Overview over this manual

Chapter 2 describes the installation of this package. Chapter 3 shows instructive examples for the usage of this package.

1.5 Credits

I would like to thank Anne Henke for offering me the interesting project of porting Specht 2.4 to the current GAP version, Max Neunhöffer for giving me an excellent introduction to the GAP 4-style of programming and Benjamin Wilson for supporting the project and helping me to understand the mathematics behind Hecke.

Also I thank Andrew Mathas for allowing me to use his GAP 3-code of the Specht 2.4 package.
The lastest version of Hecke can be obtained from
http://home.in.tum.de/~traytel/hecke/.
Dmitriy Traytel
traytel@in.tum.de
Technische Universität München, 2010.

Chapter 2

Installation of the hecke-Package

To install this package just extract the package's archive file to the GAP `pkg` directory.

By default the hecke package is not automatically loaded by GAP when it is installed. You must load the package with `LoadPackage("hecke");` before its functions become available.

Please, send me an e-mail if you have any questions, remarks, suggestions, etc. concerning this package. Also, I would like to hear about applications of this package.

Dmitriy Traytel

Chapter 3

Specht functionality

3.1 Porting notes

Porting the Specht 2.4 package to GAP 4 did not influence the algorithms but required a completely new object oriented design of the underlying data structures. In GAP 3 records were used to represent algebra objects, modules and decomposition matrices of Specht 2.4. Further functions were stored inside of such records to provide name safety.

In Hecke objects represent all the data that was named above. The overloading mechanism the former record-internal functions to be available on the toplevel. The operation selection mechanism of GAP 4 allows one to concentrate on the computation code instead of dealing with different possible argument inputs.

Since variable argument length operations are not yet supported by GAP 4, we introduced our own dispatcher facility to enable the former possibility of passing patrition arguments as sequences of integers (see 3.3).

3.1.1 Structure of Hecke

The data structure hierarchy in GAP 4 is defined through filters and their dependencies.

▷ `IsAlgebraObj` (filter)
▷ `IsHecke` (filter)
▷ `IsSchur` (filter)

`IsAlgebraObj` is a generic filter for the objects returned by the functions `Specht` (3.2.1) and `Schur` (3.2.7). Concretely, `Specht` (3.2.1) returns an `IsHecke` object (automatically also an `IsAlgebraObj` object). For design reasons `IsSchur` is a subfilter of `IsHecke`. This allows to use the same functions for Schur-algebras as for Hecke-algebras with minor restrictions.

▷ `IsAlgebraObjModule` (filter)
▷ `IsHeckeModule` (filter)
▷ `IsHeckeSpecht` (filter)
▷ `IsHeckePIM` (filter)
▷ `IsHeckeSimple` (filter)
▷ `IsFockModule` (filter)
▷ `IsFockSpecht` (filter)
▷ `IsFockPIM` (filter)

▷ IsFockSimple (filter)
▷ IsSchurModule (filter)
▷ IsSchurWeyl (filter)
▷ IsSchurPIM (filter)
▷ IsSchurSimple (filter)
▷ IsFockSchurModule (filter)
▷ IsFockSchurWeyl (filter)
▷ IsFockSchurPIM (filter)
▷ IsFockSchurSimple (filter)

The hierarchy of module objects is more complex. On top we have the filter IsAlgebraObjModule. Its direct descendant IsHeckeModule has IsHeckeSpecht, IsHeckePIM, IsHeckeSimple, IsFockModule and IsSchurModule as subfilters. Again the last two subfilter relations have no mathematical sense but are technically comfortable. The filter IsFockModule is superfilter of IsFockSpecht, IsFockPIM, IsFockSimple and IsFockSchurModule. Analogously, IsSchurModule is superfilter of IsSchurWeyl, IsSchurPIM, IsSchurSimple and IsFockSchurModule which itself is superfilter of IsFockSchurWeyl, IsFockSchurPIM, IsFockSchurSimple. Further, there are subfilter relations between IsFockSpecht and IsHeckeSpecht etc., IsFockSchurWeyl and IsFockSpecht etc., IsFockSchurWeyl and IsSchurWeyl etc., IsSchurWeyl and IsHeckeSpecht etc. filters.
▷ IsDecompositionMatrix (filter)
▷ IsCrystalDecompositionMatrix (filter)

For decomposition matrices we use the filter IsDecompositionMatrix and its subfilter IsCrystalDecompositionMatrix.

3.1.2 Renamings

To keep things as backwards compatible as possible, we tried not to change names and function signatures. But for the former H.***- and H.operations.***-style functions it makes more sense to use toplevel functions (especially when the H is not explicitly needed inside of the called operation). Here is an overview of some important changes:

GAP 3	GAP 4
H.S	MakeSpecht (3.2.3)
H.P	MakePIM (3.2.3)
H.D	MakeSimple (3.2.3)
H.Sq	MakeFockSpecht (3.2.6)
H.Pq	MakeFockPIM (3.2.6)
S.W	MakeSpecht (3.2.3)
S.F	MakeSimple (3.2.3)
InducedModule	RInducedModule (3.4.1)
RestrictedModule	RRestrictedModule (3.4.3)
H.operations.New	Module
H.operations.Collect	Collect

3.2 Specht functions

3.2.1 Specht (for an integer)

▷ `Specht(e)` (method)
▷ `Specht(e, p)` (method)
▷ `Specht(e, p, val)` (method)
▷ `Specht(e, p, val, ring)` (method)

Returns: object belonging to the filter `IsHecke` (3.1.1)

Let R be a field of characteristic 0, q a non-zero element of R, and let e be the smallest positive integer such that $1+q+\ldots+q^{e-1}=0$ (we set $e=0$ if no such integer exists). The object returned by `Specht(e)` allows calculations in the Grothendieck rings of the Hecke algebras H of type A which are defined over R and have parameter q. Below we also describe how to consider Hecke algebras defined over fields of positive characteristic.

`Specht` returns an object which contains information about the the family of Hecke algebras determined by R and q. This object needs to be passed to the most of the Hecke functions as an argument.

Example

```
gap> H:=Specht(5);
<Hecke algebra with e = 5>
gap> Display(last);
Specht(e=5, S(), P(), D())
gap> IsZeroCharacteristic(last);
true
```

There is also a method `Schur` (3.2.7) for doing calculations with the q-Schur algebra. See `DecompositionMatrix` (3.2.8), and `CrystalDecompositionMatrix` (3.2.9). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.2.2 Simple information access

We allow to read/store some information from/in the algebra object returned by `Specht` (3.2.1) using the following functions.

▷ `OrderOfQ(H)` (method)

Returns: e.

▷ `Characteristic(H)` (method)

Returns: p.

▷ `SetOrdering(H, Ordering)` (method)

Provides writing access to *Ordering* that is stored in H. The ordering influences the way how decomposition matrices are printed.

▷ `SpechtDirectory` (global variable)

Setting this global variable the user can tell Hecke where to find decomposition matrices that are not in the library and also not in the current directory. By default this variable is set to the current directory.

3.2.3 The functions MakeSpecht, MakePIM and MakeSimple

The functions MakeSpecht, MakePIM and MakeSimple return objects belonging to the filter `IsAlgebraObjModule` (3.1.1) which correspond to Specht modules (`IsHeckeSpecht` (3.1.1)), projective indecomposable modules (`IsHeckePIM` (3.1.1)) and simple modules (`IsHeckeSimple` (3.1.1)) respectively. Hecke allows manipulation of arbitrary linear combinations of these "modules", as well as a way of inducing and restricting them, "multiplying" them and converting between these three natural bases of the Grothendieck ring. Multiplication of modules corresponds to taking a tensor productand then inducing (thus giving a module for a larger Hecke algebra). Each of these three functions can be called in four different ways, as we now describe.

▷ `MakeSpecht(H, mu)` (method)
▷ `MakePIM(H, mu)` (method)
▷ `MakeSimple(H, mu)` (method)

In the first form, μ is a partition (either a list, or a sequence of integers) and the corresponding Specht module, PIM, or simple module (respectively), is returned.

Example

```
gap> H:=Specht(5);; MakePIM(H,4,3,2);; Display(last);
P(4,3,2)
```

▷ `MakeSpecht(x)` (method)
▷ `MakePIM(x)` (method)
▷ `MakeSimple(x)` (method)

Here, x is an H-module. In this form, `MakeSpecht` rewrites x as a linear combination of Specht modules, if possible. Similarly, `MakePIM` and `MakeSimple` rewrite x as a linear combination of PIMs and simple modules respectively. These conversions require knowledge of the relevant decomposition matrix of H; if this is not known then `fail` is returned (over fields of characteristic zero, all of the decomposition matrices are known via the algorithm of [LLT96]; various other decomposition matrices are included with Hecke). For example, `MakeSpecht(MakePIM(`H,μ`))` returns $\sum_\nu d_{\nu\mu}S(\nu)$ or `fail` if some of these decomposition multiplicities are not known.

Example

```
gap> Display( MakeSimple( MakePIM(H,4,3,2) ) );
D(5,3,1) + 2D(4,3,2) + D(2^4,1)
gap> Display( MakeSpecht( MakeSimple( MakeSpecht(H,1,1,1,1,1) ) ) );
 - S(5) + S(4,1) - S(3,1^2) + S(2,1^3)
```

As the last example shows, Hecke does not always behave as expected. The reason for this is that Specht modules indexed by e-singular partitions can always be written as a linear combination of Specht modules which involve only e-regular partitions. As such, it is not always clear when two elements are equal in the Grothendieck ring. Consequently, to test whether two modules are equal you should first rewrite both modules in the D-basis; this is *not* done by Hecke because it would be very inefficient.

▷ `MakeSpecht(d, mu)` (method)
▷ `MakePIM(d, mu)` (method)
▷ `MakeSimple(d, mu)` (method)

In the third form, d is a decomposition matrix and μ is a partition. This is useful when you are trying to calculate a new decomposition matrix d because it allows you to do calculations using the known entries of d to deduce information about the unknown ones. When used in this way, `MakePIM` and `MakeSimple` use d to rewrite $P(\mu)$ and $D(\mu)$ respectively as a linear combination of Specht modules and `MakeSpecht` uses d to write $S(\mu)$ as a linear combination of simple modules. If the values of the unknown entries in d are needed, `fail` is returned.

Example

```
gap> H:=Specht(3,3);;   # e = 3, p = 3 = characteristic of 'R'
gap>  d:=InducedDecompositionMatrix(DecompositionMatrix(H,14));;
# Inducing....
The following projectives are missing from <d>:
    [ 15 ]  [ 8, 7 ]
gap> Display(MakePIM(d,4,3,3,2,2,1));
S(4,3^2,2^2,1) + S(4,3^2,2,1^3) + S(4,3,2^3,1^2) + S(3^3,2^2,1^2)
gap> Display(MakeSpecht(d,7, 3, 3, 2));
D(11,2,1^2) + D(10,3,1^2) + D(8,5,1^2) + D(8,3^2,1) + D(7,6,1^2) + D(7,3^2,2)
gap> Display(MakeSimple(d,14,1));
fail
```

The final example returned `fail` because the partitions $(14,1)$ and (15) have the same 3-core (and $P(15)$ is missing from d).

▷ `MakeSpecht(d, x)` (method)

▷ `MakePIM(d, x)` (method)

▷ `MakeSimple(d, x)` (method)

In the final form, d is a decomposition matrix and x is a module. All three functions rewrite x in their respective basis using d. Again this is only useful when you are trying to calculate a new decomposition matrix because, for any "known" decomposition matrix d, `MakeSpecht(`x`)` and `MakeSpecht(`d,x`)` are equivalent (and similarly for `MakePIM` and `MakeSimple`).

Example

```
gap> Display(MakeSpecht(d, MakeSimple(d,10,5) ));
 - S(13,2) + S(10,5)
```

3.2.4 Decomposition numbers of the symmetric groups

The last example looked at Hecke algebras with parameter $q = 1$ and R a field of characteristic 3 (so $e = 3$); that is, the group algebra of the symmetric group over a field of characteristic 3. More generally, the command `Specht(`p,p`)` can be used to consider the group algebras of the symmetric groups over fields of characteristic p (i.e. $e = p$ and R a field of characteristic p). For example, the dimensions of the simple modules of S_6 over fields of characteristic 5 can be computed as follows:

Example

```
gap> H:=Specht(5,5);; SimpleDimension(H,6);
6       : 1
5,1     : 5
4,2     : 8
4,1^2   : 10
3^2     : 5
3,2,1   : 8
3,1^3   : 10
```

```
2^3     : 5
2^2,1^2 : 1
2,1^4   : 5
true
```

3.2.5 Hecke algebras over fields of positive characteristic

To consider Hecke algebras defined over arbitrary fields, `Specht` (3.2.1) must also be supplied with a valuation map *val* as an argument. The function *val* is a map from some PID into the natural numbers; at present it is needed only by functions which rely (at least implicitly), upon the q-analogue of Schaper's theorem. In general, *val* depends upon q and the characteristic of R; full details can be found in [JM97]. Over fields of characteristic zero and in the symmetric group case, the function *val* is automatically defined by `Specht` (3.2.1). When R is a field of characteristic zero, $val([i]_q)$ is 1 if e divides i and 0 otherwise (this is the valuation map associated to the prime ideal in $\mathbb{C}[v]$ generated by the e-th cyclotomic polynomial). When $q = 1$ and R is a field of characteristic p, *val* is the usual p-adic valuation map. As another example, if $q = 4$ and R is a field of characteristic 5 (so $e = 2$), then the valuation map sends the integer x to $v_5([4]_x)$ where $[4]_x$ is interpreted as an integer and v_5 is the usual 5-adic valuation. To consider this Hecke algebra one could proceed as follows:

Example

```
gap> val:=function(x) local v;
>       x:=Sum([0..x-1],v->4^v);  # x->[x]_q
>       v:=0; while x mod 5=0 do x:=x/5; v:=v+1; od;
>       return v;
>     end;;
gap> H:=Specht(2,5,val,"e2q4");; Display(H);
Specht(e=2, p=5, S(), P(), D(), HeckeRing="e2q4")
```

Notice the string "e2q4" which was also passed to `Specht` (3.2.1) in this example. Although it is not strictly necessary, it is a good idea when using a "non-standard" valuation map *val* to specify the value of *HeckeRing*. This string is used for internal bookkeeping by Hecke; in particular, it is used to determine filenames when reading and saving decomposition matrices. If a "standard" valuation map is used then *HeckeRing* is set to the string "e<e>p<p>"; otherwise it defaults to "unknown". The function `SaveDecompositionMatrix` (3.5.5) will not save any decomposition matrix for any Hecke algebra H with *HeckeRing* ="unknown".

3.2.6 The Fock space and Hecke algebras over fields of characteristic zero

For Hecke algebras H defined over fields of characteristic zero Lascoux, Leclerc and Thibon [LLT96] have described an easy, inductive, algorithm for calculating the decomposition matrices of H. Their algorithm really calculates the *canonical basis*, or (global) *crystal basis* of the Fock space; results of Grojnowski-Lusztig [Gro94] show that computing this basis is equivalent to computing the decomposition matrices of H (see also [Ari96]).

The *Fock space* $\mathscr{F}$ is an (integrable) module for the quantum group $U_q(\widehat{sl}_e)$ of the affine special linear group. $\mathscr{F}$ is a free $\mathbb{C}[v]$-module with basis the set of all Specht modules $S(\mu)$ for all partitions μ of all integers:

$\mathscr{F} = \bigoplus_{n\geq 0} \bigoplus_{\mu\vdash n} \mathbb{C}[v]S(\mu)$

Here v is an indeterminate over the integers (or strictly, $\mathbb{C}$) that is stored in the algebra object produced by `Specht` (3.2.1). The canonical basis elements $Pq(\mu)$ for the $U_q(\widehat{sl}_e)$-submodule of $\mathscr{F}$

generated by the 0-partition are indexed by e-regular partitions μ. Moreover, under *specialization*, $Pq(\mu)$ maps to $P(\mu)$. An eloquent description of the algorithm for computing $Pq(\mu)$ can be found in [LLT96].

To access the elements of the Fock space Hecke provides the functions:

▷ `MakeFockPIM(H, mu)` (method)

▷ `MakeFockSpecht(H, mu)` (method)

Notice that, unlike `MakePIM` (3.2.3) and `MakeSpecht` (3.2.3), the only arguments which `MakeFockPIM` and `MakeFockSpecht` accept are partitions.

The function `MakeFockPIM` computes the canonical basis element $Pq(\mu)$ of the Fock space corresponding to the e-regular partition μ (there is a canonical basis – defined using a larger quantum group – for the whole of the Fock space [LT96]; conjecturally, this basis can be used to compute the decomposition matrices for the q-Schur algebra over fields of characteristic zero). The second function returns a standard basis element $Sq(\mu)$ of $\mathscr{F}$.

Example

```
gap> H:=Specht(4);; MakeFockPIM(H,6,2);; Display(last);
Sq(6,2) + vSq(5,3)
gap> RRestrictedModule(last); Display(last);
<direct sum of 3 Sq-modules>
Sq(6,1) + (v+v^-1)Sq(5,2) + vSq(4,3)
gap> MakePIM(last);; Display(last);
Pq(6,1) + (v+v^-1)Pq(5,2)
gap> Specialized(last);; Display(last);
P(6,1) + 2P(5,2)
gap> MakeFockSpecht(H,5,3,2);; Display(last);
Sq(5,3,2)
gap> RInducedModule(last,0);; Display(last);
v^-1Sq(5,3^2)
```

The modules returned by `MakeFockPIM` and `MakeFockSpecht` behave very much like elements of the Grothendieck ring of H; however, they should be considered as elements of the Fock space. The key difference is that when induced or restricted "quantum" analogues of induction and restriction are used. These analogues correspond to the action of $U_q(\widehat{sl}_e)$ on $\mathscr{F}$ [LLT96].

In effect, the functions `MakeFockPIM` and `MakeFockSpecht` allow computations in the Fock space, using the functions `RInducedModule` (3.4.1) and `RRestrictedModule` (3.4.3). The functions `MakeSpecht` (3.2.3), `MakePIM` (3.2.3) and `MakeSimple` (3.2.3) can also be applied to elements of the Fock space, in which case they have the expected effect. In addition, any element of the Fock space can be specialized to give the corresponding element of the Grothendieck ring of H (it is because of this correspondence that we do not make a distinction between elements of the Fock space and the Grothendieck ring of H).

When working over fields of characteristic zero Hecke will automatically calculate any canonical basis elements that it needs for computations in the Grothendieck ring of H. If you are not interested in the canonical basis elements you need never work with them directly.

3.2.7 Schur (for an integer)

▷ `Schur(e)` (method)

▷ `Schur(e, p)` (method)

▷ `Schur(e, p, val)` (method)
▷ `Schur(e, p, val, ring)` (method)

Returns: object belonging to the filter `IsSchur` (3.1.1)

This function behaves almost identically to the function `Specht` (3.2.1), the only difference being that the belonging modules are printed as "W", "P", "F" and that they correspond to the q-Weyl modules, the projective indecomposable modules, and the simple modules of the q-Schur algebra respectively. Note that the `Make***`-functions (i.e. `MakeSpecht` (3.2.3) is used to generate q-Weyl modules). Further, note that our labeling of these modules is non-standard, following that used by James in [Jam90]. The standard labeling can be obtained from ours by replacing all partitions by their conjugates.

Almost all of the functions in Hecke which accept a Hecke algebra object H will also accept the object S returned by `Schur` .

In the current version of Hecke the decomposition matrices of q-Schur algebras are not fully supported. The `InducedDecompositionMatrix` (3.5.1) function can be applied to these matrices; however there are no additional routines available for calculating the columns corresponding to e-singular partitions. The decomposition matrices for the q-Schur algebras defined over a field of characteristic 0 for $n \leq 10$ are in the Hecke libraries.

Example

```
gap> S:=Schur(2);
<Schur algebra with e = 2>
gap> InducedDecompositionMatrix(DecompositionMatrix(S,3));
The following projectives are missing from <d>:
    [ 2, 2 ]
<5x5 decomposition matrix>
gap> Display(last);
4    | 1
3,1  | 1 1
2^2  | . 1 .
2,1^2| 1 1 . 1
1^4  | 1 . . 1 1

# DecompositionMatrix(S,4) returns the full decomposition matrix. The point of
# this example is to emphasize the current limitations of Schur.
```

Note that when S is defined over a field of characteristic zero then the functions `MakeFockSpecht` (3.2.6) and `MakeFockPIM` (3.2.6) will calculate the canonical basis elements (see `Specht` (3.2.1)); currently `MakeFockPIM(`μ`)` is implemented only for e-regular partitions.

See also `Specht` (3.2.1). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.2.8 DecompositionMatrix (for an algebra and an integer)

▷ `DecompositionMatrix(H, n[, Ordering])` (method)
▷ `DecompositionMatrix(H, file[, Ordering])` (method)

Returns: the decomposition matrix D of $H(S_n)$ where H is a Hecke algebra object returned by the function `Specht` (3.2.1) (or `Schur` (3.2.7)).

`DecompositionMatrix` first checks whether the required decomposition matrix exists as a library file (checking first in the current directory, next in the directory specified by

SpechtDirectory (3.2.2), and finally in the Hecke libraries). If the base field of H has characteristic zero, DecompositionMatrix next looks for *crystallized decomposition matrices* (see CrystalDecompositionMatrix (3.2.9)). If the decomposition matrix d is not stored in the library DecompositionMatrix will calculate d when H is a Hecke algebra with a base field of characteristic zero, and will return fail otherwise (in which case the function CalculateDecompositionMatrix (3.5.6) can be used to force Hecke to try and calculate this matrix).

For Hecke algebras defined over fields of characteristic zero, Hecke uses the algorithm of [LLT96] to calculate decomposition matrices The decomposition matrices for the q-Schur algebras for $n \leq 10$ are contained in the Hecke library, as are those for the symmetric group over fields of positive characteristic when $n < 15$.

Once a decomposition matrix is known, Hecke keeps an internal copy of it which is used by the functions MakeSpecht (3.2.3), MakePIM (3.2.3), and MakeSimple (3.2.3); these functions also read decomposition matrix files as needed.

If you set the variable SpechtDirectory (3.2.2), then Hecke will also search for decomposition matrix files in this directory. The files in the current directory override those in SpechtDirectory (3.2.2) and those in the Hecke libraries.

In the second form of the function, when a *filename* is supplied, DecompositionMatrix will read the decomposition matrix in the file *filename*, and this matrix will become Hecke's internal copy of this matrix.

By default, the rows and columns of the decomposition matrices are ordered DecompositionMatrix with an ordering function such as LengthLexicographic (3.8.12) or ReverseDominance (3.8.14). You do not need to specify the ordering you want every time you call DecompositionMatrix; Hecke will keep the same ordering until you change it again. This ordering can also be set "by hand" using the operation SetOrdering (3.2.2)

Example

```
gap> DecompositionMatrix(Specht(3),6,LengthLexicographic);
<11x7 decomposition matrix>
gap> Display(last);
6      | 1
5,1    | 1 1
4,2    | . . 1
3^2    | . 1 . 1
4,1^2  | . 1 . . 1
3,2,1  | 1 1 . 1 1 1
2^3    | 1 . . . . 1
3,1^3  | . . . . 1 1
2^2,1^2| . . . . . . 1
2,1^4  | . . . 1 . 1 .
1^6    | . . . 1 . . .
```

Once you have a decomposition matrix it is often nice to be able to print it. The on screen version is often good enough; There are also functions for converting Hecke decomposition matrices into GAP matrices and vice versa (see MatrixDecompositionMatrix (3.5.7) and DecompositionMatrixMatrix (3.5.8)).

Using the function InducedDecompositionMatrix (3.5.1), it is possible to induce a decomposition matrix. See also SaveDecompositionMatrix (3.5.5) and IsNewIndecomposable (3.5.2), Specht (3.2.1), Schur (3.2.7), and CrystalDecompositionMatrix (3.2.9). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.2.9 CrystalDecompositionMatrix

▷ `CrystalDecompositionMatrix(H, n[, Ordering])` (method)
▷ `CrystalDecompositionMatrix(H, file[, Ordering])` (method)

Returns: the crystal decomposition matrix D of $H(S_n)$ where H is a Hecke algebra object returned by the function `Specht` (3.2.1) (or `Schur` (3.2.7)).

This function is similar to `DecompositionMatrix` (3.2.8). The columns of decomposition matrices correspond to projective indecomposables; the columns of crystallized decomposition matrices correspond to the canonical basis elements of the Fock space (see `Specht` (3.2.1)). Consequently, the entries in these matrices are polynomials (in v), and by specializing (i.e. setting v equal to 1; see `Specialized` (3.9.1)), the decomposition matrices of H are obtained (see `Specht` (3.2.1)). Crystallized decomposition matrices are defined only for Hecke algebras over a base field of characteristic zero. Unlike "normal" decomposition matrices, crystallized decomposition matrices cannot be induced.

Example

```
gap> CrystalDecompositionMatrix(Specht(3), 6);
<11x7 decomposition matrix>
gap> Display(last);
6       |   1
5,1     |   v   1
4,2     |   .   .   1
4,1^2   |   .   v   .   1
3^2     |   .   v   .   .   1
3,2,1   |   v v^2   .   v   v   1
3,1^3   |   .   .   . v^2   .   v
2^3     | v^2   .   .   .   .   v
2^2,1^2|   .   .   .   .   .   .   1
2,1^4   |   .   .   .   .   v v^2   .
1^6     |   .   .   .   . v^2   .   .
gap> Specialized(last);   # set v equal to 1.
<11x7 decomposition matrix>
gap> Display(last);
6       | 1
5,1     | 1 1
4,2     | . . 1
4,1^2   | . 1 . 1
3^2     | . 1 . . 1
3,2,1   | 1 1 . 1 1 1
3,1^3   | . . . 1 . 1
2^3     | 1 . . . . 1
2^2,1^2| . . . . . . 1
2,1^4   | . . . . 1 1 .
1^6     | . . . . 1 . .
```

See also `Specht` (3.2.1), `Schur` (3.2.7), `DecompositionMatrix` (3.2.8) and `Specialized` (3.9.1). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.2.10 DecompositionNumber

▷ `DecompositionNumber(H, mu, nu)` (method)
▷ `DecompositionNumber(d, mu, nu)` (method)

This function attempts to calculate the decomposition multiplicity of $D(\nu)$ in $S(\mu)$ (equivalently, the multiplicity of $S(\mu)$ in $P(\nu)$). If $P(\nu)$ is known, we just look up the answer; if not `DecompositionNumber` tries to calculate the answer using "row and column removal" (see [Jam90, Theorem 6.18]).

Example

```
gap> H:=Specht(6);; DecompositionNumber(H,[6,4,2],[6,6]);
0
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.3 Partitions in Hecke

Many of the functions in Hecke take partitions as arguments. Partitions are usually represented by lists in GAP. In Hecke, all the functions which expect a partition will accept their argument either as a list or simply as a sequence of numbers. So, for example:

Example

```
gap> H:=Specht(4);; Print(MakeSpecht(MakePIM(H,6,4)),"\n");
S(6,4) + S(6,3,1) + S(5,3,1,1) + S(3,3,2,1,1) + S(2,2,2,2,2)
gap> Print(MakeSpecht(MakePIM(H,[6,4])),"\n");
S(6,4) + S(6,3,1) + S(5,3,1,1) + S(3,3,2,1,1) + S(2,2,2,2,2)
```

Some functions require more than one argument, but the convention still applies.

Example

```
gap> ECore(3, [6,4,2]);
[ 6, 4, 2 ]
gap> ECore(3, 6,4,2);
[ 6, 4, 2 ]
gap> GoodNodes(3, 6,4,2);
[ fail, fail, 3 ]
gap> GoodNodes(3, [6,4,2]);
[ fail, fail, 3 ]
```

Basically, it never hurts to put the extra brackets in, and they can be omitted so long as this is not ambiguous. One function where the brackets are needed is `DecompositionNumber` (3.2.10) this is clear because the function takes two partitions as its arguments.

3.4 Inducing and restricting modules

Hecke provides four functions `RInducedModule` (3.4.1), `RRestrictedModule` (3.4.3), `SInducedModule` (3.4.2) and `SRestrictedModule` (3.4.4) for inducing and restricting modules. All functions can be applied to Specht modules, PIMs, and simple modules. These functions all work by first rewriting all modules as a linear combination of Specht modules (or q-Weyl modules), and then inducing and restricting. Whenever possible the induced or restricted module will be written in the original basis.

All of these functions can also be applied to elements of the Fock space (see `Specht` (3.2.1)); in which case they correspond to the action of the generators E_i and F_i of $U_q(\widehat{sl}_e)$ on $\mathscr{F}$. There is also a function `InducedDecompositionMatrix` (3.5.1) for inducing decomposition matrices.

3.4.1 RInducedModule

▷ `RInducedModule(x)` (method)

▷ `RInducedModule(x, r1[, r2, ...])` (method)

Returns: the induced modules of the Specht modules, principal indecomposable modules, and simple modules (more accurately, their image in the Grothendieck ring).

There is an natural embedding of $H(S_n)$ in $H(S_{n+1})$ which in the usual way lets us define an *induced* $H(S_{n+1})$-module for every $H(S_n)$-module.

There is also a function `SInducedModule` (3.4.2) which provides a much faster way of r-inducing s times (and inducing s times).

Let μ be a partition. Then the induced module `RInducedModule(S(`μ`))` is easy to describe: it has the same composition factors as $\sum S(\nu)$ where ν runs over all partitions whose diagrams can be obtained by adding a single node to the diagram of μ.

Example

```
gap> H:=Specht(2,2);;
gap> Display(RInducedModule(MakeSpecht(H,7,4,3,1)));
S(8,4,3,1) + S(7,5,3,1) + S(7,4^2,1) + S(7,4,3,2) + S(7,4,3,1^2)
gap> Display(RInducedModule(MakePIM(H,5,3,1)));
P(6,3,1) + 2P(5,4,1) + P(5,3,2)
gap> Display(RInducedModule(MakeSimple(H,11,2,1)));
# D(<x>), unable to rewrite <x> as a sum of simples
S(12,2,1) + S(11,3,1) + S(11,2^2) + S(11,2,1^2)
```

When inducing indecomposable modules and simple modules, `RInducedModule` first rewrites these modules as a linear combination of Specht modules (using known decomposition matrices), and then induces this linear combination of Specht modules. If possible Hecke then rewrites the induced module back in the original basis. Note that in the last example above, the decomposition matrix for S_{15} is not known by Hecke this is why `RInducedModule` was unable to rewrite this module in the D-basis.

r-Induction

Two Specht modules $S(\mu)$ and $S(\nu)$ belong to the same block if and only if the corresponding partitions μ and ν have the same e-core [JM97] (see `ECore` (3.8.1)). Because the e-core of a partition is determined by its (multiset of) e -residues, if $S(\mu)$ and $S(\nu)$ appear in `RInducedModule(S(`τ`))`, for some partition τ, then $S(\mu)$ and $S(nu)$ belong to the same block if and only if μ and ν can be obtained by adding a node of the same e-residue to the diagram of τ. The second form of `RInducedModule` allows one to induce "within blocks" by only adding nodes of some fixed e-residue r; this is known as *r-induction*. Note that $0 \leq r < e$.

Example

```
gap> H:=Specht(4);; Display(RInducedModule(MakeSpecht(H,5,2,1)));
S(6,2,1) + S(5,3,1) + S(5,2^2) + S(5,2,1^2)
gap> Display(RInducedModule(MakeSpecht(H,5,2,1),0));
0S()
gap> Display(RInducedModule(MakeSpecht(H,5,2,1),1));
S(6,2,1) + S(5,3,1) + S(5,2,1^2)
gap> Display(RInducedModule(MakeSpecht(H,5,2,1),2));
0S()
gap> Display(RInducedModule(MakeSpecht(H,5,2,1),3));
S(5,2^2)
```

The function EResidueDiagram (3.7.13), prints the diagram of μ, labeling each node with its e-residue. A quick check of this diagram confirms the answers above.

Example

```
gap> EResidueDiagram(H,5,2,1);
   0   1   2   3   0
   3   0
   2
true
```

"Quantized" induction

When RInducedModule is applied to the canonical basis elements MakeFockPIM(μ) (or more generally elements of the Fock space; see Specht (3.2.1)), a "quantum analogue" of induction is applied. More precisely, the function RInducedModule($*, i$) corresponds to the action of the generator F_i of the quantum group $U_q(\widehat{sl}_e)$ on $\mathscr{F}$ [LLT96].

Example

```
gap> H:=Specht(3);; x:=RInducedModule(MakeFockPIM(H,4,2),1,2);;
gap> Display(x); Display(MakePIM(x));
Sq(6,2) + vSq(4^2) + v^2Sq(4,2^2)
Pq(6,2)
```

See also SInducedModule (3.4.2), RRestrictedModule (3.4.3) and SRestrictedModule (3.4.4). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.4.2 SInducedModule

▷ SInducedModule(x, s) (method)

▷ SInducedModule(x, s, r) (method)

The function SInducedModule, standing for "string induction", provides a more efficient way of r-inducing s times (and a way of inducing s times if the residue r is omitted); r-induction is explained in "RInducedModule (3.4.1).

Example

```
gap> SizeScreen([80,20]);;
gap> H:=Specht(4);; Display(SInducedModule(MakePIM(H,5,2,1),3));
P(8,2,1) + 3P(7,3,1) + 2P(7,2^2) + 6P(6,3,2) + 6P(6,3,1^2) + 3P(6,2,1^3) + 2P(\
5,3^2) + P(5,2^2,1^2)
gap> Display(SInducedModule(MakePIM(H,5,2,1),3,1));
P(6,3,1^2)
gap> Display(RInducedModule(MakePIM(H,5,2,1),1,1,1));
6P(6,3,1^2)
```

Note that the multiplicity of each summand of RInducedModule($x, r, \ldots, r$) is divisible by $s!$ and that SInducedModule divides by this constant.

As with RInducedModule (3.4.1) this function can also be applied to elements of the Fock space (see Specht (3.2.1)), in which case the quantum analogue of induction is used.

See also RInducedModule (3.4.1). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.4.3 RRestrictedModule

▷ `RRestrictedModule(x)` (method)
▷ `RRestrictedModule(x, r1[, r2, ...])` (method)

Returns: the corresponding module for $H(S_{n-1})$ when given a module x for $H(S_n)$

The restriction of the Specht module $S(\mu)$ is the linear combination of Specht modules $\sum S(\nu)$ where ν runs over the partitions whose diagrams are obtained by deleting a node from the diagram of μ. If only nodes of residue r are deleted then this corresponds to first restricting $S(\mu)$ and then taking one of the block components of the restriction; this process is known as *r-restriction* (cf. r-induction in `RInducedModule` (3.4.1)).

There is also a function `SRestrictedModule` (3.4.4) which provides a faster way of r-restricting s times (and restricting s times).

When more than one residue if given to `RRestrictedModule` it returns `RRestrictedModule`$(x,r_1,r_2,\ldots,r_k)$= `RRestrictedModule(RRestrictedModule`$(x,r_1),r_2,\ldots,r_k)$ (cf. `RInducedModule` (3.4.1)).

Example

```
gap> H:=Specht(6);; Display(RRestrictedModule(MakePIM(H,5,3,2,1),4));
2P(4,3,2,1)
gap> Display(RRestrictedModule(MakeSimple(H,5,3,2),1));
D(5,2^2)
```

"Quantized" restriction

As with `RInducedModule` (3.4.1), when `RRestrictedModule` is applied to the canonical basis elements `MakeFockPIM`(μ) a quantum analogue of restriction is applied; this time, `RRestrictedModule`$(*,i)$ corresponds to the action of the generator E_i of $U_q(\widehat{sl}_e)$ on $\mathscr{F}$[LLT96].

See also `RInducedModule` (3.4.1), `SInducedModule` (3.4.2) and `SRestrictedModule` (3.4.4). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.4.4 SRestrictedModule

▷ `SRestrictedModule(x, s)` (method)
▷ `SRestrictedModule(x, s, r)` (method)

As with `SInducedModule` (3.4.2) this function provides a more efficient way of r-restricting s times, or restricting s times if the residue r is omitted (cf. `SInducedModule` (3.4.2)).

Example

```
gap> H:=Specht(6);; Display(SRestrictedModule(MakeSpecht(H,4,3,2),3));
3S(4,2) + 2S(4,1^2) + 3S(3^2) + 6S(3,2,1) + 2S(2^3)
gap> Display(SRestrictedModule(MakePIM(H,5,4,1),2,4));
P(4^2)
```

See also `RInducedModule` (3.4.1), `SInducedModule` (3.4.2) and `RRestrictedModule` (3.4.3). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5 Operations on decomposition matrices

Hecke is a package for computing decomposition matrices; this section describes the functions available for accessing these matrices directly. In addition to decomposition matrices, Hecke also calculates the "crystallized decomposition matrices" of [LLT96] and the "adjustment matrices" introduced by James [Jam90] (and Geck [Gec92]).

Throughout Hecke we place an emphasis on calculating the projective indecomposable modulesand hence upon the columns of decomposition matrices. This approach seems more efficient than the traditional approach of calculating decomposition matrices by rows; ideally both approaches should be combined (as is done by `IsNewIndecomposable` (3.5.2)).

In principle, all decomposition matrices for all Hecke algebras defined over a field of characteristic zero are available from within Hecke. In addition, the decomposition matrices for all q-Schur algebras with $n \leq 10$ and all values of e and the p-modular decomposition matrices of the symmetric groups S_n for $n < 15$ are in the Hecke library files.

If you are using Hecke regularly to do calculations involving certain values of e it would be advantageous to have Hecke calculate and save the first 20 odd decomposition matrices that you are interested in. So, for $e = 4$ use the commands:

Example

```
gap> H:=Specht(4);; for n in [8..20] do
>      SaveDecompositionMatrix(DecompositionMatrix(H,n));
>    od;
```

Alternatively, you could save the crystallized decomposition matrices. Note that for $n < 2e$ the decomposition matrices are known (by Hecke) and easy to compute.

3.5.1 InducedDecompositionMatrix

▷ `InducedDecompositionMatrix(`*d*`)` (method)

If d is the decomposition matrix of $H(S_n)$, then `InducedDecompositionMatrix(`d`)` attempts to calculate the decomposition matrix of $H(S_{n+1})$. It does this by extracting each projective indecomposable from d and inducing these modules to obtain projective modules for $H(S_{n+1})$. `InducedDecompositionMatrix` then tries to decompose these projectives using the function `IsNewIndecomposable` (3.5.2). In general there will be columns of the decomposition matrix which `InducedDecompositionMatrix` is unable to decompose and these will have to be calculated "by hand". `InducedDecompositionMatrix` prints a list of those columns of the decomposition matrix which it is unable to calculate (this list is also printed by the function `MissingIndecomposables` (3.5.11)).

Example

```
gap> d:=DecompositionMatrix(Specht(3,3),14);
<135x57 decomposition matrix>
gap> InducedDecompositionMatrix(d);
# Inducing....
The following projectives are missing from <d>:
    [ 15 ]  [ 8, 7 ]
<176x70 decomposition matrix>
```

Note that the missing indecomposables come in "pairs" which map to each other under the Mullineux map (see `MullineuxMap` (3.7.3)).

Almost all of the decomposition matrices included in Hecke were calculated directly by `InducedDecompositionMatrix`. When n is "small" `InducedDecompositionMatrix` is

usually able to return the full decomposition matrix for $H(S_n)$. Finally, although the `InducedDecompositionMatrix` can also be applied to the decomposition matrices of the q-Schur algebras (see `Schur` (3.2.7)), `InducedDecompositionMatrix` is much less successful in inducing these decomposition matrices because it contains no special routines for dealing with the indecomposable modules of the q-Schur algebra which are indexed by e-singular partitions. Note also that we use a non-standard labeling of the decomposition matrices of q-Schur algebras; see `Schur` (3.2.7).

3.5.2 IsNewIndecomposable

▷ `IsNewIndecomposable(d, x[, mu])` (method)

Returns: `true` if it is able to show that x is indecomposable (and this indecomposable is not already listed in d), and `false` otherwise.

`IsNewIndecomposable` is the function which does all of the hard work when the function `InducedDecompositionMatrix` (3.5.1) is applied to decomposition matrices. `IsNewIndecomposable` will also print a brief description of its findings, giving an upper and lower bound on the *first* decomposition number μ for which it is unable to determine the multiplicity of $S(\mu)$ in x.

`IsNewIndecomposable` works by running through all of the partitions ν such that $P(\nu)$ could be a summand of x and it uses various results, such as the q-Schaper theorem of [JM97] (see `Schaper` (3.7.1)), the Mullineux map (see `MullineuxMap` (3.7.3)) and inducing simple modules, to determine if $P(\nu)$ does indeed split off. In addition, if d is the decomposition matrix for $H(S_n)$ then `IsNewIndecomposable` will probably use some of the decomposition matrices of $H(S_m)$ for $m \leq n$, if they are known. Consequently it is a good idea to save decomposition matrices as they are calculated (see `SaveDecompositionMatrix` (3.5.5)).

For example, in calculating the 2-modular decomposition matrices of S_r the first projective which `InducedDecompositionMatrix` (3.5.1) is unable to calculate is $P(10)$.

Example

```
gap> H:=Specht(2,2);;
gap> d:=InducedDecompositionMatrix(DecompositionMatrix(H,9));;
# Inducing.
# The following projectives are missing from <d>:
#  [ 10 ]
```

(In fact, given the above commands, Hecke will return the full decomposition matrix for S_{10} because this matrix is in the library; these were the commands that were used to calculate the decomposition matrix in the library.)

By inducing $P(9)$ we can find a projective H-module which contains $P(10)$. We can then use `IsNewIndecomposable` to try and decompose this induced module into a sum of PIMs.

Example

```
gap> SizeScreen([80,20]);; x:=RInducedModule(MakePIM(H,9),1);; Display(x);
# P(<x>), unable to rewrite <x> as a sum of projectives
S(10) + S(9,1) + S(8,2) + 2S(8,1^2) + S(7,3) + 2S(7,1^3) + 3S(6,3,1) + 3S(6,2^\
2) + 4S(6,2,1^2) + 2S(6,1^4) + 4S(5,3,2) + 5S(5,3,1^2) + 5S(5,2^2,1) + 2S(5,1^\
5) + 2S(4^2,2) + 2S(4^2,1^2) + 2S(4,3^2) + 5S(4,3,1^3) + 2S(4,2^3) + 5S(4,2^2,\
1^2) + 4S(4,2,1^4) + 2S(4,1^6) + 2S(3^3,1) + 2S(3^2,2^2) + 4S(3^2,2,1^2) + 3S(\
3^2,1^4) + 3S(3,2^2,1^3) + 2S(3,1^7) + S(2^3,1^4) + S(2^2,1^6) + S(2,1^8) + S(\
1^10)
gap> IsNewIndecomposable(d,x);
# The multiplicity of S(6,3,1) in P(10) is at least 1 and at most 2.
```

```
false
gap> Display(x);
S(10) + S(9,1) + S(8,2) + 2S(8,1^2) + S(7,3) + 2S(7,1^3) + 2S(6,3,1) + 2S(6,2^\
2) + 3S(6,2,1^2) + 2S(6,1^4) + 3S(5,3,2) + 4S(5,3,1^2) + 4S(5,2^2,1) + 2S(5,1^\
5) + 2S(4^2,2) + 2S(4^2,1^2) + 2S(4,3^2) + 4S(4,3,1^3) + 2S(4,2^3) + 4S(4,2^2,\
1^2) + 3S(4,2,1^4) + 2S(4,1^6) + 2S(3^3,1) + 2S(3^2,2^2) + 3S(3^2,2,1^2) + 2S(\
3^2,1^4) + 2S(3,2^2,1^3) + 2S(3,1^7) + S(2^3,1^4) + S(2^2,1^6) + S(2,1^8) + S(\
1^10)
```

Notice that some of the coefficients of the Specht modules in x have changed; this is because `IsNewIndecomposable` was able to determine that the multiplicity of $S(6,3,1)$ was at most 2 and so it subtracted one copy of $P(6,3,1)$ from x.

In this case, the multiplicity of $S(6,3,1)$ in $P(10)$ is easy to resolve because general theory says that this multiplicity must be odd. Therefore, $x - P(6,3,1)$ is projective. After subtracting $P(6,3,1)$ from x we again use `IsNewIndecomposable` to see if x is now indecomposable. We can tell `IsNewIndecomposable` that all of the multiplicities up to and including $S(6,3,1)$ have already been checked by giving it the addition argument $\mu = [6,3,1]$.

Example

```
gap> x:=x-MakePIM(d,6,3,1);; IsNewIndecomposable(d,x,6,3,1);
true
```

Consequently, $x = P(10)$ and we add it to the decomposition matrix d (and save it).

Example

```
gap> AddIndecomposable(d,x); SaveDecompositionMatrix(d);
```

A full description of what `IsNewIndecomposable` does can be found by reading the comments in `specht.gi`. Any suggestions or improvements on this function would be especially welcome.

See also `DecompositionMatrix` (3.2.8) and `InducedDecompositionMatrix` (3.5.1). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.3 InvertDecompositionMatrix

▷ `InvertDecompositionMatrix(d)` (method)

Returns: inverse of the (e-regular part of) d, where d is a decomposition matrix, or crystallized decomposition matrix, of a Hecke algebra or q-Schur algebra.

If part of the decomposition matrix d is unknown then `InvertDecompositionMatrix` will invert as much of d as possible.

Example

```
gap> H:=Specht(4);; d:=CrystalDecompositionMatrix(H,5);;
gap> Display(InvertDecompositionMatrix(d));
5    |   1
4,1  |   .   1
3,2  |  -v   .   1
3,1^2|   .   .   .   1
2^2,1| v^2   .  -v   .   1
2,1^3|   .   .   .   .   .   1
```

See also `DecompositionMatrix` (3.2.8) and `CrystalDecompositionMatrix` (3.2.9). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.4 AdjustmentMatrix

▷ `AdjustmentMatrix(dp, d)` (method)

Returns: the adjustment matrix a

James [Jam90] noticed and Geck [Gec92] proved, that the decomposition matrices dp for Hecke algebras defined over fields of positive characteristic admit a factorization $dp = d \cdot a$ where d is a decomposition matrix for a suitable Hecke algebra defined over a field of characteristic zero and a is the so-called *adjustment matrix*.

Example

```
gap> H:=Specht(2);; Hp:=Specht(2,2);;
gap> d:=DecompositionMatrix(H,13);; dp:=DecompositionMatrix(Hp,13);;
gap> a:=AdjustmentMatrix(dp,d);
<18x18 decomposition matrix>
gap> Display(a);
13     | 1
12,1   | . 1
11,2   | 1 . 1
10,3   | . . . 1
10,2,1 | . . . . 1
9,4    | 1 . 1 . . 1
9,3,1  | 2 . . . . . 1
8,5    | . 1 . . . . . 1
8,4,1  | 1 . . . . . . . 1
8,3,2  | . 2 . . . . . 1 . 1
7,6    | 1 . . . . 1 . . . . 1
7,5,1  | . . . . . . . 1 . . . . 1
7,4,2  | 1 . 1 . . 1 . . . . 1 . 1
7,3,2,1| . . . . . . . . . . . . . . 1
6,5,2  | . 1 . . . . . 1 . 1 . . . . 1
6,4,3  | 2 . . . 1 . . . . . . . . . . 1
6,4,2,1| . 2 . 1 . . . . . . . . . . . . 1
5,4,3,1| 4 . 2 . . . . . . . . . . . . . . 1
gap> MatrixDecompositionMatrix(dp)=
>           MatrixDecompositionMatrix(d)*MatrixDecompositionMatrix(a);
true
```

In the last line we have checked our calculation.

See also `DecompositionMatrix` (3.2.8) and `CrystalDecompositionMatrix` (3.2.9). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.5 SaveDecompositionMatrix

▷ `SaveDecompositionMatrix(d)` (method)

▷ `SaveDecompositionMatrix(d, filename)` (method)

The function `SaveDecompositionMatrix` saves the decomposition matrix d. After a decomposition matrix has been saved, the functions `MakeSpecht` (3.2.3), `MakePIM` (3.2.3) and `MakeSimple` (3.2.3) will automatically access it as needed. So, for example, before saving d in order to retrieve the indecomposable $P(\mu)$ from d it is necessary to type `MakePIM(`d,μ`)`; once d has been saved, the command `MakePIM(`μ`)` suffices.

Since InducedDecompositionMatrix (3.5.1) consults the decomposition matrices for smaller n, if they are available, it is advantageous to save decomposition matrices as they are calculated. For example, over a field of characteristic 5, the decomposition matrices for the symmetric groups S_n with $n \leq 20$ can be calculated as follows:

Example

```
gap> H:=Specht(5,5);;
gap> d:=DecompositionMatrix(H,9);;
gap> for r in [10..20] do
>      d:=InducedDecompositionMatrix(d);
>      SaveDecompositionMatrix(d);
>    od;
# Inducing...
# Inducing....
# Inducing....
# Inducing.....
# Inducing......
# Inducing.......
# Inducing........
# Inducing.........
# Inducing............
# Inducing.............
# Inducing..............
```

If your Hecke algebra object H is defined using a non-standard valuation map (see Specht (3.2.1)) then it is also necessary to set the string *HeckeRing*, or to supply the function with a *filename* before it will save your matrix. SaveDecompositionMatrix will also save adjustment matrices and the various other matrices that appear in Hecke (they can be read back in using DecompositionMatrix (3.2.8)). Each matrix has a default filename which you can over ride by supplying a *filename*. Using non-standard file names will stop Hecke from automatically accessing these matrices in future. See also DecompositionMatrix (3.2.8) and CrystalDecompositionMatrix (3.2.9). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.5.6 CalculateDecompositionMatrix

▷ CalculateDecompositionMatrix(*H*, *n*) (method)

CalculateDecompositionMatrix is similar to the function DecompositionMatrix (3.2.8) in that both functions try to return the decomposition matrix d of $H(S_n)$; the difference is that this function tries to calculate this matrix whereas the latter reads the matrix from the library files (in characteristic zero both functions apply the algorithm of [LLT96] to compute d). In effect this function is only needed when working with Hecke algebras defined over fields of positive characteristic (or when you wish to avoid the libraries). For example, if you want to do calculations with the decomposition matrix of the symmetric group S_{15} over a field of characteristic two, DecompositionMatrix (3.2.8) returns fail whereas CalculateDecompositionMatrix returns a part of the decomposition matrix.

Example

```
gap> H:=Specht(2,2);; d:=DecompositionMatrix(H,15);
# This decomposition matrix is not known; use CalculateDecompositionMatrix()
# or InducedDecompositionMatrix() to calculate with this matrix.
fail
gap> d:=CalculateDecompositionMatrix(H,15);;
```

```
# Projective indecomposable P(6,4,3,2) not known.
# Projective indecomposable P(6,5,3,1) not known.
# Projective indecomposable P(6,5,4) not known.
# Projective indecomposable P(7,4,3,1) not known.
# Projective indecomposable P(7,5,2,1) not known.
# Projective indecomposable P(7,5,3) not known.
# Projective indecomposable P(7,6,2) not known.
# Projective indecomposable P(8,4,2,1) not known.
# Projective indecomposable P(8,4,3) not known.
# Projective indecomposable P(8,5,2) not known.
# Projective indecomposable P(8,6,1) not known.
# Projective indecomposable P(8,7) not known.
# Projective indecomposable P(9,3,2,1) not known.
# Projective indecomposable P(9,4,2) not known.
# Projective indecomposable P(9,5,1) not known.
# Projective indecomposable P(9,6) not known.
# Projective indecomposable P(10,3,2) not known.
# Projective indecomposable P(10,4,1) not known.
# Projective indecomposable P(10,5) not known.
# Projective indecomposable P(11,3,1) not known.
# Projective indecomposable P(11,4) not known.
# Projective indecomposable P(12,2,1) not known.
# Projective indecomposable P(12,3) not known.
# Projective indecomposable P(13,2) not known.
# Projective indecomposable P(14,1) not known.
# Projective indecomposable P(15) not known.
gap> SizeScreen([80,20]);; MissingIndecomposables(d);
The following projectives are missing from <d>:
    [ 15 ]  [ 14, 1 ]  [ 13, 2 ]  [ 12, 3 ]  [ 12, 2, 1 ]  [ 11, 4 ]
[ 11, 3, 1 ]  [ 10, 5 ]  [ 10, 4, 1 ]  [ 10, 3, 2 ]  [ 9, 6 ]  [ 9, 5, 1 ]
[ 9, 4, 2 ]  [ 9, 3, 2, 1 ]  [ 8, 7 ]  [ 8, 6, 1 ]  [ 8, 5, 2 ]  [ 8, 4, 3 ]
[ 8, 4, 2, 1 ]  [ 7, 6, 2 ]  [ 7, 5, 3 ]  [ 7, 5, 2, 1 ]  [ 7, 4, 3, 1 ]
[ 6, 5, 4 ]  [ 6, 5, 3, 1 ]  [ 6, 4, 3, 2 ]
```

Actually, you are much better starting with the decomposition matrix of S_{14} and then applying `InducedDecompositionMatrix` (3.5.1) to this matrix. See also `DecompositionMatrix` (3.2.8). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.7 MatrixDecompositionMatrix

▷ `MatrixDecompositionMatrix(`*d*`)` (method)

Returns: the GAP matrix corresponding to the Hecke decomposition matrix *d*

The rows and columns of *d* are sorted by the ordering stored in the internal algebra object of the matrix *d*.

Example

```
gap> SizeScreen([80,20]);;
gap> MatrixDecompositionMatrix(DecompositionMatrix(Specht(3),5));
[ [ 1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 1, 1, 0, 0 ], [ 0, 0, 0, 1, 0 ],
  [ 1, 0, 0, 0, 1 ], [ 0, 0, 0, 0, 1 ], [ 0, 0, 1, 0, 0 ] ]
```

See also `DecompositionMatrix` (3.2.8) and `DecompositionMatrixMatrix` (3.5.8). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.8 DecompositionMatrixMatrix

▷ `DecompositionMatrixMatrix(H, m, n)` (method)

Returns: the Hecke decomposition matrix corresponding to the GAP matrix m

If p is the number of partitions of n and r the number of e-regular partitions of n, then m must be either $r \times r$, $p \times r$ or $p \times p$. The rows and columns of m are assumed to be indexed by partitions sorted by the ordering stored in the algebra object H (see `Specht` (3.2.1)).

Example

```
gap> H:=Specht(3);;
gap> m:=[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 1, 0 ],
>         [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ] ];;
gap> Display(DecompositionMatrixMatrix(H,m,4));
4    | 1
3,1  | . 1
2^2  | 1 . 1
2,1^2| . . . 1
1^4  | . . 1 .
```

See also `DecompositionMatrix` (3.2.8) and `MatrixDecompositionMatrix` (3.5.7). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.9 AddIndecomposable

▷ `AddIndecomposable(d, x)` (method)

`AddIndecomposable` inserts the indecomposable module x into the decomposition matrix d. If d already contains the indecomposable d then a warning is printed. The function `AddIndecomposable` also calculates `MullineuxMap( x)` (see `MullineuxMap` (3.7.3)) and adds this indecomposable to d (or checks to see that it agrees with the corresponding entry of d if this indecomposable is already in d).

See `IsNewIndecomposable` (3.5.2) for an example. See also `DecompositionMatrix` (3.2.8) and `CrystalDecompositionMatrix` (3.2.9). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.10 RemoveIndecomposable

▷ `RemoveIndecomposable(d, mu)` (method)

The function `RemoveIndecomposable` removes the column from d which corresponds to $P(\mu)$. This is sometimes useful when trying to calculate a new decomposition matrix using Hecke and want to test a possible candidate for a yet to be identified PIM.

See also `DecompositionMatrix` (3.2.8) and `CrystalDecompositionMatrix` (3.2.9). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.5.11 MissingIndecomposables

▷ `MissingIndecomposables(d)` (method)

The function MissingIndecomposables prints the list of partitions corresponding to the indecomposable modules which are not listed in d.

See also DecompositionMatrix (3.2.8) and CrystalDecompositionMatrix (3.2.9). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.6 Calculating dimensions

Hecke has two functions for calculating the dimensions of modules of Hecke algebras; SimpleDimension (3.6.1) and SpechtDimension (3.6.2). As yet, Hecke does not know how to calculate the dimensions of modules for q-Schur algebras (these depend up on q).

3.6.1 SimpleDimension

▷ SimpleDimension(*d*) (method)
▷ SimpleDimension(*H, n*) (method)
▷ SimpleDimension(*H|d, mu*) (method)

In the first two forms, SimpleDimension prints the dimensions of all of the simple modules specified by d or for the Hecke algebra $H(S_n)$ respectively. If a partition μ is supplied, as in the last form, then the dimension of the simple module $D(\mu)$ is returned. At present the function is not implemented for the simple modules of the q-Schur algebras.

Example

```
gap> H:=Specht(6);;
gap> SimpleDimension(H,11,3);
272
gap> d:=DecompositionMatrix(H,5);; SimpleDimension(d,3,2);
5
gap> SimpleDimension(d);
5     : 1
4,1   : 4
3,2   : 5
3,1^2 : 6
2^2,1 : 5
2,1^3 : 4
1^5   : 1
true
```

This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.6.2 SpechtDimension

▷ SpechtDimension(*mu*) (method)

Returns: the dimension of the Specht module $S(\mu)$

$\dim S(\mu)$ is equal to the number of standard μ-tableaux; the answer is given by the hook length formula (see [JK81]).

Example

```
gap> SpechtDimension(6,3,2,1);
5632
```

See also SimpleDimension (3.6.1). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.7 Combinatorics on Young diagrams

These functions range from the representation theoretic q-Schaper theorem and Kleshchev's algorithm for the Mullineux map through to simple combinatorial operations like adding and removing rim hooks from Young diagrams.

3.7.1 Schaper

▷ Schaper(*H*, *mu*) (method)

Returns: a linear combination of Specht modules which have the same composition factors as the sum of the modules in the "Jantzen filtration " of $S(\mu)$; see [JM97]. In particular, if ν strictly dominates μ then $D(\nu)$ is a composition factor of $S(\mu)$ if and only if it is a composition factor of Schaper(μ).

Schaper uses the valuation map attached to H (see Specht (3.2.1) and [JM97]).

One way in which the q-Schaper theorem can be applied is as follows. Suppose that we have a projective module x, written as a linear combination of Specht modules and suppose that we are trying to decide whether the projective indecomposable $P(\mu)$ is a direct summand of x. Then, providing that we know that $P(\nu)$ is not a summand of x for all (e-regular) partitions ν which strictly dominate μ (see Dominates (3.8.11)), $P(\mu)$ is a summand of x if and only if InnerProduct(Schaper(H,μ),x) is non-zero (note, in particular, that we don't need to know the indecomposable $P(\mu)$ in order to perform this calculation).

The q-Schaper theorem can also be used to check for irreduciblity; in fact, this is the basis for the criterion employed by IsSimpleModule (3.7.2).

Example

```
gap> SizeScreen([80,20]);; H:=Specht(2);;
gap> Display(Schaper(H,9,5,3,2,1));
S(17,2,1) - S(15,2,1^3) + S(13,2^3,1) - S(11,3^2,2,1) + S(10,4,3,2,1) - S(9,8,\
3) - S(9,8,1^3) + S(9,6,3,2) + S(9,6,3,1^2) + S(9,6,2^2,1)
gap> Display(Schaper(H,9,6,5,2));
0S()
```

The last calculation shows that $S(9,6,5,2)$ is irreducible when R is a field of characteristic zero and $e = 2$ (cf. IsSimpleModule(H,9,6,5,2)). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.7.2 IsSimpleModule

▷ IsSimpleModule(*H*, *mu*) (method)

Returns: true if $S(\mu)$ is simple and false otherwise.

μ an e-regular partition.

This calculation uses the valuation function of H; see Specht (3.2.1). Note that the criterion used by IsSimpleModule is completely combinatorial; it is derived from the q-Schaper theorem [JM97].

Example

```
gap> H:=Specht(3);;
gap> IsSimpleModule(H,45,31,24);
false
```

See also Schaper (3.7.1). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.7.3 MullineuxMap

▷ MullineuxMap(*e*|*H*, *mu*) (method)

The sign representation $D(1^n)$ of the Hecke algebra is the (one dimensional) representation sending T_w to $(-1)^{l(w)}$. The Hecke algebra H is not a Hopf algebra so there is no well defined action of H upon the tensor product of two H-modules; however, there is an outer automorphism # of H which corresponds to tensoring with $D(1^n)$. This sends an irreducible module $D(\mu)$ to an irreducible $D(\mu)^{\#} \cong D(\mu^{\#})$ for some e-regular partition $\mu^{\#}$. In the symmetric group case, Mullineux gave a conjectural algorithm for calculating $\mu^{\#}$; consequently the map sending μ to $\mu^{\#}$ is known as the *Mullineux map*.

Deep results of Kleshchev [Kle96] for the symmetric group give another (proven) algorithm for calculating the partition $\mu^{\#}$ (Ford and Kleshchev have deduced Mullineux's conjecture from this). Using the canonical basis, it was shown by [LLT96] that the natural generalization of Kleshchev's algorithm to H gives the Mullineux map for Hecke algebras over fields of characteristic zero. The general case follows from this, so the Mullineux map is now known for all Hecke algebras.

Kleshchev's map is easy to describe; he proved that if *gns* is any good node sequence for μ, then the sequence obtained from *gns* by replacing each residue r by $-r$ mod e is a good node sequence for $\mu^{\#}$ (see GoodNodeSequence (3.7.8)).

Example

```
gap> MullineuxMap(Specht(2),12,5,2);
[ 12, 5, 2 ]
gap> MullineuxMap(Specht(4),12,5,2);
[ 4, 4, 4, 2, 2, 1, 1, 1 ]
gap> MullineuxMap(Specht(6),12,5,2);
[ 4, 3, 2, 2, 2, 2, 1, 1 ]
gap> MullineuxMap(Specht(8),12,5,2);
[ 3, 3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1 ]
gap> MullineuxMap(Specht(10),12,5,2);
[ 3, 3, 3, 3, 2, 1, 1, 1, 1, 1 ]
```

Returns: the image of μ under the Mullineux map

▷ MullineuxMap(*d*, *mu*) (method)

The Mullineux map can also be calculated using a decomposition matrix. To see this recall that "tensoring" a Specht module $S(\mu)$ with the sign representation yields a module isomorphic to the dual of $S(\lambda)$, where λ is the partition conjugate to μ. It follows that $d_{\mu\nu} = d_{\lambda\nu^{\#}}$ for all e-regular partitions ν. Therefore, if μ is the last partition in the lexicographic order such that $d_{\mu\nu} \neq 0$ then we must have $\nu^{\#} = \lambda$. The second form of MullineuxMap uses d to calculate $\mu^{\#}$ rather than the Kleshchev-[LLT96] result. ▷ MullineuxMap(*x*) (method)

Returns: returns $x^{\#}$, the image of x under #.

Note that the above remarks show that $P(\mu)$ is mapped to $P(\mu^{\#})$ via the Mullineux map; this observation is useful when calculating decomposition matrices (and is used by the function `InducedDecompositionMatrix` (3.5.1)).

See also `GoodNodes` (3.7.6) and `GoodNodeSequence` (3.7.8). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.4 MullineuxSymbol

▷ `MullineuxSymbol(`*e*`,`*H*`,` *mu*`)` (method)

Returns: the Mullineux symbol of the e-regular partition μ.

Example

```
gap> MullineuxSymbol(5,[8,6,5,5]);
[ [ 10, 6, 5, 3 ], [ 4, 4, 3, 2 ] ]
```

See also `PartitionMullineuxSymbol` (3.7.5). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.5 PartitionMullineuxSymbol

▷ `PartitionMullineuxSymbol(`*e*`,`*H*`,` *ms*`)` (method)

Returns: the e-regular partition corresponding to the given Mullineux symbol *ms*

Example

```
gap> PartitionMullineuxSymbol(5, MullineuxSymbol(5,[8,6,5,5]) );
[ 8, 6, 5, 5 ]
```

See also `MullineuxSymbol` (3.7.4). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.6 GoodNodes

▷ `GoodNodes(`*e*`,`*H*`,` *mu*`)` (method)

Returns: a list of the rows of μ which end in a good node. The good node of residue r (if it exists) is the $(r+1)$-st element in this list.

▷ `GoodNodes(`*e*`,`*H*`,` *mu*`,` *r*`)` (method)

Returns: the number of the row which ends with the good node of residue r or `fail` if there is no good node of residue r.

Given a partition and an integer e, Kleshchev [K] defined the notion of *good node* for each residue r ($0 \leq r < e$). When e is prime and μ is e-regular, Kleshchev showed that the good nodes describe the restriction of the socle of $D(\mu)$ in the symmetric group case. Brundan [Bru98] has recently generalized this result to the Hecke algebra.

By definition, there is at most one good node for each residue r and this node is a removable node (in the diagram of μ).

Example

```
gap> GoodNodes(5,[5,4,3,2]);
[ fail, fail, 2, fail, 1 ]
gap> GoodNodes(5,[5,4,3,2],0);
fail
gap> GoodNodes(5,[5,4,3,2],4);
1
```

The good nodes also determine the Kleshchev-Mullineux map (see GoodNodeSequence (3.7.8) and MullineuxMap (3.7.3)). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.7.7 NormalNodes

▷ NormalNodes(*e*|*H*, *mu*) (method)

Returns: the numbers of the rows of μ which end in one of Kleshchev's [Kle96] normal nodes.

▷ NormalNodes(*e*|*H*, *mu*, *r*) (method)

Returns: the rows corresponding to normal nodes of the specified residue.

Example

```
gap> NormalNodes(5,[6,5,4,4,3,2,1,1,1]);
[ [ 1, 4 ], [  ], [  ], [ 2, 5 ], [  ] ]
gap> NormalNodes(5,[6,5,4,4,3,2,1,1,1],0);
[ 1, 4 ]
```

See also GoodNodes (3.7.6). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.7.8 GoodNodeSequence

▷ GoodNodeSequence(*e*|*H*, *mu*) (method)

Given an e-regular partition μ of n, a *good node sequence* for μ is a sequence gns of n residues such that μ has a good node of residue r, where r is the last residue in gns and the first $n-1$ residues in gns are a good node sequence for the partition obtained from μ by deleting its (unique) good node with residue r (see GoodNodes (3.7.6)). In general, μ will have more than one good node sequence; however, any good node sequence uniquely determines μ (see PartitionGoodNodeSequence (3.7.9)).

Example

```
gap> H:=Specht(4);; GoodNodeSequence(H,4,3,1);
[ 0, 3, 1, 0, 2, 2, 1, 3 ]
gap> GoodNodeSequence(H,4,3,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3 ]
gap> GoodNodeSequence(H,4,4,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3, 2 ]
gap> GoodNodeSequence(H,5,4,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3, 2, 0 ]
```

▷ GoodNodeSequences(*e*|*H*, *mu*) (method)

Returns: list of all good node sequences for μ

Example

```
gap> H:=Specht(4);; GoodNodeSequences(H,5,2,1);
[ [ 0, 1, 2, 3, 3, 2, 0, 0 ], [ 0, 3, 1, 2, 2, 3, 0, 0 ],
  [ 0, 1, 3, 2, 2, 3, 0, 0 ], [ 0, 1, 2, 3, 3, 0, 2, 0 ],
  [ 0, 1, 2, 3, 0, 3, 2, 0 ], [ 0, 1, 2, 3, 3, 0, 0, 2 ],
  [ 0, 1, 2, 3, 0, 3, 0, 2 ] ]
```

The good node sequences determine the Mullineux map (see GoodNodes (3.7.6) and MullineuxMap (3.7.3)). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.7.9 PartitionGoodNodeSequence

▷ `PartitionGoodNodeSequence(e|H, gns)` (method)

Returns: the unique e-regular partition corresponding to *gns* (or `fail` if in fact *gns* is not a good node sequence).

Example

```
gap> H:=Specht(4);;
gap> PartitionGoodNodeSequence(H,0, 3, 1, 0, 2, 2, 1, 3, 3, 2);
[ 4, 4, 2 ]
```

See also `GoodNodes` (3.7.6), `GoodNodeSequence` (3.7.8) and `MullineuxMap` (3.7.3). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.10 GoodNodeLatticePath

▷ `GoodNodeLatticePath(e|H, mu)` (method)

Returns: a sequence of partitions which give a path in the e-good partition lattice from the empty partition to μ.

▷ `GoodNodeLatticePaths(e|H, mu)` (method)

Returns: the list of all paths in the e-good partition lattice which end in μ.

▷ `LatticePathGoodNodeSequence(e|H, gns)` (method)

Returns: the path corresponding to a given good node sequence *gns*

Example

```
gap> GoodNodeLatticePath(3,3,2,1);
[ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 1, 1 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ]
gap> GoodNodeLatticePaths(3,3,2,1);
[ [ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 1, 1 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ],
  [ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 2 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ] ]
gap> GoodNodeSequence(4,6,3,2);
[ 0, 3, 1, 0, 2, 2, 3, 3, 0, 1, 1 ]
gap> LatticePathGoodNodeSequence(4,last);
[ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 2 ], [ 3, 2 ], [ 3, 2, 1 ], [ 4, 2, 1 ],
  [ 4, 2, 2 ], [ 5, 2, 2 ], [ 6, 2, 2 ], [ 6, 3, 2 ] ]
```

See also `GoodNodes` (3.7.6). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.11 LittlewoodRichardsonRule

▷ `LittlewoodRichardsonRule(mu, nu)` (method)

▷ `LittlewoodRichardsonCoefficient(mu, nu, tau)` (method)

Given partitions μ of n and ν of m the module $S(\mu)\otimes S(\nu)$ is naturally an $H(S_n\times S_m)$- module and, by inducing, we obtain an $H(S_{n+m})$-module. This module has the same composition factors as $\sum_\nu a^\lambda_{\mu\nu}S(\lambda)$, where the sum runs over all partitions λ of $n+m$ and the integers $a^\lambda_{\mu\nu}$ are the Littlewood-Richardson coefficients. The integers $a^\lambda_{\mu\nu}$ can be calculated using a straightforward combinatorial algorithm known as the Littlewood-Richardson rule (see [JK81]). The function

`LittlewoodRichardsonRule` returns an (unordered) list of partitions of $n+m$ in which each partition λ occurs $a^{\lambda}_{\mu\nu}$ times. The Littlewood-Richardson coefficients are independent of e; they can be read more easily from the computation $S(\mu)\otimes S(\nu)$.

Example

```
gap> SizeScreen([80,20]);;
gap> H:=Specht(0);; # the generic Hecke algebra with R=C[q]
gap> LittlewoodRichardsonRule([3,2,1],[4,2]);
[ [ 4, 3, 2, 2, 1 ], [ 4, 3, 3, 1, 1 ], [ 4, 3, 3, 2 ], [ 4, 4, 2, 1, 1 ],
  [ 4, 4, 2, 2 ], [ 4, 4, 3, 1 ], [ 5, 2, 2, 2, 1 ], [ 5, 3, 2, 1, 1 ],
  [ 5, 3, 2, 2 ], [ 5, 4, 2, 1 ], [ 5, 3, 2, 1, 1 ], [ 5, 3, 3, 1 ],
  [ 5, 4, 1, 1, 1 ], [ 5, 4, 2, 1 ], [ 5, 5, 1, 1 ], [ 5, 3, 2, 2 ],
  [ 5, 3, 3, 1 ], [ 5, 4, 2, 1 ], [ 5, 4, 3 ], [ 5, 5, 2 ], [ 6, 2, 2, 1, 1 ],
  [ 6, 3, 1, 1, 1 ], [ 6, 3, 2, 1 ], [ 6, 4, 1, 1 ], [ 6, 2, 2, 2 ],
  [ 6, 3, 2, 1 ], [ 6, 4, 2 ], [ 6, 3, 2, 1 ], [ 6, 3, 3 ], [ 6, 4, 1, 1 ],
  [ 6, 4, 2 ], [ 6, 5, 1 ], [ 7, 2, 2, 1 ], [ 7, 3, 1, 1 ], [ 7, 3, 2 ],
  [ 7, 4, 1 ] ]
gap> Display(MakeSpecht(H,3,2,1)*MakeSpecht(H,4,2));
S(7,4,1) + S(7,3,2) + S(7,3,1^2) + S(7,2^2,1) + S(6,5,1) + 2S(6,4,2) + 2S(6,4,\
1^2) + S(6,3^2) + 3S(6,3,2,1) + S(6,3,1^3) + S(6,2^3) + S(6,2^2,1^2) + S(5^2,2\
) + S(5^2,1^2) + S(5,4,3) + 3S(5,4,2,1) + S(5,4,1^3) + 2S(5,3^2,1) + 2S(5,3,2^\
2) + 2S(5,3,2,1^2) + S(5,2^3,1) + S(4^2,3,1) + S(4^2,2^2) + S(4^2,2,1^2) + S(4\
,3^2,2) + S(4,3^2,1^2) + S(4,3,2^2,1)
gap> LittlewoodRichardsonCoefficient([3,2,1],[4,2],[5,4,2,1]);
3
```

The function `LittlewoodRichardsonCoefficient` returns a single Littlewood-Richardson coefficient (although you are really better off asking for all of them, since they will all be calculated anyway).

See also `RInducedModule` (3.4.1) and `InverseLittlewoodRichardsonRule` (3.7.12). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.12 InverseLittlewoodRichardsonRule

▷ `InverseLittlewoodRichardsonRule(`*tau*`)` (method)

Returns: a list of all pairs of partitions $[\mu,\nu]$ such that the Littlewood-Richardson coefficient $a^{\tau}_{\mu\nu}$ is non-zero (see `LittlewoodRichardsonRule` (3.7.11)). The list returned is unordered and $[\mu,\nu]$ will appear $a^{\tau}_{\mu\nu}$ times in it.

Example

```
gap> SizeScreen([80,20]);; InverseLittlewoodRichardsonRule(3,2,1);
[ [ [  ], [ 3, 2, 1 ] ], [ [ 1 ], [ 3, 2 ] ], [ [ 1 ], [ 2, 2, 1 ] ],
  [ [ 1 ], [ 3, 1, 1 ] ], [ [ 1, 1 ], [ 2, 2 ] ], [ [ 1, 1 ], [ 3, 1 ] ],
  [ [ 1, 1 ], [ 2, 1, 1 ] ], [ [ 1, 1, 1 ], [ 2, 1 ] ], [ [ 2 ], [ 2, 2 ] ],
  [ [ 2 ], [ 3, 1 ] ], [ [ 2 ], [ 2, 1, 1 ] ], [ [ 2, 1 ], [ 3 ] ],
  [ [ 2, 1 ], [ 2, 1 ] ], [ [ 2, 1 ], [ 2, 1 ] ], [ [ 2, 1 ], [ 1, 1, 1 ] ],
  [ [ 2, 1, 1 ], [ 2 ] ], [ [ 2, 1, 1 ], [ 1, 1 ] ], [ [ 2, 2 ], [ 2 ] ],
  [ [ 2, 2 ], [ 1, 1 ] ], [ [ 2, 2, 1 ], [ 1 ] ], [ [ 3 ], [ 2, 1 ] ],
  [ [ 3, 1 ], [ 2 ] ], [ [ 3, 1 ], [ 1, 1 ] ], [ [ 3, 1, 1 ], [ 1 ] ],
  [ [ 3, 2 ], [ 1 ] ], [ [ 3, 2, 1 ], [  ] ] ]
```

See also `LittlewoodRichardsonRule` (3.7.11). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.13 EResidueDiagram

▷ `EResidueDiagram(H|e, mu)` (method)
▷ `EResidueDiagram(x)` (method)

The e-residue of the (i,j)-th node in the diagram of a partition μ is $(j-i) \bmod e$. `EResidueDiagram(`e,μ`)` prints the diagram of the partition μ replacing each node with its e-residue. If x is a module then `EResidueDiagram(`x`)` prints the e-residue diagrams of all of the e-regular partitions appearing in x (such diagrams are useful when trying to decide how to restrict and induce modules and also in applying results such as the "Scattering theorem" of [JM96]). It is not necessary to supply the integer e in this case because x "knows" the value of e.

Example

```
gap> H:=Specht(2);; EResidueDiagram(MakeSpecht(MakePIM(H,7,5)));
[ 7, 5 ]
   0   1   0   1   0   1   0
   1   0   1   0   1
[ 6, 5, 1 ]
   0   1   0   1   0   1
   1   0   1   0   1
   0
[ 5, 4, 2, 1 ]
   0   1   0   1   0
   1   0   1   0
   0   1
   1
# There are 3 2-regular partitions.
true
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.14 HookLengthDiagram

▷ `HookLengthDiagram(mu)` (method)

Prints the diagram of μ, replacing each node with its hook length (see [JK81]).

Example

```
gap> HookLengthDiagram(11,6,3,2);
  14  13  11   9   8   7   5   4   3   2   1
   8   7   5   3   2   1
   4   3   1
   2   1
true
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.7.15 RemoveRimHook

▷ `RemoveRimHook(mu, row, col)` (method)

Returns: the partition obtained from μ by removing the (row,col)-th rim hook from (the diagram of) μ.

Example

```
gap> RemoveRimHook([6,5,4],1,2);
[ 4, 3, 1 ]
gap> RemoveRimHook([6,5,4],2,3);
[ 6, 3, 2 ]
gap> HookLengthDiagram(6,5,4);
   8   7   6   5   3   1
   6   5   4   3   1
   4   3   2   1
true
```

See also AddRimHook (3.7.16). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.7.16 AddRimHook

▷ AddRimHook(*mu, r, h*) (method)

Returns: a list $[\nu, l]$ where ν is the partition obtained from μ by adding a rim hook of length h with its "foot" in the r-th row of (the diagram of) μ and l is the leg length of the wrapped on rim hook (see, for example, [JK81]). If the resulting diagram ν is not the diagram of a partition then fail is returned.

Example

```
gap> AddRimHook([6,4,3],1,3);
[ [ 9, 4, 3 ], 0 ]
gap> AddRimHook([6,4,3],2,3);
fail
gap> AddRimHook([6,4,3],3,3);
[ [ 6, 5, 5 ], 1 ]
gap> AddRimHook([6,4,3],4,3);
[ [ 6, 4, 3, 3 ], 0 ]
gap> AddRimHook([6,4,3],5,3);
fail
```

See also RemoveRimHook (3.7.15). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.8 Operations on partitions

This section contains functions for manipulating partitions and also several useful orderings on the set of partitions.

3.8.1 ECore

▷ ECore(*e|H, mu*) (method)

Returns: the e-core of the partition μ.

▷ EAbacus(*e|H, mu*) (method)

The e-core of a partition μ is what remains after as many rim e-hooks as possible have been removed from the diagram of μ (that this is well defined is not obvious; see [JK81]).

Example

```
gap> H:=Specht(6);; ECore(H,16,8,6,5,3,1);
[ 4, 3, 1, 1 ]
```

The e-core is calculated here using James'; notation of an *abacus* there is also an `EAbacus` function; but it is more "pretty" than useful.

See also `IsECore` (3.8.2), `EQuotient` (3.8.3) and `EWeight` (3.8.5). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.2 IsECore

▷ `IsECore(`*e*`|`*H*`, `*mu*`)` (method)

Returns: `true` if μ is an e-core and `false` otherwise.

See also `ECore` (3.8.1). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.3 EQuotient

▷ `EQuotient(`*e*`|`*H*`, `*mu*`)` (method)

Returns: the e-quotient of μ; this is a sequence of e partitions whose definition can be found in [JK81].

Example

```
gap> H:=Specht(8);; EQuotient(H,22,18,16,12,12,1,1);
[ [ 1, 1 ], [  ], [  ], [  ], [  ], [ 2, 2 ], [  ], [ 1 ] ]
```

See also `ECore` (3.8.1) and `CombineEQuotientECore` (3.8.4). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.4 CombineEQuotientECore

▷ `CombineEQuotientECore(`*e*`|`*H*`, `*q*`, `*C*`)` (method)

Returns: the partition which has e-quotient q and e -core C.

A partition is uniquely determined by its e-quotient and its e -core (see `EQuotient` (3.8.3) and `ECore` (3.8.1)).

Example

```
gap> H:=Specht(11);; mu:=[100,98,57,43,12,1];;
gap> Q:=EQuotient(H,mu);
[ [ 9 ], [  ], [  ], [  ], [  ], [  ], [ 3 ], [ 1 ], [ 9 ], [  ], [ 5 ] ]
gap> C:=ECore(H,mu);
[ 7, 2, 2, 1, 1, 1 ]
gap> CombineEQuotientECore(H,Q,C);
[ 100, 98, 57, 43, 12, 1 ]
```

See also `ECore` (3.8.1) and `EQuotient` (3.8.3). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.5 EWeight

▷ `EWeight(e|H, mu)` (method)

The e-weight of a partition is the number of e-hooks which must be removed from the partition to reach the e-core (see `ECore` (3.8.1)).

Example

```
gap> EWeight(6,[16,8,6,5,3,1]);
5
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.6 ERegularPartitions

▷ `ERegularPartitions(e|H, n)` (method)

Returns: the list of e-regular partitions of n, ordered reverse lexicographically (see `Lexicographic` (3.8.13)).

A partition $\mu = (\mu_1, \mu_2, \ldots)$ is *e-regular* if there is no integer i such that $\mu_i = \mu_{i+1} = \cdots = \mu_{i+e-1} > 0$.

Example

```
gap> H:=Specht(3);; ERegularPartitions(H,6);
[ [ 2, 2, 1, 1 ], [ 3, 2, 1 ], [ 3, 3 ], [ 4, 1, 1 ], [ 4, 2 ], [ 5, 1 ],
  [ 6 ] ]
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.7 IsERegular

▷ `IsERegular(e|H, mu)` (method)

Returns: `true` if μ is e-regular and `false` otherwise.

This functions requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.8 ConjugatePartition

▷ `ConjugatePartition(mu)` (method)

Returns: the partition whose diagram is obtained by interchanging the rows and columns in the diagram of μ.

Example

```
gap> ConjugatePartition(6,4,3,2);
[ 4, 4, 3, 2, 1, 1 ]
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.9 PartitionBetaSet

▷ `PartitionBetaSet(bn)` (method)

Returns: the partitions corresponding to the given set of beta numbers *bn*. Note in particular that *bn* must be a set of integers.

Example

```
gap> PartitionBetaSet([ 2, 3, 6, 8 ]);
[ 5, 4, 2, 2 ]
```

This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.8.10 ETopLadder

▷ ETopLadder(*e*|*H*, *mu*) (method)

The ladders in the diagram of a partition are the lines connecting nodes of constant e-residue, having slope $e-1$ (see [JK81]). A new partition can be obtained from μ by sliding all nodes up to the highest possible rungs on their ladders. **Returns:** the partition obtained in this way; it is automatically e-regular (this partition is denoted μ^R in [JK81]).

Example

```
gap> H:=Specht(4);;
gap> ETopLadder(H,1,1,1,1,1,1,1,1,1,1);
[ 4, 3, 3 ]
gap> ETopLadder(6,1,1,1,1,1,1,1,1,1,1);
[ 2, 2, 2, 2, 2 ]
```

This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.8.11 Dominates

▷ Dominates(*mu*, *nu*) (method)

Returns: true if either μ=ν or $\forall i \geq 1 : \sum_{j=1}^{i} \mu_j \geq \sum_{j=1}^{i} \nu_j$ and false otherwise.

The dominance ordering is an important partial order in the representation theory of Hecke algebra because $d_{\mu\nu} = 0$ unless ν dominates μ.

Example

```
gap> Dominates([5,4],[4,4,1]);
true
```

This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.8.12 LengthLexicographic

▷ LengthLexicographic(*mu*, *nu*) (method)

Returns: true if the length of μ is less than the length of ν or if the length of μ equals the length of ν and Lexicographic(μ,ν).

Example

```
gap> p:=Partitions(6);;Sort(p,LengthLexicographic); p;
[ [ 6 ], [ 5, 1 ], [ 4, 2 ], [ 3, 3 ], [ 4, 1, 1 ], [ 3, 2, 1 ], [ 2, 2, 2 ],
  [ 3, 1, 1, 1 ], [ 2, 2, 1, 1 ], [ 2, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ] ]
```

This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.8.13 Lexicographic

▷ `Lexicographic(mu, nu)` (method)

Returns: `true` if μ is lexicographically greater than or equal to ν.

Example

```
gap> p:=Partitions(6);;Sort(p,Lexicographic); p;
[ [ 6 ], [ 5, 1 ], [ 4, 2 ], [ 4, 1, 1 ], [ 3, 3 ], [ 3, 2, 1 ],
  [ 3, 1, 1, 1 ], [ 2, 2, 2 ], [ 2, 2, 1, 1 ], [ 2, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1 ] ]
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.8.14 ReverseDominance

▷ `ReverseDominance(mu, nu)` (method)

Returns: `true` if $\forall i > 0 : \sum_{j \geq i} \mu_j > \sum_{j \geq i} \nu_j$.

This is another total order on partitions which extends the dominance ordering (see `Dominates` (3.8.11)).

Example

```
gap> p:=Partitions(6);;Sort(p,ReverseDominance); p;
[ [ 6 ], [ 5, 1 ], [ 4, 2 ], [ 3, 3 ], [ 4, 1, 1 ], [ 3, 2, 1 ], [ 2, 2, 2 ],
  [ 3, 1, 1, 1 ], [ 2, 2, 1, 1 ], [ 2, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1, 1 ] ]
```

This is the ordering used by James in the appendix of his Springer lecture notes book.

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.9 Miscellaneous functions on modules

This section contains some functions for looking at the partitions in a given module for the Hecke algebras. Most of them are used internally by Hecke.

3.9.1 Specialized

▷ `Specialized(x[, q])` (method)

▷ `Specialized(d[, q])` (method)

Returns: the corresponding element of the Grothendieck ring or the corresponding decomposition matrix of the Hecke algebra when given an element of the Fock space x (see `Specht` (3.2.1)), or a crystallized decomposition matrix (see `CrystalDecompositionMatrix` (3.2.9)), respectively.

By default the indeterminate v is specialized to 1; however v can be specialized to any (integer) q by supplying a second argument.

Example

```
gap> SizeScreen([80,20]);; H:=Specht(2);; x:=MakeFockPIM(H,6,2);; Display(x);
Sq(6,2) + vSq(6,1^2) + vSq(5,3) + v^2Sq(5,1^3) + vSq(4,3,1) + v^2Sq(4,2^2) + (\
v^3+v)Sq(4,2,1^2) + v^2Sq(4,1^4) + v^2Sq(3^2,1^2) + v^3Sq(3,2^2,1) + v^3Sq(3,1\
^5) + v^3Sq(2^3,1^2) + v^4Sq(2^2,1^4)
gap> Display(Specialized(x));
S(6,2) + S(6,1^2) + S(5,3) + S(5,1^3) + S(4,3,1) + S(4,2^2) + 2S(4,2,1^2) + S(\
4,1^4) + S(3^2,1^2) + S(3,2^2,1) + S(3,1^5) + S(2^3,1^2) + S(2^2,1^4)
gap> Display(Specialized(x,2));
```

```
S(6,2) + 2S(6,1^2) + 2S(5,3) + 4S(5,1^3) + 2S(4,3,1) + 4S(4,2^2) + 10S(4,2,1^2\
) + 4S(4,1^4) + 4S(3^2,1^2) + 8S(3,2^2,1) + 8S(3,1^5) + 8S(2^3,1^2) + 16S(2^2,\
1^4)
```

An example of `Specialized` being applied to a crystallized decomposition matrix can be found in `CrystalDecompositionMatrix` (3.2.9). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.9.2 ERegulars

▷ `ERegulars(`x`)` (method)
▷ `ERegulars(`d`)` (method)
▷ `ListERegulars(`x`)` (method)

`ERegulars(`x`)` prints a list of the e-regular partitions, together with multiplicities, which occur in the module x. `ListERegulars(`x`)` returns an actual list of these partitions rather than printing them.

Example

```
gap> H:=Specht(8);;
gap> x:=MakeSpecht(RInducedModule(MakePIM(H,8,5,3)));; Display(x);
S(9,5,3) + S(8,6,3) + S(8,5,4) + S(8,5,3,1) + S(6,5,3^2) + S(5^2,4,3) + S(5^2,\
3^2,1)
gap> ERegulars(x);
[ 9, 5, 3 ]  [ 8, 6, 3 ]  [ 8, 5, 4 ]  [ 8, 5, 3, 1 ]
[ 6, 5, 3, 3 ]  [ 5, 5, 4, 3 ]  [ 5, 5, 3, 3, 1 ]
gap> Display(MakePIM(x));
P(9,5,3) + P(8,6,3) + P(8,5,4) + P(8,5,3,1)
```

This example shows why these functions are useful: given a projective module x, as aboveand the list of e-regular partitions in x we know the possible indecomposable direct summands of x.

Note that it is not necessary to specify what e is when calling this function because x "knows" the value of e.

The function `ERegulars` can also be applied to a decomposition matrix d; in this case it returns the unitriangular submatrix of d whose rows and columns are indexed by the e-regular partitions.

These function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.9.3 SplitECores

▷ `SplitECores(`x`)` (method)

Returns: a list $[b_1,\ldots,b_k]$ where the Specht modules in each b_i all belong to the same block (i.e. they have the same e-core).

▷ `SplitECores(`x`, `mu`)` (method)

Returns: the component of x which is in the same block as μ.

▷ `SplitECores(`x`, `y`)` (method)

Returns: the component of x which is in the same block as y.

Example

```
gap> H:=Specht(2);;
gap> Display(SplitECores(RInducedModule(MakeSpecht(H,5,3,1))));
[ S(6,3,1) + S(5,3,2) + S(5,3,1,1), S(5,4,1) ]
gap> Display(RInducedModule(MakeSpecht(H,5,3,1),0));
```

```
S(5,4,1)
gap> Display(RInducedModule(MakeSpecht(H,5,3,1),1));
S(6,3,1) + S(5,3,2) + S(5,3,1^2)
```

See also ECore (3.8.1), RInducedModule (3.4.1) and RRestrictedModule (3.4.3). This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.9.4 Coefficient

▷ Coefficient(*x*, *mu*) (method)

Returns: the coefficient of $S(\mu)$ in x (resp. $D(\mu)$, or $P(\mu)$).

Example

```
gap> SizeScreen([80,20]);;
gap> H:=Specht(3);; x:=MakeSpecht(MakePIM(H,7,3));; Display(x);
S(7,3) + S(7,2,1) + S(6,2,1^2) + S(5^2) + S(5,2^2,1) + S(4^2,1^2) + S(4,3^2) +\
 S(4,3,2,1)
gap> Coefficient(x,5,2,2,1);
1
```

This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.9.5 InnerProduct

▷ InnerProduct(*x*, *y*) (method)

Here x and y are some modules of the Hecke algebra (i.e. Specht modules, PIMS, or simple modules). InnerProduct computes the standard inner product of these elements. This is sometimes a convenient way to compute decomposition numbers (for example).

Example

```
gap> H:=Specht(2);; InnerProduct(MakeSpecht(H,2,2,2,1), MakePIM(H,4,3));
1
gap> DecompositionNumber(H,[2,2,2,1],[4,3]);
1
```

This function requires the package hecke (see LoadPackage (**Reference: LoadPackage**)).

3.10 Semi-standard and standard tableaux

These functions are not really part of Hecke proper; however they are related and may well be of use to someone. Tableaux are represented by objects, that can be constructed from a list of lists.

3.10.1 Tableau

▷ Tableau(*tab*) (method)

Returns: tableau object corresponding to the given list of lists

This is the constructor for tableau objects. The first entry of the given argument list is the list corresponding to the first row of the tableau.

3.10.2 SemiStandardTableaux

▷ `SemiStandardTableaux(mu, nu)` (method)

Returns: list of the semistandard μ-tableaux of type ν [JK81]

μ a partition, ν a composition.

Example

```
gap> SizeScreen([80,20]);; Display(SemiStandardTableaux([4,3],[1,1,1,2,2]));
[ Tableau( [ [ 1, 2, 3, 4 ], [ 4, 5, 5 ] ] ),
  Tableau( [ [ 1, 2, 3, 5 ], [ 4, 4, 5 ] ] ),
  Tableau( [ [ 1, 2, 4, 4 ], [ 3, 5, 5 ] ] ),
  Tableau( [ [ 1, 2, 4, 5 ], [ 3, 4, 5 ] ] ),
  Tableau( [ [ 1, 3, 4, 4 ], [ 2, 5, 5 ] ] ),
  Tableau( [ [ 1, 3, 4, 5 ], [ 2, 4, 5 ] ] ) ]
```

See also `StandardTableaux` (3.10.3). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.10.3 StandardTableaux

▷ `StandardTableaux(mu)` (method)

Returns: list of the standard μ-tableaux

μ a partition

Example

```
gap> SizeScreen([80,20]);; Display(StandardTableaux(4,2));
[ Tableau( [ [ 1, 2, 3, 4 ], [ 5, 6 ] ] ),
  Tableau( [ [ 1, 2, 3, 5 ], [ 4, 6 ] ] ),
  Tableau( [ [ 1, 2, 3, 6 ], [ 4, 5 ] ] ),
  Tableau( [ [ 1, 2, 4, 5 ], [ 3, 6 ] ] ),
  Tableau( [ [ 1, 2, 4, 6 ], [ 3, 5 ] ] ),
  Tableau( [ [ 1, 2, 5, 6 ], [ 3, 4 ] ] ),
  Tableau( [ [ 1, 3, 4, 5 ], [ 2, 6 ] ] ),
  Tableau( [ [ 1, 3, 4, 6 ], [ 2, 5 ] ] ),
  Tableau( [ [ 1, 3, 5, 6 ], [ 2, 4 ] ] ) ]
```

See also `SemiStandardTableaux` (3.10.2). This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.10.4 ConjugateTableau

▷ `ConjugateTableau(tab)` (method)

Returns: tableau obtained from *tab* by interchangings its rows and columns

Example

```
gap> Display(ConjugateTableau(Tableau([ [ 1, 3, 5, 6 ], [ 2, 4 ] ])));
Standard Tableau:
1        2
3        4
5
6
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.10.5 ShapeTableau

▷ `ShapeTableau(tab)` (method)

Returns: the partition (or composition) obtained from *tab*

Example

```
gap> ShapeTableau( Tableau([ [ 1, 1, 2, 3 ], [ 4, 5 ] ]) );
[ 4, 2 ]
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

3.10.6 TypeTableau

▷ `TypeTableau(tab)` (method)

Returns: the type of the (semistandard) tableau *tab*

The type of a tableau is, the composition $\sigma = (\sigma_1, \sigma_2, \ldots)$ where σ_i is the number of entries in *tab* which are equal to i.

Example

```
gap> SizeScreen([80,20]);;
gap> List(SemiStandardTableaux([5,4,2],[4,3,0,1,3]),TypeTableau);
[ [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ],
  [ 4, 3, 0, 1, 3 ] ]
```

This function requires the package hecke (see `LoadPackage` (**Reference: LoadPackage**)).

References

[Ari96] Susumu Ariki. On the decomposition numbers of the Hecke algebra of $G(m,1,n)$. *J. Math. Kyoto Univ.*, 36(4):789–808, 1996. 14

[Bru98] Jonathan Brundan. Modular branching rules and the Mullineux map for Hecke algebras of type A. *Proc. London Math. Soc. (3)*, 77(3):551–581, 1998. 33

[DJ86] Richard Dipper and Gordon James. Representations of Hecke algebras of general linear groups. *Proc. London Math. Soc. (3)*, 52(1):20–52, 1986. 5

[DJ87] Richard Dipper and Gordon James. Blocks and idempotents of Hecke algebras of general linear groups. *Proc. London Math. Soc. (3)*, 54(1):57–82, 1987. 5

[Gec92] Meinolf Geck. Brauer trees of Hecke algebras. *Comm. Algebra*, 20(10):2937–2973, 1992. 23, 26

[Gro94] I. Grojnowski. Representations of affine Hecke algebras (and affine quantum GL_n) at roots of unity. *Internat. Math. Res. Notices*, 1994(5):215 ff., approx. 3 pp. (electronic), 1994. 14

[Jam90] Gordon James. The decomposition matrices of $\mathrm{GL}_n(q)$ for $n \leq 10$. *Proc. London Math. Soc. (3)*, 60(2):225–265, 1990. 16, 19, 23, 26

[JK81] Gordon James and Adalbert Kerber. *The representation theory of the symmetric group*, volume 16 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley Publishing Co., Reading, Mass., 1981. With a foreword by P. M. Cohn, With an introduction by Gilbert de B. Robinson. 30, 35, 37, 38, 39, 41, 45

[JM96] Gordon James and Andrew Mathas. Hecke algebras of type $\mathbf{A}$ with $q = -1$. *J. Algebra*, 184(1):102–158, 1996. 4, 37

[JM97] Gordon James and Andrew Mathas. A q-analogue of the Jantzen-Schaper theorem. *Proc. London Math. Soc. (3)*, 74(2):241–274, 1997. 4, 5, 14, 20, 24, 31

[Kle96] A. S. Kleshchev. Branching rules for modular representations of symmetric groups. III. Some corollaries and a problem of Mullineux. *J. London Math. Soc. (2)*, 54(1):25–38, 1996. 4, 32, 34

[LLT96] Alain Lascoux, Bernard Leclerc, and Jean-Yves Thibon. Hecke algebras at roots of unity and crystal bases of quantum affine algebras. *Comm. Math. Phys.*, 181(1):205–263, 1996. 4, 5, 6, 12, 14, 15, 17, 21, 22, 23, 27, 32

[LT96] Bernard Leclerc and Jean-Yves Thibon. Canonical bases of q-deformed Fock spaces. *Internat. Math. Res. Notices*, 1996(9):447–456, 1996. 6, 15

Index