

Handbook of Magma Functions

Part II: Sets, Sequences, and Mappings

Version 2.30-1

Sydney

November 10, 2026

Editors

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors

Eran Assaf *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *Bruce Cox* *Brendan Creutz* *Steve Donnelly* *Tim Dokchitser* *Willem de Graaf* *Andreas-Stephan Elsenhans* *Claus Fieker* *Damien Fisher* *Volker Gebhardt* *Sergei Haller* *Michael Harrison* *Florian Hess* *Derek Holt* *David Howden* *Al Kasprzyk* *Markus Kirschmer* *David Kohel* *Axel Kohnert* *Dimitri Leemans* *Paulette Lieby* *Graham Matthews* *Scott Murray* *Eamonn O'Brien* *Dan Roozemond* *Ben Smith* *Bernd Souvignier* *William Stein* *Allan Steel* *Damien Stehlé* *Nicole Sutherland* *Don Taylor* *Bill Unger* *Alexa van der Waall* *Paul van Wamelen* *Helena Verrill* *John Voight* *Mark Watkins* *Greg White*

Production Editors

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production

Claus Fieker *Allan Steel*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

9 Introduction to Aggregates **1**
- 9.1 Introduction . . . 1
- 9.2 Restrictions on Sets and Sequences . . . 1
- 9.3 Dual iterators in constructors . . . 5
- 9.4 Nested Aggregates . . . 5

10 Sets **7**
- 10.1 Introduction . . . 7
- 10.2 Creating Sets . . . 8
- 10.3 Power Sets . . . 13
- 10.4 Sets from Structures . . . 14
- 10.5 Accessing and Modifying Sets . . . 14
- 10.6 Operations on Sets . . . 19
- 10.7 Quantifiers . . . 21
- 10.8 Reduction and Iteration . . . 23

11 Sequences **26**
- 11.1 Introduction . . . 26
- 11.2 Creating Sequences . . . 27
- 11.3 Power Sequences . . . 29
- 11.4 Operators on Sequences . . . 29
- 11.5 Predicates on Sequences . . . 36
- 11.6 Recursion, Reduction, and Iteration . . . 37
- 11.7 Bibliography . . . 39

12 Tuples and Cartesian Products **40**
- 12.1 Introduction . . . 40
- 12.2 Cartesian Product Constructor and Functions . . . 40
- 12.3 Creating and Modifying Tuples . . . 41
- 12.4 Tuple Access Functions . . . 42
- 12.5 Equality . . . 42
- 12.6 Iteration . . . 42
- 12.7 Other Operations . . . 43

13 Lists **44**
- 13.1 Introduction . . . 44
- 13.2 Construction of Lists . . . 44
- 13.3 Creation of New Lists . . . 44
- 13.4 Access Functions . . . 45
- 13.5 Assignment Operator . . . 45
- 13.6 Iteration . . . 45

14 Associative Arrays **46**
- 14.1 Introduction . . . 46
- 14.2 Operations . . . 46
- 14.3 Iteration . . . 49

15 Coproducts **50**
15.1 Introduction . . . 50
15.2 Creation Functions . . . 50
15.3 Accessing Functions . . . 50
15.4 Retrieve . . . 51
15.5 Flattening . . . 51
15.6 Universal Map . . . 51

16 Records **52**
16.1 Introduction . . . 52
16.2 The Record Format Constructor . . . 52
16.3 Creating a Record . . . 53
16.4 Access and Modification Functions . . . 53

17 Mappings **55**
17.1 Introduction . . . 55
17.2 Creation Functions . . . 56
17.3 Operations on Mappings . . . 58
17.4 Images and Preimages . . . 59
17.5 Parents of Maps . . . 60

References **61**

Index **62**

Chapter 9

Introduction to Aggregates

9.1 Introduction

This part of the Handbook comprises seven chapters on aggregate objects in Magma as well as a chapter on maps. Sets, sequences, tuples, and lists are the four main types of aggregates, and each has its own specific purpose. *Sets* are used to collect objects that are elements of some common structure, and the most important operation is to test element membership. *Sequences* also contain objects of a common structure, but here the emphasis is on the ordering of the objects, and the most important operation is that of accessing (or modifying) elements at given positions. Sets will contain at most one copy of any element, whereas sequences may contain arbitrarily many copies of the same object.

Enumerated sets and sequences are of arbitrary but finite length and will store all elements explicitly (with the exception of arithmetic progressions), while *formal* sets and sequences may be infinite, and use a Boolean function to test element membership. *Indexed* sets are a hybrid form of sets allowing indexing like sequences.

Elements of *Cartesian products* of structures in Magma will be called *tuples*; they are of fixed length, and each coefficient must be in the corresponding structure of the defining Cartesian product. *Lists* are arbitrary finite ordered collections of objects of any type, and are mainly provided to the user to store assorted data to which access is not critical.

9.2 Restrictions on Sets and Sequences

Here we will explain the subtleties behind the mechanism dealing with enumerated sets and sequences and their universes and parents. In principle, every object e in Magma has some parent structure S such that $e \in S$; this structure can be used for type checking (are we allowed to apply function f to e?), algorithm look-up, etc. To avoid storing the structure with every element of a set or sequence and having to look up the structure of every element separately, only elements of a *common structure* are allowed in sets or sequences, and that common parent will only be stored once.

9.2.1 Universe of a Set or Sequence

This common structure is called the *universe* of the set or sequence. In the general constructors it may be specified up front to make clear what the universe for the set or sequence will be; the difference between the sets i and s in

```
> i := { IntegerRing() | 1, 2, 3 };
> s := { RationalField() | 1, 2, 3 };
```

lies entirely in their universes. The specification of the universe may be omitted if there is an obvious common overstructure for the elements. Thus the following provides a shorter way to create the set containing 1, 2, 3 and having the ring of integers as universe:

```
> i := { 1, 2, 3 };
```

Only empty sets and sequences that have been obtained directly from the constructions

```
> S := { };
> T := [ ];
```

do not have their universe defined – we will call them the *null* set or sequence. (There are two other ways in which empty sets and sequences arise: it is possible to create empty sets and sequences with a prescribed universe, using

```
> S := { U | };
> T := [ U | ];
```

and it may happen that a non-empty set/sequence becomes empty in the course of a computation. In both cases these empty objects have their universe defined and will not be *null*.) Usually (but not always; the exception will be explained below) the universe of a set or sequence is the parent for all its elements; thus the ring of integers is the parent of 2 in the set $i = \{1, 2, 3\}$, rather than that set itself. The universe is not static, and it is not necessarily the same structure as the parent of the elements *before* they were put in the set or sequence. To illustrate this point, suppose that we try to create a set containing integers and rational numbers, say $T = \{1, 2, 1/3\}$; then we run into trouble with the rule that the universe must be common for all elements in T. The way this problem is solved in Magma is by automatic coercion: The obvious universe for T is the field of rational numbers of which $1/3$ is already an element and into which any integer can be coerced in a natural way. Hence the assignment

```
> T := { 1, 2, 1/3 }
```

will result in a set with universe the field of rationals (which is also present when Magma is started up). Consequently, when we take the element 1 of the set T, it will have the rational field as its parent rather than the integer ring! It will now be clear that

```
> s := { 1/1, 2, 3 };
```

is a shorter way to specify the set of rational numbers 1, 2, 3 than the way we saw before, but in general it is preferable to declare the universe beforehand using the `{ U | }` notation. Of course

```
> T := { Integers() | 1, 2, 1/3 }
```

would result in an error because $1/3$ cannot be coerced into the ring of integers. So, usually not every element of a given structure can be coerced into another structure, and even if it can, it will not always be done automatically. The possible (automatic) coercions are listed in the descriptions of the various Magma modules. For instance, the table in the introductory chapter on rings shows that integers can be coerced automatically into the rational field. In general, every Magma structure is valid as a universe. This includes enumerated sets and sequences themselves; that is, it is possible to define a set or sequence whose elements are confined to be elements of a given set or sequence. So, for example,

```
> S := [ [ 1..10 ] | x^2+x+1 : x in { -3 .. 2 by 1 } ];
```

produces the sequence $[7, 3, 1, 1, 3, 7]$ of values of the polynomial $x^2 + x + 1$ for $x \in \mathbb{Z}$ with $-3 \leq x \leq 2$. However, an entry of S will in fact have the ring of integers as its parent (and *not* the sequence [1..10]), because the effect of the above assignment is that the values after the | are calculated and coerced into the universe, which is [1..10]; but coercing an element into a sequence or set means that it will in fact be coerced into the *universe* of that sequence/set, in this case the integers. So the main difference between the above assignment and

```
> T := [ Integers() | x^2+x+1 : x in { -3 .. 2 by 1} ];
```

is that in the first case it is checked that the resulting values y satisfy $1 \leq y \leq 10$, and an error would occur if this is violated:

```
> S := [ [ 1..10 ] | x^2+x+1 : x in { -3 .. 3 by 1} ];
```

leads to a run-time error. In general then, the parent of an element of a set or sequence will be the universe of the set or sequence, unless that universe is itself a set or sequence, in which case the parent will be the universe of this universe, and so on, until a non-set or sequence is encountered.

9.2.2 Modifying the Universe of a Set or Sequence

Once a (non-null) set or sequence S has been created, the universe has been defined. If one attempts to *modify* S (that is, to add elements, change entries etc. using a procedure that will not reassign the result to a new set or sequence), the universe will not be changed, and the modification will only be successful if the new element can be coerced into the current universe. Thus,

```
> Z := Integers();
> T := [ Z | 1, 2, 3/3 ];
> T[2] := 3/4;
```

will result in an error, because $3/4$ cannot be coerced into Z. The universe of a set or sequence S can be explicitly modified by creating a *parent* for S with the desired universe and using the `!` operator for the coercion; as we will see in the next subsection, such a parent can be created using the `PowerSet` and `PowerSequence` commands. Thus, for example, the set $\{1, 2\}$ can be made into a set of rationals as follows:

```
> I := { 1, 2 };
> P := PowerSet( RationalField() );
> J := P ! I;
```

The coercion will be successful if every element of the sequence can be coerced into the new universe, and it is *not* necessary that the old universe could be coerced completely into the new one: the set $\{3/3\}$ of rationals can be coerced into `PowerSet(Integers())`. As a consequence, the empty set (or sequence) with any universe can be coerced into the power set (power sequence) of any other universe. Binary functions on sets or sequences (like `join` or `cat`) can only be applied to sets and sequences that are *compatible*: the operation on S with universe A and T with universe B can only be performed if a common universe C can be found such that the elements of S and T are all elements of C. The compatibility conditions are dependent on the particular Magma module to which A and B belong (we refer to the corresponding chapters of this manual for further information) and do also apply to elements of $a \in A$ and $b \in B$ — that is, the compatibility conditions for S and T are the same as the ones that determine whether binary operations on $a \in A$ and $b \in B$ are allowed. For example, we are able to join a set of integers and a set of rationals:

```
> T := { 1, 2 } join { 1/3 };
```

for the same reason that we can do

```
> c := 1 + 1/3;
```

(automatic coercion for rings). The resulting set T will have the rationals as universe.

The basic rules for compatibility of two sets or sequences are then:

(1) every set/sequence is compatible with the null set/sequence (which has no universe defined (see above));

(2) two sets/sequences with the same universe are compatible;

(3) a set/sequence S with universe A is compatible with set/sequence T with universe B if the elements of A can be automatically coerced into B, or vice versa;

(4) more generally, a set/sequence S with universe A is also compatible with set/sequence T with universe B if Magma can automatically find an *over-structure* for the parents A and B (see below);

(5) nested sets and sequences are compatible only when they are of the same 'depth' and 'type' (that is, sets and sequences appear in exactly the same recursive order in both) and the universes are compatible.

The possibility of finding an overstructure C for the universe A and B of sets or sequences S and T (such that $A \subset C \supset B$), is again module-dependent. We refer the reader for details to the introductions of the chapters on the structures concerned, and we give some examples here; the next subsection contains the rules for parents of sets and sequences.

Perhaps the most common example of universes that are *not* compatible would be a prime finite field with the rationals, as not *every* rational can be coerced into the finite field, while Magma does not allow coercion from finite fields into the rationals in any event.

9.2.3 Parents of Sets and Sequences

The universe of a set or sequence S is the common parent for all its elements; but S itself is a Magma object as well, so it should have a parent too. The parent of a set is a *power set*: the set of all subsets of the universe of S. It can be created using the `PowerSet` function. Similarly, `PowerSequence(A)` creates the parent structure for a sequence of elements from the structure A – that is, the elements of `PowerSequence(A)` are all sequences of elements of A. The rules for finding a common overstructure for structures A and B, where either A or B is a set/sequence or the parent of a set/sequence, are as follows. (If neither A nor B is a set, sequence, or its parent we refer to the Part of this manual describing the operations on A and B.)

(1) The overstructure of A and B is the same as that of B and A.

(2) If A is the null set or sequence (empty, and no universe specified) the overstructure of A and B is B.

(3) If A is a set or sequence with universe U, the overstructure of A and B is the overstructure of U and B; in particular, the overstructure of A and A will be the universe U of A.

(4) If A is the parent of a set (a power set), then A and B can only have a common overstructure if B is also the parent of a set, in which case the overstructure is the power set of the overstructure of the universes U and V of A and B respectively. Likewise for sequences instead of sets.

We give two examples to illustrate rules (3) and (4). It is possible to create a set with a set as its universe:

```
> S := { { 1..100 } | x^3 : x in [ 1..3 ] };
```

If we wish to intersect this set with some set of integers, say the set of odd integers between 1 and 99

```
> T := { x : x in [ 1..99 ] | IsOdd(x) };
> W := S meet T;
```

then we can only do that if we can find a universe for W, which must be the common overstructure of the universe $U = \{1, 2, \ldots, 100\}$ of S and the universe 'ring of integers' of T. By rule (3) above, this overstructure of $U = \{1, 2, \ldots, 100\}$ will be the overstructure of the universe of U and the ring of integers; but the universe of U is the ring of integers (because it is the default for the set $\{1, 2, \ldots, 100\}$), and hence the overstructure we are looking for (and the universe for W) will be the ring of integers. For the second example we look at sequences of sequences:

```
> a := [ [ 1 ], [ 1, 2, 3 ] ];
> b := [ [ 2/3 ] ];
```

so a is a sequence of sequences of integers, and b is a sequence of sequences of rationals. If we wish to concatenate a and b,

```
> c := a cat b;
```

we will only succeed if we find a universe for c. This universe must be the common overstructure of the universes of a and b, which are the 'power sequence of the integers' and the 'power sequence of the

rationals' respectively. By rule (4), the overstructure of these two power sequences is the power sequence of the common overstructure of the rationals and the integers, which is the rationals themselves. Hence c will be a sequence of sequences of rationals, and the elements of a will have to be coerced.

9.3 Dual iterators in constructors

Most aggregate types support various constructors (as described in their specific sections), some of which use the syntax `x in S` to iterate over the elements of a structure S during the construction. If this structure S supports dual iteration, then the syntax `i -> x in S` is also allowed in such situations (and consequently both i and x may be used in the associated expressions). The meanings assigned to i and x during dual iteration will be explained in the sections for those types which support it.

9.4 Nested Aggregates

Enumerated sets and sequences can be arbitrarily nested (that is, one may create sets of sets, as well as sequences of sets etc.); tuples can also be nested and may be freely mixed with sets and sequences (as long as the proper Cartesian product parent can be created). Lists can be nested, and one may create lists of sets or sequences or tuples.

9.4.1 Multi-indexing

Since sequences (and lists) can be nested, assignment functions and mutation operators allow you to use *multi-indexing*, that is, one can use a multi-index $i_1, i_2, \ldots, i_r$ rather than a single i to reach r levels deep. Thus, for example, if $S = [\ [1,2],\ [2,3]\ ]$, instead of

```
> S[2][2] := 4;
```

one may use the multi-index $2, 2$ to obtain the same effect of changing the 3 into a 4:

```
> S[2,2] := 4;
```

All i_j in the multi-index $i_1, i_2, \ldots, i_r$ have to be greater than 0, and an error will also be flagged if any i_j indexes beyond the length at level j, that is, if $i_j > \#S[i_1, \ldots, i_{j-1}]$, (which means $i_1 > \#S$ for $j = 1$). There is one exception: the last index i_r is allowed to index beyond the current length of the sequence at level r if the multi-index is used on the left-hand side of an assignment, in which case any intermediate terms will be undefined. This generalizes the possibility to assign beyond the length of a 'flat' sequence. In the above example the following assignments are allowed:

```
> S[2,5] := 7;
```

(and the result will be $S = [\ [1,2],\ [2,3,\text{undef},\text{undef},7]\ ]$)

```
> S[4] := [7];
```

(and the result will be $S = [\ [1,2],\ [2,3],\ \text{undef},\ [7]\ ]$). But the following results in an error:

```
> S[4,1] := 7;
```

Finally we point out that multi-indexing should not be confused with the use of sequences as indexes to create subsequences. For example, to create a subsequence of $S = [5, 13, 17, 29]$ consisting of the second and third terms, one may use

```
> S := [ 5, 13, 17, 29 ];
> T := S[ [2, 3] ];
```

To obtain the second term of this subsequence one could have done:

```
> x := S[ [2, 3] ][2];
```

(so x now has the value $S[3] = 17$), but it would have been more efficient to index the indexing sequence, since it is rather expensive to build the subsequence [$S[2], S[3]$] first, so:

```
> x :=  S[ [2, 3][2] ];
```

has the same effect but is better (of course `x := S[3]` would be even better in this simple example.) To add to the confusion, it is possible to mix the above constructions for indexing, since one can use lists of sequences and indices for indexing; continuing our example, there is now a third way to do the same as above, using an indexing list that first takes out the subsequence consisting of the second and third terms and then extracts the second term of that:

```
> x :=  S[ [2, 3], 2 ];
```

Similarly, the construction

```
> X :=  S[ [2, 3], [2] ];
```

pulls out the subsequence consisting of the second term of the subsequence of terms two and three of S, in other words, this assigns the *sequence* consisting of the element 17, not just the element itself!

Chapter 10

Sets

10.1 Introduction

A *set* in Magma is a (usually unordered) collection of objects belonging to some common structure (called the *universe* of the set). There are three basic types of sets: *enumerated sets*, whose elements are all stored explicitly (with one exception, see below); *indexed sets*, which are restricted enumerated sets having a numbering on elements; and *multisets*, which are enumerated sets with possible repetition of elements. Enumerated sets, indexed sets, and multisets are always finite.

10.1.1 Enumerated Sets

Enumerated sets are finite, and can be specified in three basic ways (see also *Creating Sets*): by listing all elements; by an expression involving elements of some finite structure; and by an arithmetic progression. If an arithmetic progression is specified, the elements are not calculated explicitly until a modification of the set necessitates it; in all other cases all elements of the enumerated set are stored explicitly.

10.1.2 Indexed Sets

For some purposes it is useful to be able to access elements of a set through an index map, which numbers the elements of the set. For that purpose Magma has indexed sets, on which a very few basic set operations are allowed (element membership testing) as well as some sequence-like operations (such as accessing the i-th term, getting the index of an element, appending, and pruning).

10.1.3 Multisets

For some purposes it is useful to construct a set with some of its members repeated. For that purpose Magma has multisets, which take into account the repetition of members. The number of times an object x occurs in a multiset S is called the *multiplicity* of x in S. Magma has the $\widehat{\ }\widehat{\ }$ operator to specify a multiplicity: the expression `x^^ n` means the object x with multiplicity n. In the following, whenever any multiset constructor or function expects an element y, the expression $x \mathbin{\widehat{\ }\widehat{\ }} n$ may usually be used.

10.1.4 Compatibility

The binary operators for sets do not allow mixing of the three types of sets; functions are provided for making an enumerated set out of an indexed set or a multiset (and vice versa). The elements of enumerated sets are restricted, in the sense that either some universe must be specified upon creation, or Magma must be able to find such universe automatically. The rules for compatibility of elements and the way Magma deals with these universes are the same for sequences and sets, and are described in the previous chapter. The restrictions on indexed sets are the same as those for enumerated sets.

10.1.5 Notation

Certain expressions appearing in the sections below (possibly with subscripts) have a standard interpretation:

U the universe: any Magma structure;

E the carrier set for enumerated sets: any enumerated structure (it must be possible to loop over its elements — see the Introduction to this Part (Chapter *Introduction to Aggregates*));

x a free variable which successively takes the elements of E as its values;

P a Boolean expression that usually involves the variable(s) x, $x_1, \ldots, x_k$;

e an expression that also usually involves the variable(s) x, $x_1, \ldots, x_k$.

10.2 Creating Sets

The customary braces `{` and `}` are used to define enumerated sets. For indexed sets `{@` and `@}` are used. For multisets `{*` and `*}` are used.

10.2.1 The Enumerated Set Constructor

Enumerated sets can be constructed by expressions enclosed in braces, provided that the values of all expressions can be automatically coerced into some common structure, as outlined in the Introduction, (Chapter *Introduction to Aggregates*). All general constructors have an optional universe (U in the list below) up front, that allows the user to specify into which structure all terms of the sets should be coerced.

`{ }: Null -> Set`

The null set: an empty set that does not have its universe defined.

`{ U | }: Str -> Set`

The empty set with universe U.

`{ e1, e2, ..., en }: Elt, ..., Elt -> Set`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the set { $a_1, a_2, ..., a_n$ } of elements of U.

Example: Universe of an Enumerated Set `(ex-4076ed)`

We create a set by listing its elements explicitly.

```
> S := { (7^2+1)/5, (8^2+1)/5, (9^2-1)/5 };
> S;
{ 10, 13, 16 }
> Parent(S);
Set of subsets of Rational Field
```

Thus S was created as a set of rationals, because `/` on integers has a rational result. If one wishes to obtain a set of integers, one could specify the universe (or one could use `div`, or one could use `!` on every element to coerce it into the ring of integers):

```
> T := { Integers() | (7^2+1)/5, (8^2+1)/5, (9^2-1)/5 };
> T;
{ 10, 13, 16 }
> Parent(T);
Set of subsets of Integer Ring
```

`{ U | e1, e2, ..., en }: Str, Elt, ..., Elt -> Set`

Given a list of expressions $e_1, \ldots, e_n$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the set $\{a_1, a_2, ..., a_n\}$ of elements of U.

`{ e(x) : x in E | P(x) }`

Form the set of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter *Introduction to Aggregates*) (in particular, E must be a finite structure that can be enumerated). If $P(x)$ is true for every value of x in E, then the set constructor may be abbreviated to `{ e(x) : x in E }`.

`{ U | e(x) : x in E | P(x) }`

Form the set of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before. If P is always true, it may be omitted (including the |).

`{ e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) }`

The set consisting of those elements $e(x_1, \ldots, x_k)$, in some common structure, for which $x_1, \ldots, x_k$ in $E_1, \ldots, E_k$ have the property that $P(x_1, \ldots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter *Introduction to Aggregates*). Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to `x_i, x_{i+1} in E_i`. Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted (including the |).

`{ U | e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) }`

As in the previous entry, the set consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true, is formed, as a set of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are elements of or coercible into U). Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example: Almost Fermat Cubes (ex-8d723f)

Now that Fermat's last theorem has been proven, it may be of interest to find integers that almost satisfy $x^n + y^n = z^n$. In this example we find all $2 < x, y, z < 1000$ such that $x^3 + y^3 = z^3 + 1$. First we build a set of cubes, then two sets of pairs for which the sum of cubes differs from a cube by 1. Note that we build a *set* rather than a sequence of cubes because we only need fast membership testing. Also note that the resulting sets of pairs do not have their elements in the order in which they were found.

```
> cubes := { Integers() | x^3 : x in [1..1000] };
> plus := { <a, b> : a in [2..1000], b in [2..1000] | \\
>    b ge a and (a^3+b^3-1) in cubes };
> plus;
{
       < 9, 10 >,
       < 135, 235 >
       < 334, 438 >,
       < 73, 144 >,
       < 64, 94 >,
       < 244, 729 >
}
```

Note that we spend a lot of time cubing integers this way. A more efficient approach is shown in a subsequent example.

10.2.2 The Indexed Set Constructor

The creation of indexed sets is similar to that of enumerated sets.

{@ @}: Null -> SetIndx

The null set: an empty indexed set that does not have its universe defined.

{@ U | @}: Str -> SetIndx

The empty indexed set with universe U.

{@ e_1, e_2, ..., e_n @}: Elt, ..., Elt -> SetIndx

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the indexed set $Q = \{ a_1, a_2, ..., a_n \}$ of elements of U.

{@ U | e_1, e_2, ..., e_m @}: Str, Elt, ..., Elt -> SetIndx

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the indexed set $Q = \{a_1, a_2, ..., a_n \}$ of elements of U.

{@ e(x) : x in E | P(x) @}

Form the indexed set of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter *Introduction to Aggregates*) (in particular, E must be a finite structure that can be enumerated). If P is always true, it may be omitted (including the |).

{@ U | e(x) : x in E | P(x) @}

Form the indexed set of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before. If P is always true, it may be omitted (including the |).

{@ e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) @}

The indexed set consisting of those elements $e(x_1, \ldots, x_k)$ (in some common structure), for which $x_1, \ldots, x_k$ in $E_1 \times \ldots \times E_k$ have the property that $P(x_1, \ldots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter *Introduction to Aggregates*). Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to `x_i, x_{i+1} in E_i`. Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted.

{@ U | e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)@}

As in the previous entry, the indexed set consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as an indexed set of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are elements of or coercible into U). Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example: Almost Fermat Cubes with Indexed Sets (ex-c67e42)

In the previous example we found pairs x, y such that $x^3 + y^3$ differs by one from some cube z^3. Using indexed sets it is somewhat easier to retrieve the integer z as well. We give a small example. Note also that it is beneficial to know here that evaluation of expressions proceeds left to right.

```
> cubes := {@ Integers() | z^3 : z in [1..25] @};
> plus := { <x, y, z> : x in [-10..10], y in [-10..10], z in [1..25] |
>    y ge x and Abs(x) gt 1 and Abs(y) gt 1 and (x^3+y^3-1) in cubes
>    and (x^3+y^3-1) eq cubes[z] };
> plus;
{ <-6, 9, 8>, <9, 10, 12>, <-8, 9, 6> }
```

10.2.3 The Multiset Constructor

The creation of multisets is similar to that of enumerated sets. An important difference is that repetitions are significant and the operator ^^ (mentioned above) may be used to specify the multiplicity of an element.

`{* *}: Null -> SetMulti`

The null set: an empty multiset that does not have its universe defined.

`{* U | *}: Str -> SetMulti`

The empty multiset with universe U.

`{* e1, e2, ..., en *}: Elt, ..., Elt -> SetMulti`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the multiset $Q = \{* \, a_1, a_2, ..., a_n \, *\}$ of elements of U.

`{* U | e1, e2, ..., em *}: Str, Elt, ..., Elt -> SetMulti`

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the multiset $Q = \{* \, a_1, a_2, ..., a_n \, *\}$ of elements of U.

`{* e(x) : x in E | P(x) *}`

Form the multiset of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter *Introduction to Aggregates*) (in particular, E must be a finite structure that can be enumerated). If P is always true, it may be omitted (including the |).

`{* U | e(x) : x in E | P(x) *}`

Form the multiset of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before. If P is always true, it may be omitted (including the |).

`{* e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) *}`

The multiset consisting of those elements $e(x_1, \ldots, x_k)$ (in some common structure), for which $x_1, \ldots, x_k$ in $E_1 \times \ldots \times E_k$ have the property that $P(x_1, \ldots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter *Introduction to Aggregates*). Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to `x_i, x_{i+1} in E_i`. Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted.

`{* U | e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) *}`

As in the previous entry, the multiset consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as a multiset of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are elements of or coercible into U). Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example: Multiset Constructors `(ex-ca72ec)`

Here we demonstrate the use of the multiset constructors.

```
> M := {* 1, 1, 1, 3, 5 *};
> M;
{* 1^^3, 3, 5 *}
> M := {* 1^^4, 2^^5, 1/2^^3 *};
> M;
> // Count frequency of digits in first 1000 digits of pi:
> pi := Pi(RealField(1001));
> dec1000 := Round(10^1000*(pi-3));
> I := IntegerToString(dec1000);
```

(continues on next page)

(continued from previous page)

```
> F := {* I[i]: i in [1 .. #I] *};
> F;
{* 7^^95, 3^^102, 6^^94, 2^^103, 9^^106, 5^^97,
1^^116, 8^^101, 4^^93, 0^^93 *}
> for i := 0 to 9 do i, Multiplicity(F, IntegerToString(i)); end for;
0 93
1 116
2 103
3 102
4 93
5 97
6 94
7 95
8 101
9 106
```

10.2.4 The Arithmetic Progression Constructors

Some special constructors exist to create and store enumerated sets of integers in arithmetic progression efficiently. This only works for arithmetic progressions of elements of the ring of integers.

`{ i..j }: RngIntElt, RngIntElt -> Set`

`{ U | i..j }: Str, RngIntElt, RngIntElt -> SetIndx`

The enumerated set whose elements form the arithmetic progression $i, i+1, i+2, \ldots, j$, where i and j are (expressions defining) integers. If j is less than i then the empty set will be created. The only universe U that is legal here is the ring of integers.

`{ i .. j by k }: RngIntElt, RngIntElt, RngIntElt -> Set`

`{ U | i .. j by k }: Str, RngIntElt, RngIntElt, RngIntElt -> Set`

The enumerated set consisting of the integers forming the arithmetic progression $i, i+k, i+2*k, \ldots, j$, where i, j, and k are (expressions defining) integers (but $k \neq 0$). If k is positive then the last element in the progression will be the greatest integer of the form $i + n * k$ that is less than or equal to j. If j is less than i, the empty set will be constructed. If k is negative then the last element in the progression will be the least integer of the form $i+n*k$ that is greater than or equal to j. If j is greater than i, the empty set will be constructed. As for the previous constructor, only the ring of integers is allowed as a legal universe U.

Example: Arithmetic Progressions in Sets (ex-91fc28)

It is possible to use the arithmetic progression constructors to save typing in the creation of 'arithmetic progressions' of elements of other structures than the ring of integers, but it should be kept in mind that the result will not be treated especially efficiently like the integer case. Here is the 'wrong' way, as well as two correct ways to create a set of 10 finite field elements.

```
> S := { FiniteField(13) | 1..10 };
Runtime error in { .. }: Invalid set universe
> S := { FiniteField(13) | x : x in { 1..10 } };
> S;
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
> G := PowerSet(FiniteField(13));
> S := G ! { 1..10 };
> S;
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

10.3 Power Sets

The `PowerSet` constructor returns a structure comprising the subsets of a given structure R; it is mainly useful as a parent for other set and sequence constructors. The only operations that are allowed on power sets are printing, testing element membership, and coercion into the power set (see the examples below).

`PowerSet(R): Str -> PowSetEnum`

The structure comprising all enumerated subsets of structure R.

`PowerIndexedSet(R): Str -> PowSetIndx`

The structure comprising all indexed subsets of structure R.

`PowerMultiset(R): Str -> PowSetMulti`

The structure consisting of all submultisets of the structure R.

`S in P: SetEnum, PowSetEnum -> BoolElt`

Returns `true` if enumerated set S is in the power set P, that is, if all elements of the set S are contained in or coercible into R, where P is the power set of R; `false` otherwise.

`S in P: SetIndx, PowSetIndx -> BoolElt`

Returns `true` if indexed set S is in the power set P, that is, if all elements of the set S are contained in or coercible into R, where P is the power set of R; `false` otherwise.

`S in P: SetMulti, PowSetMulti -> BoolElt`

Returns `true` if multiset S is in the power set P, that is, if all elements of the set S are contained in or coercible into R, where P is the power set of R; `false` otherwise.

`P ! S: PowSetEnum, SetEnum -> SetEnum`

Return a set with universe R consisting of the elements of the set S, where P is the power set of R. An error results if not all elements of S can be coerced into R.

`P ! S: PowSetIndx, SetIndx -> SetIndx`

Return an indexed set with universe R consisting of the elements of the set S, where P is the power indexed set of R. An error results if not all elements of S can be coerced into R.

`P ! S: PowSetMulti, SetMulti -> SetMulti`

Return a multiset with universe R consisting of the elements of the set S, where P is the power multiset of R. An error results if not all elements of S can be coerced into R.

Example: Power Sets `(ex-0340bf)`

```
> S := { 1 .. 10 };
> P := PowerSet(S);
> P;
Set of subsets of { 1 .. 10 }
> F := { 6/3, 12/4 };
> F in P;
true
> G := P ! F;
> Parent(F);
Set of subsets of Rational Field
> Parent(G);
Set of subsets of { 1 .. 10 }
```

10.3.1 The Cartesian Product Constructors

Using `car< >` and `CartesianProduct( )`, it is possible to create the Cartesian product of sets (or, in fact, of any combination of structures), but the result will be of type 'Cartesian product' rather than set, and the elements are tuples — we refer the reader to Chapter *Tuples and Cartesian Products* for details.

10.4 Sets from Structures

`Set(M): Str -> SetEnum`

Given a finite structure that allows explicit enumeration of its elements, return the set containing its elements (having M as its universe).

10.5 Accessing and Modifying Sets

Enumerated sets can be modified by inserting or removing elements. Indexed sets allow some sequence-like operators for modification and access.

10.5.1 Accessing Sets and their Associated Structures

`# R: SetIndx -> RngIntElt`

`# R: SetEnum -> RngIntElt`

`# R: SetMulti -> RngIntElt`

Cardinality of the enumerated, indexed, or multi- set R. Note that for a multiset, repetitions are significant, so the result may be greater than the underlying set.

`Category(S): Any -> Cat`

`Type(S): Any -> Cat`

The category of the object S. For a set this will be one of `SetEnum`, `SetIndx`, or `SetMulti`. For a power set the type is one of `PowSetEnum`, `PowSetIndx`, `PowSetMulti`.

`Parent(R): Set -> Str`

Returns the parent structure of R, that is, the structure consisting of all (enumerated) subsets of the universe of R.

`Universe(R): Set -> Str`

Returns the 'universe' of the (enumerated or indexed or multi-) set R, that is, the common structure to which all elements of the set belong. An error is signalled when R is the null set.

`Index(S, x): SetIndx, Elt -> RngIntElt`

`Position(S, x): SetIndx, Elt -> RngIntElt`

Given an indexed set S, and an element x, returns the index i such that $S[i] = x$ if such index exists, or return 0 if x is not in S. If x is not in the universe of S, an attempt will be made to coerce it; an error occurs if this fails.

`S[i]: SetIndx, RngIntElt -> Elt`

Return the i-th entry of indexed set S. If $i < 1$ or $i > \#S$ an error occurs. Note that indexing is *not* allowed on the left hand side.

`S[I]: SetIndx, [RngIntElt] -> SetIndx`

The indexed set $\{S[i_1], \ldots, S[i_r]\}$ consisting of terms selected from the indexed set S, according to the terms of the integer sequence I. If any term of I lies outside the range 1 to $\#S$, then an error results. If I is the empty sequence, then the empty set with universe the same as that of S is returned.

Example: Indexed Set of Sets (ex-3d41ac)

We build an indexed set of sets to illustrate the use of the above functions.

```
> B := {@ { i : i in [1..k] } :  k in [1..5] @};
> B;
{@
   { 1 },
   { 1, 2 },
```

(continues on next page)

(continued from previous page)

```
   { 1, 2, 3 },
   { 1, 2, 3, 4 },
   { 1, 2, 3, 4, 5 },
@}
> #B;
5
> Universe(B);
Set of subsets of Integer Ring
> Parent(B);
Set of indexed subsets of Set of subsets of Integer Ring
> Category(B);
SetIndx
> Index(B, { 2, 1});
2
> #B[2];
2
> Universe(B[2]);
Integer Ring
```

10.5.2 Selecting Elements of Sets

Most finite structures in Magma, including enumerated sets, allow one to obtain a random element using `Random`. There is an alternative (and often preferable) option for enumerated sets in the `random{ }` constructor. This makes it possible to choose a random element of the set without generating the whole set first. Likewise, `rep{ }` is an alternative to the general `Rep` function returning a representative element of a structure, having the advantage of aborting the construction of the set as soon as one element has been found. Here, E will again be an enumerable structure, that is, a structure that allows enumeration of its elements (see the Appendix for an exhaustive list). Note that `random{ e(x) : x in E | P(x)}` does *not* return a random element of the set of values $e(x)$, but rather a value of $e(x)$ for a random x in E which satisfies P (and mutatis mutandis for `rep`). See the subsection on Notation in the Introduction (Chapter *Introduction to Aggregates*) for conventions regarding e, x, E, P.

`Random(R): SetIndx -> Elt`

`Random(R): SetEnum -> Elt`

A random element chosen from the enumerated, indexed, or multi- set R. Every element has an equal probability of being chosen for enumerated or indexed sets, and a weighted probability in proportion to its multiplicity for multisets. Successive invocations of the function will result in independently chosen elements being returned as the value of the function. If R is empty an error occurs.

`random{ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression P, return the value of the expression $e(y)$ for a randomly chosen element y of E for which $P(y)$ is true. The expression P may be omitted if it is always true.

`random{ e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) }`

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, return the value of the expression $e(y_1, \cdots, y_k)$ for a randomly chosen element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, for which $P(y_1, \ldots, y_k)$ is true. The expression P may be omitted if it is always true. If successive structures E_i and E_{i+1} are identical, then the abbreviation `x_i, x_{i+1} in E_i` may be used.

Example: Random Primitive Elements `(ex-1b553f)`

Here are two ways to find a 'random' primitive element for a finite field.

```
> p := 10007;
> F := FiniteField(p);
```

(continues on next page)

(continued from previous page)

```
> proots := {  z : z in F | IsPrimitive(z) };
> #proots;
5002
> Random(proots);
5279
```

This way, a set of 5002 elements is built (and primitivity is checked for all elements of F), and a random choice is made. Alternatively, we use `random`.

```
> random{ x : x in F | IsPrimitive(x) };
4263
```

In this case random elements in F are chosen until one is found that is primitive. Since almost half of F's elements are primitive, only very few primitivity tests will be done before success occurs.

`Representative(R): SetIndx -> Elt`

`Rep(R): SetIndx -> Elt`

`Representative(R): SetEnum -> Elt`

`Rep(R): SetEnum -> Elt`

An arbitrary element chosen from the enumerated, indexed, or multi- set R.

`ExtractRep(~R, ~r): SetEnum, Elt`

Assigns an arbitrary element chosen from the enumerated set R to r, and removes it from R. Thus the set R is modified, as well as the element r. An error occurs if R is empty.

`rep{ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression P, return the value of the expression `e(y)` for the first element y of E for which $P(y)$ is true. If $P(x)$ is false for every element of E, an error will occur.

`rep{ e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) }`

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, return the value of the expression $e(y_1, \cdots, y_k)$ for the first element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, for which $P(y_1, \ldots, y_k)$ is true. An error occurs if no element of $E_1 \times \cdots \times E_k$ satisfies P. The expression P may be omitted if it is always true. If successive structures E_i and E_{i+1} are identical, then the abbreviation `x_i, x_{i+1} in E_i` may be used.

Example: Cubes Found with ExtractRep `(ex-e07b22)`

As an illustration of the use of `ExtractRep`, we modify an earlier example, and find cubes satisfying $x^3 + y^3 = z^3 - 1$ (with $x, y, z \leq 1000$).

```
> cubes := { Integers() | x^3 : x in [1..1000] };
> cc := cubes;
> min := { };
> while not IsEmpty(cc) do
>    ExtractRep(~cc, ~a);
>    for b in cc do
>       if a+b+1 in cubes then
>          min join:= { <a, b> };
>       end if;
>    end for;
> end while;
> { < Iroot(x[1], 3), Iroot(x[2], 3) > : x in min };
{ <138, 135>, <823, 566>, <426, 372>, <242, 720>,
      <138, 71>, <426, 486>, <6, 8> }
```

Note that instead of taking cubes over again, we only have to take cube roots in the last line (on the small resulting set) once.

`Minimum(S): SetIndx -> Elt, RngIntElt`

`Min(S): SetIndx -> Elt, RngIntElt`

`Minimum(S): SetEnum -> Elt`

`Min(S): SetEnum -> Elt`

`Minimum(S): SetMulti -> Elt`

`Min(S): SetMulti -> Elt`

Given a non-empty enumerated, indexed, or multi- set S, such that `lt` and `eq` are defined on the universe of S, this function returns the minimum of the elements of S. If S is an indexed set, the position of the minimum is also returned.

`Maximum(S): SetIndx -> Elt, RngIntElt`

`Max(S): SetIndx -> Elt, RngIntElt`

`Maximum(S): SetEnum -> Elt`

`Max(S): SetEnum -> Elt`

`Maximum(S): SetMulti -> Elt`

`Max(S): SetMulti -> Elt`

Given a non-empty enumerated, indexed, or multi- set S, such that `lt` and `eq` are defined on the universe of S, this function returns the maximum of the elements of S. If S is an indexed set, the position of the maximum is also returned.

`Hash(x): Elt -> RngIntElt`

Given a Magma object x which can be placed in a set, return the hash value of x used by the set machinery. This is a fixed but arbitrary non-negative integer (whose maximum value is the maximum value of a C unsigned long on the particular machine). The crucial property is that if x and y are objects and x equals y then the hash values of x and y are equal (even if x and y have different internal structures). Thus one could implement sets manually if desired by the use of this function.

10.5.3 Modifying Sets

`Include(~S, x): SetEnum, Elt`

`Include(S, x): SetEnum, Elt -> SetEnum`

`Include(~S, x): SetIndx, Elt`

`Include(S, x): SetIndx, Elt -> SetIndx`

`Include(~S, x): SetMulti, Elt`

`Include(S, x): SetMulti, Elt -> SetMulti`

Create the enumerated, indexed, or multi- set obtained by putting the element x in S (S is unchanged if S is not a multiset and x is already in S). If S is an indexed set, the element will be appended at the end. If S is a multiset, the multiplicity of x will be increased accordingly. If x is not in the universe of S, an attempt will be made to coerce it; an error occurs if this fails. There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the set S will not be copied.

`Exclude(~S, x): SetEnum, Elt`

`Exclude(S, x): SetEnum, Elt -> SetEnum`

`Exclude(~S, x): SetMulti, Elt`

`Exclude(S, x): SetMulti, Elt -> SetMulti`

Create a new set by removing the element x from S. If S is an enumerated set, nothing happens if x is not in S. If S is a multiset, the multiplicity of x will be decreased accordingly. If x is not in the universe of S, an attempt will be made to coerce it; an error occurs if this fails. There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the set S will not be copied.

ChangeUniverse(~S, V): SetEnum, Str

ChangeUniverse(S, V): SetEnum, Str -> SetEnum

ChangeUniverse(~S, V): SetIndx, Str

ChangeUniverse(S, V): SetIndx, Str -> SetIndx

ChangeUniverse(~S, V): SetMulti, Str

ChangeUniverse(S, V): SetMulti, Str -> SetMulti

Given an enumerated, indexed, or multi- set S with universe U and a structure V which contains U, construct a new set of the same type which consists of the elements of S coerced into V. There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the set S will not be copied.

CanChangeUniverse(S, V): SetEnum, Str -> Bool, SeqEnum

CanChangeUniverse(S, V): SetIndx, Str -> Bool, SeqEnum

CanChangeUniverse(S, V): SetMulti, Str -> Bool, SeqEnum

Given an enumerated, indexed, or multi- set S with universe U and a structure V which contains U, attempt to construct a new set T of the same type which consists of the elements of S coerced into V; if successful, return `true` and T, otherwise return `false`.

Example: Sums of Two Cubes with Include (ex-708b1a)

This example uses `Include` and `Exclude` to find a set (if it exists) of cubes of integers such that the elements of a given set R can be expressed as the sum of two of those.

```
> R := { 218, 271, 511 };
> x := 0;
> cubes := { 0 };
> while not IsEmpty(R) do
>    x +:= 1;
>    c := x^3;
>    Include(~cubes, c);
>    Include(~cubes, -c);
>    for z in cubes do
>        Exclude(~R, z+c);
>        Exclude(~R, z-c);
>    end for;
> end while;
```

We did not record how the elements of R were obtained as sums of a pair of cubes. For that, the following suffices.

```
> R := { 218, 271, 511 }; // it has been emptied !
> { { x, y } : x, y in cubes | x+y in R };
{
    { -729, 1000 },
    { -125, 343 },
    { -1, 512 },
}
```

SetToIndexedSet(E): SetEnum -> SetIndx

Given an enumerated set E, this function returns an indexed set with the same elements (and universe) as E.

IndexedSetToSet(S): SetIndx -> SetEnum

Isetset(S): SetIndx -> SetEnum

Given an indexed set S, this function returns an enumerated set with the same elements (and universe) as S.

`IndexedSetToSequence(S): SetIndx -> SeqEnum`

`Isetseq(S): SetIndx -> SeqEnum`

Given an indexed set S, this function returns a sequence with the same elements (and universe) as S.

`MultisetToSet(S): SetMulti -> SetEnum`

Given a multiset S, this function returns an enumerated set with the same elements (and universe) as S.

`SetToMultiset(E): SetEnum -> SetMulti`

Given an enumerated set E, this function returns a multiset with the same elements (and universe) as E.

`SequenceToMultiset(Q): SeqEnum -> SetMulti`

Given an enumerated sequence E, this function returns a multiset with the same elements (and universe) as E.

10.6 Operations on Sets

10.6.1 Boolean Functions and Operators

As explained in the Introduction (Chapter *Introduction to Aggregates*), when elements are taken out of a set their parent will be the universe of the set (or, if the universe is itself a set, the universe of the universe, etc.); in particular, the set itself is not the parent. Hence equality testing on set elements is in fact equality testing between two elements of certain algebraic structures, and the sets are irrelevant. We only list the (in)equality operator for convenience here. Element membership testing is of critical importance for all types of sets. Testing whether or not R is a subset of S can be done when R and S are enumerated, indexed, or multisets.

`IsNull(R): SetEnum -> BoolElt`

`IsNull(R): SetIndx -> BoolElt`

`IsNull(R): SetMulti -> BoolElt`

Returns `true` if and only if the enumerated, indexed, or multi- set R is empty and does not have its universe defined.

`IsEmpty(R): SetEnum -> BoolElt`

`IsEmpty(R): SetIndx -> BoolElt`

`IsEmpty(R): SetMulti -> BoolElt`

Returns `true` if and only if the enumerated, indexed, or multi- set R is empty.

`x eq y: Elt, Elt -> BoolElt`

Given an element x of a set R with universe U and an element y of a set S with universe V, where a common overstructure W can be found with $U \subset W \supset V$ (see the Introduction (Chapter *Introduction to Aggregates*) for details on overstructures), return `true` if and only if x and y are equal as elements of W.

`x ne y: Elt, Elt -> BoolElt`

Given an element x of a set R with universe U and an element y of a set S with universe V, where a common overstructure W can be found with $U \subset W \supset V$ (see the Introduction (Chapter *Introduction to Aggregates*) for details on overstructures), return `true` if and only if x and y are distinct as elements of W.

`x in R: Elt, Set -> BoolElt`

Returns `true` if and only if the element x is a member of the set R. If x is not an element of the universe U of R, it is attempted to coerce x into U; if this fails, an error occurs.

`x notin R: Elt, Set -> BoolElt`

Returns `true` if and only if the element x is not a member of the set R. If x is not an element of the parent structure U of R, it is attempted to coerce x into U; if this fails, an error occurs.

`R subset S: SetEnum, Set -> BoolElt`

`R subset S: SetIndx, Set -> BoolElt`

`R subset S: SetMulti, Set -> BoolElt`

Returns `true` if the enumerated, indexed, or multi- set R is a subset of the set S, `false` otherwise. For multisets, if an element x of R has multiplicity n in R, the multiplicity of x in S must be at least n. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`R notsubset S: SetEnum, Set -> BoolElt`

`R notsubset S: SetIndx, Set -> BoolElt`

`R notsubset S: SetMulti, Set -> BoolElt`

Returns `true` if the enumerated, indexed, or multi- set R is a not a subset of the set S, `false` otherwise. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`R eq S: Set, Set -> BoolElt`

Returns `true` if and only if R and S are identical sets, where R and S are enumerated, indexed, or multi- sets For indexed sets, the index function is irrelevant for deciding equality. For multisets, matching multiplicities must also be equal. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`R ne S: Set, Set -> BoolElt`

Returns `true` if and only if R and S are distinct sets, where R and S are enumerated indexed, or multi- sets. For indexed sets, the index function is irrelevant for deciding equality. For multisets, matching multiplicities must also be equal. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`IsDisjoint(R, S): SetEnum, SetEnum -> BoolElt`

`IsDisjoint(R, S): SetIndx, SetIndx -> BoolElt`

`IsDisjoint(R, S): SetMulti, SetMulti -> BoolElt`

Returns `true` iff the enumerated, indexed, or multi- sets R and S are disjoint. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

10.6.2 Binary Set Operators

For each of the following operators, R and S are sets of the same type (both enumerated, both indexed, or both multisets), and an error occurs unless the universes of R and S are compatible, as defined in the Introduction to this Part (Chapter *Introduction to Aggregates*).

`R join S: SetEnum, SetEnum -> SetEnum`

`R join S: SetIndx, SetIndx -> SetIndx`

`R join S: SetMulti, SetMulti -> SetMulti`

Union of the sets R and S (see above for the restrictions on R and S). For multisets, matching multiplicities are added in the union.

`R meet S: SetEnum, SetEnum -> SetEnum`

`R meet S: SetIndx, SetIndx -> SetIndx`

`R meet S: SetMulti, SetMulti -> SetMulti`

Intersection of the sets R and S (see above for the restrictions on R and S). For multisets, the minimum of matching multiplicities is stored in the intersection.

`R diff S: SetEnum, SetEnum -> SetEnum`

`R diff S: SetIndx, SetIndx -> SetIndx`

`R diff S: SetMulti, SetMulti -> SetMulti`

Difference of the sets R and S. i.e., the set consisting of those elements of R which are not members of S (see above for the restrictions on R and S). For multisets, the difference contains any elements of R remaining after removing the corresponding elements of S the appropriate number of times.

`R sdiff S: SetEnum, SetEnum -> SetEnum`

`R sdiff S: SetIndx, SetIndx -> SetIndx`

`R sdiff S: SetMulti, SetMulti -> SetMulti`

Symmetric difference of the sets R and S. i.e., the set consisting of those elements which are members of either R or S but not both (see above for the restrictions on R and S). Alternatively, it is the union of the difference of R with S and the difference of S with R.

Example: Binary Set Operators (ex-527afc)

```
> R := { 1, 2, 3 };
> S := { 1, 1/2, 1/3 };
> R join S;
{ 1/3, 1/2, 1, 2, 3 }
> R meet S;
{ 1 }
> R diff S;
{ 2, 3 }
> S diff R;
{ 1/3, 1/2 }
> R sdiff S;
{ 1/3, 1/2, 2, 3 }
```

10.6.3 Other Set Operations

`Multiplicity(S, x): SetMulti, Elt -> RngIntElt`

Return the multiplicity in multiset S of element x. If x is not in S, zero is returned.

`Multiplicities(S): SetMulti -> SeqEnum`

Returns the sequence of multiplicities of distinct elements in the multiset S. The order is the same as the internal enumeration order of the elements.

`Subsets(S): SetEnum -> SetEnum`

The set of all subsets of S.

`Subsets(S, k): SetEnum, RngIntElt -> SetEnum`

The set of subsets of S of size k. If k is larger than the cardinality of S then the result will be empty.

`RandomSubset(S, k): SetEnum, RngIntElt -> SetEnum`

A random subset of S of size k. It is an error if k is larger than the size of S.

`Multisets(S, k): SetEnum, RngIntElt -> SetEnum`

The set of multisets consisting of k not necessarily distinct elements of S.

`Subsequences(S, k): SetEnum, RngIntElt -> SetEnum`

The set of sequences of length k with elements from S.

`Permutations(S): SetEnum -> SetEnum;`

The set of permutations (stored as sequences) of the elements of S.

`Permutations(S, k): SetEnum, RngIntElt -> SetEnum;`

The set of permutations (stored as sequences) of each of the subsets of S of cardinality k.

10.7 Quantifiers

To test whether some enumerated set is empty or not, one may use the `IsEmpty` function. However, to use `IsEmpty`, the set has to be created in full first. The existential quantifier `exists` enables one to do the test and abort the construction of the set as soon as an element is found; moreover, the element found will be assigned to a variable. Likewise, `forall` enables one to abort the construction of the set

as soon as an element not satisfying a certain property is encountered. Note that `exists(t){ e(x) : x in E | P(x) }` is *not* designed to return true if an element of the set of values $e(x)$ satisfies P, but rather if there is an $x \in E$ satisfying $P(x)$ (in which case $e(x)$ is assigned to t). For the notation used here, see the beginning of this chapter.

`exists(t){ e(x): x in E | P(x) }`

`exists(t`$_1$`, ..., t`$_r$`){ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression $P(x)$, the Boolean value `true` is returned if E contains at least one element x for which $P(x)$ is true. If $P(x)$ is not true for any element x of E, then the Boolean value `false` is returned. Moreover, if $P(x)$ is found to be true for the element y, say, of E, then in the first form of the exists expression, variable t will be assigned the value of the expression $e(y)$. If $P(x)$ is never true for an element of E, t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x)$ is never true. The clause `(t)` may be omitted entirely. The expression P may be omitted if it is always true.

`exists(t){ e(x`$_1$`, ..., x`$_k$`): x`$_1$` in E`$_1$`, ..., x`$_k$` in E`$_k$` | P(x`$_1$`, ..., x`$_k$`)}`

`exists(t`$_1$`, ..., t`$_r$`){ e(x`$_1$`, ..., x`$_k$`) : x`$_1$` in E`$_1$`, ..., x`$_k$` in E`$_k$` | P }`

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, the Boolean value `true` is returned if there is an element $< y_1, \ldots, y_k >$ in the Cartesian product $E_1 \times \cdots \times E_k$, such that $P(y_1, \ldots, y_k)$ is true. If $P(x_1, \ldots, x_k)$ is not true for any element $(y_1, \ldots, y_k)$ of $E_1 \times \cdots \times E_k$, then the Boolean value `false` is returned. Moreover, if $P(x_1, \ldots, x_k)$ is found to be true for the element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \cdots, y_k)$. If $P(x_1, \ldots, x_k)$ is never true for an element of $E_1 \times \cdots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y_1, \cdots, y_k)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x_1, \ldots, x_k)$ is never true. The clause `(t)` may be omitted entirely. The expression P may be omitted if it is always true. If successive structures E_i and E_{i+1} are identical, then the abbreviation `x_i, x_{i+1} in E_i` may be used.

Example: Sums of Cubes with Exists (ex-eb9af3)

As a variation on an earlier example, we check whether or not some integers can be written as sums of cubes (less than 10^3 in absolute value):

```
> exists(t){ <x, y> : x, y in [ t^3 : t in [-10..10] ] | x + y eq 218 };
true
> t;
<-125, 343>
> exists(t){ <x, y> : x, y in [ t^3 : t in [1..10] ] | x + y eq 218 };
false
> t;
>> t;
   ^
User error: Identifier 't' has not been declared
```

`forall(t){ e(x) : x in E | P(x) }`

`forall(t`$_1$`, ..., t`$_r$`){ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression $P(x)$, the Boolean value `true` is returned if $P(x)$ is true for every element x of E. If $P(x)$ is not true for at least one element x of E, then the Boolean value `false` is returned. Moreover, if $P(x)$ is found to be false for the element y, say, of E, then in the first form of the exists expression, variable t will be assigned the value of the expression $e(y)$. If $P(x)$ is true for every element of E, t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x)$ is always true. The clause `(t)` may be omitted entirely. The expression P may be omitted if it is always true.

`forall(t){ e(x1, ..., xk): x1 in E1, ..., xk in Ek | P(x1, ..., xk)}`

`forall(t1, ..., tr){ e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P }`

Given sets $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, the Boolean value `true` is returned if $P(x_1, \ldots, x_k)$ is true for every element $(x_1, \ldots, x_k)$ in the Cartesian product $E_1 \times \cdots \times E_k$. If $P(x_1, \ldots, x_k)$ fails to be true for some element $(y_1, \ldots, y_k)$ of $E_1 \times \cdots \times E_k$, then the Boolean value `false` is returned. Moreover, if $P(x_1, \ldots, x_k)$ is false for the element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \cdots, y_k)$. If $P(x_1, \ldots, x_k)$ is true for every element of $E_1 \times \cdots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y_1, \cdots, y_k)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x_1, \ldots, x_k)$ is never true. The clause `(t)` may be omitted entirely. The expression P may be omitted if it is always true. If successive structures E_i and E_{i+1} are identical, then the abbreviation `x_i, x_{i+1} in E_i` may be used.

Example: Nested Quantifiers and Sums of Squares (ex-8bbcd9)

This example shows that `forall` and `exists` may be nested. It is well known that every prime that is 1 modulo 4 can be written as the sum of two squares, but not every integer m congruent to 1 modulo 4 can. In this example we explore for small m whether perhaps $m \pm \epsilon$ (with $|\epsilon| \leq 1$) is always a sum of squares.

```
> forall(u){ m : m in [5..1000 by 4] |
>       exists{ <x, y, z> : x, y in [0..30], z in [-1, 0, 1] |
>          x^2+y^2+z eq m } };
false
> u;
77
```

10.8 Reduction and Iteration

10.8.1 Reduction

Instead of using a loop to apply the same binary associative operator to all elements of a set, it is in certain cases possible to use the *reduction operator* `&`.

`& o S: Op, SetEnum -> Elt`

`& o S: Op, SetIndx -> Elt`

`& o S: Op, SetMulti -> Elt`

Given an enumerated, indexed, or multiset $S = \{ a_1, a_2, \ldots, a_n\}$ of elements belonging to an algebraic structure U, and an (associative) operator $\circ : U \times U \to U$, form the element $a_{i_1} \circ a_{i_2} \circ a_{i_3} \circ \ldots \circ a_{i_n}$, for some permutation $i_1, \ldots, i_n$ of $1, \ldots, n$. Currently, the following operators may be used to reduce such sets: `+, *, and, or, join,` and `meet`. An error will occur if the operator is not defined on U. If S contains a single element a, then the value returned is a. If S is the null set (empty and no universe specified) or S is empty with universe U (and the operation is defined in U), then the result (or error) depends on the operation and upon U. The following table defines the return value:

```
             EMPTY     NULL
     &+       U!0      error
     &*       U!1      error
     &and    true      true
     &or     false     false
     &join   empty     null
     &meet   error     error
```

Warning: since the reduction may take place in an arbitrary order on the arguments $a_1, \ldots, a_n$, the result is not unambiguously defined if the operation is not commutative on the arguments!

Example: Subsets of Given Cardinality by Reduction (ex-50129c)

The function `choose` defined below takes a set S and an integer k as input, and produces a set of all subsets of S with cardinality k.

```
> function choose(S, k)
>    if k eq 0 then
>       return { { } };
>    else
>       return &join{{  s join { x} : s in choose(S diff { x}, k-1) } : x in S};
>    end if;
> end function;
```

So, for example:

```
>  S := { 1, 2, 3, 4 };
> choose(S, 2);
{
       { 1, 3 },
       { 1, 4 },
       { 2, 4 },
       { 2, 3 },
       { 1, 2 },
       { 3, 4 }
}
```

Try to guess what happens if $k < 0$.

10.8.2 Iteration

The iteration operator `in` can be used in loops or in the set and sequence constructors to enumerate elements of a set. Enumerated, indexed, and multisets allow enumeration of their elements. For indexed sets the enumeration will occur according to the order given by the indexing.

`x in S`

Enumerate the elements of an enumerated, indexed, or multiset S. Note that if S is a multiset then each element will be returned a number of times equal to its multiplicity in S.

`i -> x in S`

Enumerate the elements x of an indexed set S, together with the corresponding index value i.

`x -> v in M`

Enumerate the elements x of the underlying set of the multiset M, together with the corresponding multiplicity v. Note that elements with multiplicity larger than 1 will still only be returned once by this version of the iterator.

Example: Letter Frequencies in a String (ex-00bab7)

We display a simple frequency count of a certain string.

```
> M := Multiset(Eltseq("hello world"));
> for letter -> count in M do
>    if count eq 1 then
>       are := "is"; s := "";
>    else
>       are := "are"; s := "s";
>    end if;
>    printf "There %o %o '%o'%o\n", are, count, letter, s;
> end for;
There is 1 ' '
```

(continues on next page)

(continued from previous page)

```
There is 1 'e'
There are 3 'l's
There is 1 'w'
There is 1 'h'
There are 2 'o's
There is 1 'd'
There is 1 'r'
```

Note the difference between the use of the dual iterator above and the single form below.

```
> for letter in M do
>    printf "'%o'\n", letter;
> end for;
' '
'e'
'l'
'l'
'l'
'w'
'h'
'o'
'o'
'd'
'r'
```

Chapter 11

Sequences

11.1 Introduction

A *sequence* in Magma is a linearly ordered collection of objects belonging to some common structure (called the *universe* of the sequence). There are two types of sequence: *enumerated sequences*, of which the elements are all stored explicitly (with one exception, see below); and *formal sequences*, of which elements are stored implicitly by means of a predicate that allows for testing membership. In particular, enumerated sequences are always finite, and formal sequences are allowed to be infinite. In this chapter a *sequence* will be either a formal or an enumerated sequence.

11.1.1 Enumerated Sequences

An *enumerated sequence of length* l is an array of indefinite length of which only finitely many terms – including the l-th term, but no term of bigger index — have been defined to be elements of some common structure. Such a sequence is called *complete* if all of the terms (from index 1 up to the length l) are defined. In practice the length of an enumerated sequence must be less than 2^{30}. Incomplete enumerated sequences are allowed as a convenience for the programmer in building complete enumerated sequences. Some sequence functions require their arguments to be complete; if that is the case, it is mentioned explicitly in the description below. However, all functions using sequences in *other* Magma modules always assume that a sequence that is passed in as an argument is complete. Note that the following line converts a possibly incomplete sequence S into a complete sequence T:

```
T := [ s : s in S ];
```

because the enumeration using the `in` operator simply ignores undefined terms. Enumerated sequences of *Booleans* are highly optimized (stored as bit-vectors).

11.1.2 Formal Sequences

A formal sequence consists of elements of some range set on which a certain predicate assumes the value 'true'. There is only a very limited number of operations that can be performed on them.

11.1.3 Compatibility

The binary operators for sequences do not allow mixing of the formal and enumerated sequence types (so one cannot take the concatenation of an enumerated sequence and a formal sequence, for example); but it is easy to convert an enumerated sequence into a formal sequence – see the section on binary operators below. By the limitation on their construction formal sequences can only contain elements from one structure in Magma. The elements of enumerated sequences are also restricted, in the sense that either some common structure must be specified upon creation, or Magma must be able to find such universe automatically. The rules for compatibility of elements and the way Magma deals with these parents is the same for sequences and sets, and is outlined in Chapter *Introduction to Aggregates*.

11.2 Creating Sequences

Square brackets are used for the definition of enumerated sequences; formal sequences are delimited by the composite brackets `[!` and `!]`. Certain expressions appearing below (possibly with subscripts) have the standard interpretation:

U the universe: any Magma structure;

E the range set for enumerated sequences: any enumerated structure (it must be possible to loop over its elements – see the Introduction to this Part);

F the range set for formal sequences: any structure for which membership testing using `in` is defined – see the Introduction to this Part);

x a free variable which successively takes the elements of E (or F in the formal case) as its values;

P a Boolean expression that usually involves the variable(s) x, $x_1, \ldots, x_k$;

e an expression that also usually involves the variable(s) x, $x_1, \ldots, x_k$.

11.2.1 The Formal Sequence Constructor

The formal sequence constructor has the following fixed format (the expressions appearing in the construct are defined above):

`[! x in F | P(x) !]`

Create the formal sequence consisting of the subsequence of elements x of F for which $P(x)$ is true. If $P(x)$ is true for every element of F, the sequence constructor may be abbreviated to `[! x in F !]`

11.2.2 The Enumerated Sequence Constructor

Sequences can be constructed by expressions enclosed in square brackets, provided that the values of all expressions can be automatically coerced into some common structure, as outlined in the Introduction. All general constructors have the universe U optionally up front, which allows the user to specify into which structure all terms of the sequences should be coerced.

`[ ]: Null -> ESeqEnum`

The null sequence (empty, and no universe specified).

`[ U | ]: Str -> SeqEnum`

The empty sequence with universe U.

`[` e_1`,` e_2`, ..., ` e_n `]: Elt, ..., Elt -> SeqEnum`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the sequence $Q = [a_1, a_2, ..., a_n\ ]$ of elements of U. As for multisets, one may use the expression `x^^ n` to specify the object x with multiplicity n: this is simply interpreted to mean x repeated n times (i.e., no internal compaction of the repetition is done).

`[ U |` e_1`,` e_2`, ...,` e_m `]: Str, Elt, ..., Elt -> SeqEnum`

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the sequence $Q = [a_1, a_2, ..., a_n\ ]$ of elements of U.

`[ e(x) : x in E | P(x) ]`

Form the sequence of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given at the beginning of this section. If $P(x)$ is true for every element of E, the sequence constructor may be abbreviated to `[ e(x) : x in E ]` .

`[ U | e(x) : x in E | P(x) ]`

Form the sequence of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as above.

`[ e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) ]`

The sequence consisting of those elements $e(x_1, \ldots, x_k)$, in some common structure, for which $x_1, \ldots, x_k$ in $E_1, \ldots, E_k$ have the property that $P(x_1, \ldots, x_k)$ is true. The expressions appearing in this construct have the interpretation given at the beginning of this section. Note that if two successive ranges E_i and E_{i+1} are identical, then the specification of the ranges for x_i and x_{i+1} may be abbreviated to `x_i, x_{i+1} in E_i`. Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted.

`[ U | e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) ]`

As in the previous entry, the sequence consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as a sequence of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are coercible into U).

11.2.3 The Arithmetic Progression Constructors

Since enumerated sequences of integers arise so often, there are a few special constructors to create and handle them efficiently in case the entries are in arithmetic progression. The universe must be the ring of integers. Some effort is made to preserve the special way of storing arithmetic progressions under sequence operations.

`[ i..j ]: RngIntElt, RngIntElt -> SeqEnum`

`[ U | i..j ]: Str, RngIntElt, RngIntElt -> SeqEnum`

The enumerated sequence of integers whose elements form the arithmetic progression $i, i+1, i+2, \ldots, j$, where i and j are (expressions defining) arbitrary integers. If j is less than i then the empty sequence of integers will be created. The universe U, if it is specified, has to be the ring of integers; any other universe will lead to an error.

`[ i .. j by k ]: RngIntElt, RngIntElt, RngIntElt -> SeqEnum`

`[ U | i .. j by k ]: Str, RngIntElt, RngIntElt, RngIntElt -> SeqEnum`

The enumerated sequence consisting of the integers forming the arithmetic progression $i, i+k, i+2*k, \ldots, j$, where i, j, and k are (expressions defining) arbitrary integers (but $k \neq 0$). If k is positive then the last element in the progression will be the greatest integer of the form $i + n * k$ that is less than or equal to j; if j is less than i, the empty sequence of integers will be constructed. If k is negative then the last element in the progression will be the least integer of the form $i + n * k$ that is greater than or equal to j; if j is greater than i, the empty sequence of integers will be constructed. The universe U, if it is specified, has to be the ring of integers; any other universe will lead to an error.

Example: Arithmetic Progressions in Sequences (ex-264a9e)

As in the case of sets, it is possible to use the arithmetic progression constructors to save some typing in the creation of sequences of elements of rings other than the ring of integers, but the result will not be treated especially efficiently.

```
> s := [ IntegerRing(200) | x : x in [ 25..125 ] ];
```

11.2.4 Literal Sequences

A literal sequence is an enumerated sequence all of whose terms are from the same structure and all of these are 'typed in' literally. The sole purpose of literal sequences is to load certain enumerated sequences very fast and very space-efficiently; this is only useful when reading in very large sequences (all of whose elements must have been specified literally, that is, not as some expression other than a literal), but then it may save a lot of time. The result will be an enumerated sequence, that is, not distinguished in any way from other such sequences. At present, only literal sequences of integers are supported.

`\[ m1, ..., mn ]: RngIntElt, ..., RngIntElt -> [ RngIntElt ]`

Given a succession of literal integers $m_1, \ldots, m_n$, build the enumerated sequence $[m_1, \ldots, m_n]$, in a time and space efficient way.

11.3 Power Sequences

The `PowerSequence` constructor returns a structure comprising the enumerated sequences of a given structure R; it is mainly useful as a parent for other set and sequence constructors. The only operations that are allowed on power sequences are printing, testing element membership, and coercion into the power sequence (see the examples below).

`PowerSequence(R): Str -> PowSeqEnum`

The structure comprising all enumerated sequences of elements of structure R. If R itself is a sequence (or set) then the power structure of its universe is returned.

`S in P: SeqEnum, PowSeqEnum -> BoolElt`

Returns `true` if enumerated sequence S is in the power sequence P, that is, if all elements of the sequence S are contained in or coercible into R, where P is the power sequence of R; `false` otherwise.

`P ! S: PowSeqEnum, SeqEnum -> SeqEnum`

Return a sequence with universe R consisting of the entries of the enumerated sequence S, where P is the power sequence of R. An error results if not all elements of S can be coerced into R.

Example: Power Sequences (ex-e8f60b)

```
> S := [ 1 .. 10 ];
> P := PowerSequence(S);
> P;
Set of sequences over [ 1 .. 10 ]
> F := [ 6/3, 12/4 ];
> F in P;
true
> G := P ! F;
> Parent(F);
Set of sequences over Rational Field
> Parent(G);
Set of sequences over [ 1 .. 10 ]
```

11.4 Operators on Sequences

This section lists functions for obtaining information about existing sequences, for modifying sequences, and for creating sequences from others. Most of these operators only apply to enumerated sequences.

11.4.1 Access Functions

`# S: SeqEnum -> RngIntElt`

Returns the length of the enumerated sequence S, which is the index of the last term of S whose value is defined. The length of the empty sequence is zero.

`Parent(S): SeqEnum -> Str`

Returns the parent structure for a sequence S, that is, the structure consisting of all (enumerated) sequences over the universe of S.

`Universe(S): SeqEnum -> Str`

Returns the 'universe' of the sequence S, that is, the common structure to which all elements of the sequence belong. This universe may itself be a set or sequence. An error is signalled when S is the null sequence.

`S[i]: SeqEnum, RngIntElt -> Elt`

The i-th term s_i of the sequence S. If $i \leq 0$, or $i > \#S + 1$, or $S[i]$ is not defined, then an error results. Here i is allowed to be a multi-index (see Introduction for the interpretation). This can be used as the left hand side of an assignment: `S[i] := x` redefines the i-th term of the sequence S to be x. If $i \leq 0$, then an error results. If $i > n$, then the sequence $[s_1, \ldots, s_n, s_{n+1}, \ldots, s_{i-1}, x]$ replaces S, where $s_{n+1}, \ldots, s_{i-1}$ are all undefined. Here i is allowed to be a multi-index. An error occurs if x cannot be coerced into the universe of S.

11.4.2 Selection Operators on Enumerated Sequences

Here, S denotes an enumerated sequence $[s_1, \ldots, s_n]$. Further, i and j are integers or multi-indices (see Introduction).

`S[I]: SeqEnum, [RngIntElt] -> SeqEnum`

The sequence $[s_{i_1}, \ldots, s_{i_r}]$ consisting of terms selected from the sequence S, according to the terms of the integer sequence I. If any term of I lies outside the range 1 to $\#S$, then an error results. If I is the empty sequence, then the empty sequence with universe the same as that of S is returned. The effect of `T := S[I]` differs from that of `T := [ S[i] : i in I ]`: if in the first case an undefined entry occurs for $i \in I$ between 1 and $\#S$ it will be copied over; in the second such undefined entries will lead to an error.

`Minimum(S): SeqEnum -> Elt, RngIntElt`

`Min(S): SeqEnum -> Elt, RngIntElt`

Given a non-empty, complete enumerated sequence S such that `lt` and `eq` are defined on the universe of S, this function returns two values: a minimal element s in S, as well as the first position i such that $s = S[i]$.

`Maximum(S): SeqEnum -> Elt, RngIntElt`

`Max(S): SeqEnum -> Elt, RngIntElt`

Given a non-empty, complete enumerated sequence S such that `gt` and `eq` are defined on the universe of S, this function returns two values: a maximal element s in S, as well as the first position i such that $s = S[i]$.

`Index(S, x): SeqEnum, Elt -> RngIntElt`

`Index(S, x, f): SeqEnum, Elt, RngIntElt -> RngIntElt`

`Position(S, x): SeqEnum, Elt -> RngIntElt`

`Position(S, x, f): SeqEnum, Elt, RngIntElt -> RngIntElt`

Returns either the position of the first occurrence of x in the sequence S, or zero if S does not contain x. The second variants of each function starts the search at position f. This can save time in second (and subsequent) searches for the same entry further on. If no occurrence of x in S from position f onwards is found, then zero is returned.

`Representative(R): SeqEnum -> Elt`

`Rep(R): SeqEnum -> Elt`

An (arbitrary) element chosen from the enumerated sequence R

`Random(R): SeqEnum -> Elt`

A random element chosen from the enumerated sequence R. Every element has an equal probability of being chosen. Successive invocations of the function will result in independently chosen elements being returned as the value of the function. If R is empty an error occurs.

`Explode(R): SeqEnum -> List`

Given an enumerated sequence R of length r this function returns the r entries of the sequence (in order).

`Eltseq(R): SeqEnum -> SeqEnum`

The enumerated sequence R itself. This function is just included for completeness.

11.4.3 Modifying Enumerated Sequences

The operations given here are available as both procedures and functions. In the procedure version, the given sequence is destructively modified 'in place'. This is very efficient, since it is not necessary to make a copy of the sequence. In the function version, the given sequence is not changed, but a modified version of it is returned. This is more suitable if the old sequence is still required. Some of the functions also return useful but non-obvious values. Here, S denotes an enumerated sequence, and x an element of some structure V. The modifications involving S and x will only be successful if x can be coerced into the universe of S; an error occurs if this fails. (See the Introduction to this Part).

`Append(~S, x): SeqEnum, Elt`

`Append(S, x): SeqEnum, Elt -> SeqEnum`

Create an enumerated sequence by adding the object x to the end of S, i.e., the enumerated sequence $[s_1, \dots s_n, x]$. There are two versions of this: a procedure, where S is replaced by the appended sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Exclude(~S, x): SeqEnum, Elt`

`Exclude(S, x): SeqEnum, Elt -> SeqEnum`

Create an enumerated sequence obtained by removing the first occurrence of the object x from S, i.e., the sequence $[s_1, \dots\ s_{i-1},\ s_{i+1},\ \dots,\ s_n]$, where s_i is the first term of S that is equal to x. If x is not in S then this is just S. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Include(~S, x): SeqEnum, Elt`

`Include(S, x): SeqEnum, Elt -> SeqEnum`

Create a sequence by adding the object x to the end of S, provided that no term of S is equal to x. Thus, if x does not occur in S, the enumerated sequence $[s_1, \dots, s_n, x]$ is created. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Insert(~S, i, x): SeqEnum, RngIntElt, Elt`

`Insert(S, i, x): SeqEnum, RngIntElt, Elt -> SeqEnum`

Create the sequence formed by inserting the object x at position i in S and moving the terms $S[i], \dots, S[n]$ down one place, i.e., the enumerated sequence $[s_1, \dots\ s_{i-1},\ x,\ s_i,\ \dots,\ s_n]$. Note that i may be bigger than the length n of S, in which case the new length of S will be i, and the entries $S[n+1], \dots, S[i-1]$ will be undefined. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Insert(~S, k, m, T): SeqEnum, RngIntElt, RngIntElt, SeqEnum`

`Insert(S, k, m, T): SeqEnum, RngIntElt, RngIntElt, SeqEnum -> SeqEnum`

Create the sequence $[s_1, \dots,\ s_{k-1},\ t_1,\ \dots,\ t_l,\ s_{m+1},\ \dots,\ s_n]$. If $k \leq 0$ or $k > m+1$, then an error results. If $k = m+1$ then the terms of T will be inserted into S immediately before the term s_k. If $k > n$, then the sequence $[s_1, \dots, s_n, s_{n+1}, \dots, s_{k-1}, t_1, \dots, t_l]$ is created, where $s_{n+1}, \dots, s_{k-1}$ are all undefined. In the case where T is the empty sequence, terms $s_k, \dots, s_m$ are deleted from S. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Prune(~S): SeqEnum`

`Prune(S): SeqEnum -> SeqEnum`

Create the enumerated sequence formed by removing the last term of the sequence S, i.e., the sequence $[s_1, \ldots, s_{n-1}]$. An error occurs if S is empty. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Remove(~S, i): SeqEnum, RngIntElt`

`Remove(S, i): SeqEnum, RngIntElt -> SeqEnum`

Create the enumerated sequence formed by removing the i-th term from S, i.e., the sequence $[s_1, \ldots s_{i-1}, s_{i+1}, \ldots, s_n]$. An error occurs if $i < 1$ or $i > n$. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Reverse(~S): SeqEnum`

`Reverse(S): SeqEnum -> SeqEnum`

Create the enumerated sequence formed by reversing the order of the terms in the complete enumerated sequence S, i.e., the sequence $[s_n, \ldots, s_1]$. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Rotate(~S, p): SeqEnum, RngIntElt`

`Rotate(S, p): SeqEnum, RngIntElt -> SeqEnum`

Given a complete sequence S and an integer p, create the enumerated sequence formed by cyclically rotating the terms of the sequence p terms: if p is positive, rotation will be to the right; if p is negative, S is cyclically rotated $-p$ terms to the left; if p is zero nothing happens. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Sort(~S): SeqEnum`

`Sort(S): SeqEnum -> SeqEnum`

Given a complete enumerated sequence S whose terms belong to a structure on which `lt` and `eq` are defined, create the enumerated sequence formed by (quick-)sorting the terms of S into increasing order. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`Sort(~S, C): SeqEnum, UserProgram`

`Sort(~S, C, ~p): SeqEnum, UserProgram, GrpPermElt`

`Sort(S, C): SeqEnum, UserProgram -> SeqEnum`

Given a complete enumerated sequence S and a comparison function C which compares elements of S, create the enumerated sequence formed by sorting the terms of S into increasing order with respect to C. The comparison function C must take two arguments and return an integer less than, equal to, or greater than 0 according to whether the first argument is less than, equal to, or greater than the second argument (e.g.: `func<x, y | x - y>`). There are three versions of this: a procedure, where S is replaced by the new sequence, a procedure, where S is replaced by the new sequence and the corresponding permutation p is set, and a function, which returns the new sequence and the corresponding permutation. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`ParallelSort(~S, ~T): SeqEnum, SeqEnum`

Given a complete enumerated sequence S, sorts it in place and simultaneously sorts T in the same order. That is, whenever the sorting process would swap the two elements `S[i]` and `S[j]` then the two elements `T[i]` and `T[j]` are also swapped.

Undefine(~S, i): SeqEnum, RngIntElt

Undefine(S, i): SeqEnum, RngIntElt -> SeqEnum

Create the sequence which is the same as the enumerated sequence S but with the i-th term of S undefined; i may be bigger than $\#S$, but $i \leq 0$ produces an error. There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

ChangeUniverse(S, V): SeqEnum, Str

ChangeUniverse(S, V): SeqEnum, Str -> SeqEnum

Given a sequence S with universe U and a structure V which contains U, construct a sequence which consists of the elements of S coerced into V.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

CanChangeUniverse(S, V): SeqEnum, Str -> Bool, SeqEnum

Given a sequence S with universe U and a structure V which contains U, attempt to construct a sequence T which consists of the elements of S coerced into V; if successful, return `true` and T, otherwise return `false`.

Example: Farey Series (ex-67d3e0)

We present three ways to obtain the Farey series F_n of degree n. The Farey series F_n of degree n consists of all rational numbers with denominator less than or equal to n, in order of magnitude. Since we will need numerator and denominator often, we first abbreviate those functions.

```
> D := Denominator;
> N := Numerator;
```

The first method calculates the entries in order. It uses the fact that for any three consecutive Farey fractions $\frac{p}{q}$, $\frac{p'}{q'}$, $\frac{p''}{q''}$ of degree n:

$$p'' = \lfloor \frac{q+n}{q'} \rfloor p' - p, \quad q'' = \lfloor \frac{q+n}{q'} \rfloor q' - q.$$

```
> farey := function(n)
>    f := [ RationalField() | 0, 1/n ];
>    p := 0;
>    q := 1;
>    while p/q lt 1 do
>       p := ( D(f[#f-1]) + n) div D(f[#f]) * N(f[#f])  - N(f[#f-1]);
>       q := ( D(f[#f-1]) + n) div D(f[#f]) * D(f[#f])  - D(f[#f-1]);
>       Append(~f, p/q);
>    end while;
>    return f;
> end function;
```

The second method calculates the Farey series recursively. It uses the property that F_n may be obtained from F_{n-1} by inserting a new fraction (namely $\frac{p+p'}{q+q'}$) between any two consecutive rationals $\frac{p}{q}$ and $\frac{p'}{q'}$ in F_{n-1} for which $q + q'$ equals n.

```
> function farey(n)
>    if n eq 1 then
>       return [RationalField() | 0, 1 ];
>    else
```

(continues on next page)

(continued from previous page)

```
>       f := farey(n-1);
>       i := 0;
>       while i lt #f-1 do
>          i +:= 1;
>          if D(f[i]) + D(f[i+1]) eq n then
>             Insert( ~f, i+1, (N(f[i]) + N(f[i+1]))/(D(f[i]) + D(f[i+1])));
>          end if;
>       end while;
>       return f;
>    end if;
> end function;
```

The third method is very straightforward, and uses `Sort` and `Setseq` (defined above).

```
> farey := func< n |
>               Sort(Setseq({ a/b : a in { 0..n}, b in { 1..n} | a le b }))>;
> farey(6);
[ 0, 1/6, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 1 ]
```

11.4.4 Creating New Enumerated Sequences from Existing Ones

`S cat T: SeqEnum, SeqEnum -> SeqEnum`

The enumerated sequence formed by concatenating the terms of S with the terms of T, i.e. the sequence $[s_1, \ldots, s_n, t_1, \ldots, t_m]$. If the universes of S and T are different, an attempt to find a common overstructure is made; if this fails an error results (see the Introduction).

`S cat:= T: SeqEnum, SeqEnum`

Mutation assignment: change S to be the concatenation of S and T. Functionally equivalent to `S := S cat T`. If the universes of S and T are different, an attempt to find a common overstructure is made; if this fails an error results (see the Introduction).

`Partition(S, p): SeqEnum, RngIntElt -> SeqEnum(SeqEnum)`

Given a complete non-empty sequence S as well as an integer p that divides the length n of S, construct the sequence whose terms are the sequences formed by taking p terms of S at a time.

`Partition(S, P): SeqEnum, [RngIntElt] -> SeqEnum(SeqEnum)`

Given a complete non-empty sequence S as well as a complete sequence of positive integers P, such that the sum of the entries of P equals the length of S, construct the sequence whose terms are the sequences formed by taking $P[i]$ terms of S, for $i = 1, \ldots, \#P$.

`Setseq(S): SetEnum -> SeqEnum`

`SetToSequence(S): SetEnum -> SeqEnum`

Given a set S, construct a sequence whose terms are the elements of S taken in some arbitrary order.

`Seqset(S): SeqEnum -> SetEnum`

`SequenceToSet(S): SeqEnum -> SetEnum`

Given a sequence S, create a set whose elements are the distinct terms of S.

Example: Egyptian Fractions (ex-1c4c84)

The following example illustrates several of the access, creation, and modification operations on sequences. Given a rational number r, this function returns a sequence of different integers d_i such that $r = \sum 1/d_i$ [Bee93].

```
> egyptian := function(r)
>       n := Numerator(r);
```

(continues on next page)

(continued from previous page)

```
>       d := Denominator(r);
>       s := [d : i in [1..n]];
>       t := { d};
>       i := 2;
>       while i le #s do
>              c := s[i];
>              if c in t then
>                     Remove(~s, i);
>                     s cat:= [c+1, c*(c+1)];
>              else
>                     t join:= { c};
>                     i := i+1;
>              end if;
>       end while;
>       return s;
> end function;
```

Note that the result may be rather larger than necessary:

```
> e := egyptian(11/13);
> // Check the result!
> &+[1/d : d in e];
11/13
> #e;
2047
> #IntegerToString(Maximum(e));
1158
```

while instead of this sequence of 2047 integers, the biggest of the entries having 1158 decimal digits, the following equation also holds:

$$\frac{1}{3}+\frac{1}{4}+\frac{1}{6}+\frac{1}{12}+\frac{1}{78}=\frac{11}{13}.$$

Operations on Sequences of Booleans

The following operations work pointwise on sequences of booleans of equal length.

`And(S, T): [ BoolElt ], [ BoolElt ] -> [BoolElt]`

`And(~S, T): [ BoolElt ], [ BoolElt ]`

The sequence whose ith entry is the logical and of the ith entries of S and T. The result is placed in S if it is given by reference (~).

`Or(S, T): [ BoolElt ], [ BoolElt ] -> [ BoolElt ]`

`Or(~S, T): [ BoolElt ], [ BoolElt ]`

The sequence whose ith entry is the logical or of the ith entries of S and T. The result is placed in S if it is given by reference.

`Xor(S, T): [ BoolElt ], [ BoolElt ] -> [ BoolElt ]`

`Xor(~S, T): [ BoolElt], [ BoolElt ]`

The sequence whose ith entry is the logical xor of the ith entries of S and T. The result is placed in S if it is given by reference.

`Not(S): [ BoolElt ] -> [ BoolElt ]`

`Not(~S): [ BoolElt ]`

The sequence whose ith entry is the logical not of the ith entry of S. The result is placed in S if it is given by reference.

11.5 Predicates on Sequences

Boolean valued operators and functions on enumerated sequences exist to test whether entries are defined (see previous section), to test for membership and containment, and to compare sequences with respect to an ordering on its entries. On formal sequences, only element membership can be tested.

`IsComplete(S): SeqEnum -> BoolElt`

Boolean valued function, returning `true` if and only if each of the terms $S[i]$ for $1 \leq i \leq \#S$ is defined, for an enumerated sequence S.

`IsDefined(S, i): SeqEnum, RngIntElt -> BoolElt`

Given an enumerated sequence S and an index i, this returns `true` if and only if $S[i]$ is defined. (Hence the result is `false` if $i > \#S$, but an error results if $i < 1$.) Note that the index i is allowed to be a multi-index; if $i = [i_1, \ldots, i_r]$ is a multi-index and $i_j > \#S[i_1, \ldots, i_{j-1}]$ the function returns false, but if S is s levels deep and $r > s$ while $i_j \leq \#S[i_1, \ldots, i_{j-1}]$ for $1 \leq j \leq s$, then an error occurs.

`IsEmpty(S): SeqEnum -> BoolElt`

Boolean valued function, returning `true` if and only if the enumerated sequence S is empty.

`IsNull(S): SeqEnum -> BoolElt`

Boolean valued function, returning `true` if and only if the enumerated sequence S is empty and its universe is undefined, `false` otherwise.

11.5.1 Membership Testing

Here, S and T denote sequences. The element x is always assumed to be compatible with S.

`x in S: Elt, SeqEnum -> BoolElt`

Returns `true` if the object x occurs as a term of the enumerated or formal sequence S, `false` otherwise. If x is not in the universe of S, coercion is attempted. If that fails, an error results.

`x notin S: Elt, SeqEnum -> BoolElt`

Returns `true` if the object x does not occur as a term of the enumerated or formal sequence S, `false` otherwise. If x is not in the universe of S, coercion is attempted. If that fails, an error results.

`IsSubsequence(S, T): SeqEnum, SeqEnum -> BoolElt`

`IsSubsequence(S, T: Kind := option): SeqEnum, SeqEnum, MonStgElt -> BoolElt`

`Kind: MonStgElt`	`Default: "Consecutive"`

Returns `true` if the enumerated sequence S appears as a subsequence of consecutive elements of the enumerated sequence T, `false` otherwise. By changing the default value `"Consecutive"` of the parameter `Kind` to `"Sequential"` or to `"Setwise"`, this returns `true` if and only if the elements of S appear in order (but not necessarily consecutively) in T, or if and only if all elements of S appear as elements of T; so in the latter case the test is merely whether the set of elements of S is contained in the set of elements of T. If the universes of S and T are not the same, coercion is attempted.

`S eq T: SeqEnum, SeqEnum -> BoolElt`

Returns `true` if the enumerated sequences S and T are equal, `false` otherwise. If the universes of S and T are not the same, coercion is attempted.

`S ne T: SeqEnum, SeqEnum -> BoolElt`

Returns `true` if the enumerated sequences S and T are not equal, `false` otherwise. If the universes of S and T are not the same, coercion is attempted.

11.5.2 Testing Order Relations

Here, S and T denote complete enumerated sequences with universe U and V respectively, such that a common overstructure W for U and V can be found (as outlined in the Introduction), and such that on W an ordering on the elements is defined allowing the Magma operators `eq` ($=$), `le` ($\leq$), `lt` ($<$), `gt` ($>$), and `ge` ($\geq$) to be invoked on its elements.

With these comparison operators the *lexicographical* ordering is used to order complete enumerated sequences. Sequences S and T are equal (`S eq T`) if and only if they have the same length and all terms are the same. A sequence S precedes T (`S lt T`) in the ordering imposed by that of the terms if at the first index i where S and T differ then $S[i] < T[i]$. If the length of T exceeds that of S and S and T agree in all places up to the length of S, then `S lt T` is true also. In all other cases where $S \neq T$ one has `S gt T`.

`S lt T: SeqEnum, SeqEnum -> BoolElt`

Returns `true` if the sequence S precedes the sequence T under the ordering induced from S, `false` otherwise. Thus, `true` is returned if and only if either $S[k] < T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#S$ and $\#S < \#T$.

`S le T: SeqEnum, SeqEnum -> BoolElt`

Returns `true` if the sequence S either precedes the sequence T, under the ordering induced from S, or is equal to T, `false` otherwise. Thus, `true` is returned if and only if either $S[k] < T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#S$ and $\#S \leq \#T$.

`S ge T: SeqEnum, SeqEnum -> BoolElt`

Returns `true` if the sequence S either comes after the sequence T, under the ordering induced from S, or is equal to T, `false` otherwise. Thus, `true` is returned if and only if either $S[k] > T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#T$ and $\#S \geq \#T$.

`S gt T: SeqEnum, SeqEnum -> BoolElt`

Returns `true` if the sequence S comes after the sequence T under the ordering induced from S, `false` otherwise. Thus, `true` is returned if and only if either $S[k] > T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#T$ and $\#S > \#T$.

11.6 Recursion, Reduction, and Iteration

11.6.1 Recursion

It is often very useful to be able to refer to a sequence currently under construction, for example to define the sequence recursively. For this purpose the `Self` operator is available.

`Self(n): RngIntElt -> Elt`

`Self() -> SeqEnum`

This operator enables the user to refer to an already defined previous entry $s[n]$ of the enumerated sequence s inside the sequence constructor, or the sequence s itself.

Example: Recursive Fibonacci Sequence `(ex-5ee4f8)`

The example below shows how the sequence of the first 100 Fibonacci numbers can be created recursively, using `Self`. Next it is shown how to use reduction on these 100 integers.

```
> s := [ i gt 2 select Self(i-2)+Self(i-1) else 1 : i in [1..100] ];
> &+s;
927372692193078999175
```

11.6.2 Reduction

Instead of using a loop to apply the same binary associative operator to all elements of a complete enumerated sequence, it is possible to use the *reduction operator* `&`.

& ∘ S: Op, SeqEnum -> Elt

Given a complete enumerated sequence $S = [a_1, a_2, \ldots, a_n]$ of elements belonging to an algebraic structure U, and an (associative) operator $\circ : U \times U \to U$, form the element $a_1 \circ a_2 \circ a_3 \circ \ldots \circ a_n$. Currently, the following operators may be used to reduce sequences: **+, *, and, or, join, meet, cat**. An error will occur if the operator is not defined on U. If S contains a single element a, then the value returned is a. If S is the null sequence (empty and no universe specified), then reduction over S leads to an error; if S is empty with universe U in which the operation is defined, then the result (or error) depends on the operation and upon U. The following table defines the return value:

```
             EMPTY     NULL
    &+        U!0      error
    &*        U!1      error
    &and     true      true
    &or      false     false
    &join    empty     null
    &meet    error     error
    &cat     empty     null
```

11.6.3 Iteration

The iteration operator **in** can be used in loops or in the set and sequence constructors to enumerate the defined elements of a sequence.

When multiple range sequences are used, it is important to know in which order the range are iterated over; the rule is that the repeated iteration takes place as nested loops where the first range forms the innermost loop, etc. See the examples below.

x in S

i -> x in S

Enumerate the defined elements of the sequence S. x takes on the successive defined values in the sequence; if the dual iteration form is used then i is the corresponding numeric index of x.

Example: Nested Iteration (ex-c6e3f6)

The first example shows how repeated iteration inside a sequence constructor corresponds to nesting of loops.

```
> [ <number, letter> : number in [1..5], letter in ["a", "b", "c"]];
[ <1, a>, <2, a>, <3, a>, <4, a>, <5, a>, <1, b>, <2, b>, <3, b>, <4, b>, <5,
b>, <1, c>, <2, c>, <3, c>, <4, c>, <5, c> ]
> r := [];
> for letter in ["a", "b", "c"] do
>     for number in [1..5] do
>         Append(~r, <number, letter>);
>     end for;
> end for;
> r;
[ <1, a>, <2, a>, <3, a>, <4, a>, <5, a>, <1, b>, <2, b>, <3, b>, <4, b>, <5,
b>, <1, c>, <2, c>, <3, c>, <4, c>, <5, c> ]
```

This explains why the first construction below leads to an error, whereas the second leads to the desired sequence.

```
> // The following produces an error:
> [ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 16 ];
                                            ^
User error: Identifier 'x' has not been declared
```

(continues on next page)

(continued from previous page)

```
> [ <x, y> : x in [0..y], y in [0..5] | x^2+y^2 lt 16 ];
[ <0, 0>, <0, 1>, <1, 1>, <0, 2>, <1, 2>, <2, 2>, <0, 3>, <1, 3>, <2, 3> ]
```

Note the following! In the last line below there are two different things with the name x. One is the (inner) loop variable, the other just an identifier with value 1000 that is used in the bound for the other (outer) loop variable y: the limited scope of the inner loop variable x makes it invisible to y, whence the error in the first case.

```
> // The following produces an error:
> #[ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 100 ];
                                          ^
User error: Identifier 'x' has not been declared

> x := 1000;
> #[ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 100 ];
59
```

Example: Dual Iteration over Sequences (ex-13375d)

This example shows how dual iteration allows us to get both the index and the value simultaneously.

```
> letters := Eltseq("abc");
> [ <i, x> : i -> x in letters ];
[ <1, "a">, <2, "b">, <3, "c"> ]
> for i -> x in letters do
>     printf "letters[%o] is %o\n", i, x;
> end for;
letters[1] is a
letters[2] is b
letters[3] is c
```

Dual iteration allows us to easily get the indices of defined elements; note the use of an underscore in the dual iteration below to ignore the values, since we do not use them.

```
> S := [ 7, 3 ];
> S[5] := 2;
> S;
[ 7, 3, undef, undef, 2 ]
> [ i : i -> _ in S ];
[ 1, 2, 5 ]
```

11.7 Bibliography

- [Bee93] L. Beeckmans. The splitting algorithm for Egyptian fractions. *J. Number Th.*, 43:173–185, 1993.

Chapter 12

Tuples and Cartesian Products

12.1 Introduction

A cartesian product may be constructed from a finite number of factors, each of which may be a set or algebraic structure. The term *tuple* will refer to an element of a cartesian product. Note that the rules for tuples are quite different to those for sequences. Sequences are elements of a cartesian product of n copies of a fixed set (or algebraic structure) while tuples are elements of cartesian products where the factors may be different sets (structures). The semantics for tuples are quite different to those for sequences. In particular, the parent cartesian product of a tuple is fixed once and for all. This is in contrast to a sequence, which may grow and shrink during its life (thus implying a varying parent cartesian product).

12.2 Cartesian Product Constructor and Functions

The special constructor `car< ... >` is used for the creation of cartesian products of structures.

`car< R1, ..., Rk >: Str, ..., Str -> SetCart`

Given a list of sets or algebraic structures $R_1, \ldots, R_k$, construct the cartesian product set $R_1 \times \cdots \times R_k$.

`CartesianProduct(R, S): Str, ..., Str -> SetCart`

Given structures R and S, construct the cartesian product set $R \times S$. This is the same as calling the `car` constructor with the two arguments R and S.

`CartesianProduct(L): [Str] -> SetCart`

`CartesianProduct(L): <Str> -> SetCart`

Given a sequence or tuple L of structures, construct the cartesian product of the elements of L.

`CartesianPower(R, k): Str, RngIntElt -> SetCart`

Given a structure R and an integer k, construct the cartesian power set R^k.

`Flat(C): SetCart -> SetCart`

Given a cartesian product C of structures which may themselves be cartesian products, return the cartesian product of the base structures, considered in depth-first order (see *Flat* for the element version).

`NumberOfComponents(C): SetCart -> RngIntElt`

Given a cartesian product C, return the number of components of C.

`Component(C, i): SetCart, RngIntElt -> Str`

`C[i]: SetCart, RngIntElt -> Str`

The i-th component of C.

`Components(C): SetCart -> List`

The list of components of a cartesian product.

`# C: SetCart -> RngIntElt`

Given a cartesian product C, return the cardinality of C.

`Rep(C): SetCart -> Elt`

Given a cartesian product C, return a representative of C.

`Random(C): SetCart -> Elt`

Given a cartesian product C, return a random element of C.

Example: Cartesian Product of Q and Z (ex-c2f16a)

We create the product of $\mathbb{Q}$ and $\mathbb{Z}$.

```
> C := car< RationalField(), Integers() >;
> C;
Cartesian Product<Rational Field, Ring of Integers>
```

12.3 Creating and Modifying Tuples

`elt< C | a1, a2, ..., ak >: SetCart, Elt, ..., Elt -> Tup`

`C ! < a1, a2, ..., ak >: SetCart, Elt, ..., Elt -> Tup`

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a sequence of elements $a_1, a_2, \ldots, a_k$, such that a_i belongs to the set R_i $(i = 1, \ldots, k)$, create the tuple $T =< a_1, a_2, ..., a_k >$ of C.

`< a1, a2, ..., ak >: Elt, ..., Elt -> Tup`

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a list of elements $a_1, a_2, \ldots, a_k$, such that a_i belongs to the set R_i, $(i = 1, \ldots, k)$, create the tuple $T =< a_1, a_2, ..., a_k >$ of C. Note that if C does not already exist, it will be created at the time this expression is evaluated.

`Append(T, x): Tup, Elt -> Tup`

Return the tuple formed by adding the object x to the end of the tuple T. Note that the result lies in a new cartesian product of course.

`Append(~T, x): Tup, Elt`

(Procedure.) Destructively add the object x to the end of the tuple T. Note that the new T lies in a new cartesian product of course.

`Prune(T): Tup -> Tup`

Return the tuple formed by removing the last term of the tuple T. The length of T must be greater than 1. Note that the result lies in a new cartesian product of course.

`Prune(~T): Tup`

(Procedure.) Destructively remove the last term of the tuple T. The length of T must be greater than 1. Note that the new T lies in a new cartesian product of course.

`Flat(T): Tup -> Tup`

Construct the flattened version of the tuple T. The flattening is done in the same way as *Flat*, namely depth-first.

Example: Pairs of Primes and Reciprocals (ex-cc3d86)

We build a set of pairs consisting of primes and their reciprocals.

```
> C := car< Integers(), RationalField() >;
> C ! < 26/13, 13/26 >;
<2, 1/2>
> S := { C | <p, 1/p> : p in [1..25] | IsPrime(p) };
> S;
```

(continues on next page)

(continued from previous page)

```
{ <5, 1/5>, <7, 1/7>, <2, 1/2>, <19, 1/19>, <17, 1/17>, <23, 1/23>, <11, 1/11>,
<13, 1/13>, <3, 1/3> }
```

12.4 Tuple Access Functions

Parent(T): Tup -> SetCart

The cartesian product to which the tuple T belongs.

 # T: Tup -> RngIntElt

Number of components of the tuple T.

T[i]: Tup -> Elt

Return the i-th component of tuple T. Note that this indexing can also be used on the left hand side for modification of T.

Explode(T): Tup -> t1, t2, ...

Given a tuple T of length n, this function returns the n entries of T (in order).

TupleToList(T): Tup -> List

Tuplist(T): Tup -> List

Given a tuple T return a list containing the entries of T.

Example: Accessing Tuple Entries (ex-58b218)

```
> f := < 11/2, 13/3, RootOfUnity(3, CyclotomicField(3)) >;
> f;
<11/2, 13/3, (zeta_3)>
> #f;
3
> Parent(f);
Cartesian Product<Rational Field, Rational Field, Cyclotomic field Q(zeta_3)>
> f[1]+f[2]+f[3];
(1/6) * (59 + 6*zeta_3)
> f[3] := 7;
> f;
<11/2, 13/3, 7>
```

12.5 Equality

T eq U: Tup, Tup -> BoolElt

Return **true** if and only if the tuples T and U are equal.

T ne U: Tup, Tup -> BoolElt

Return **true** if and only if the tuples T and U are distinct.

12.6 Iteration

Iteration over the components of a tuple is allowed. Note that, unlike most other iterators, the values returned by iteration over a tuple may be of differing types.

x in T

i -> x in T

The identifier x will take on the value of successive components of the tuple T; if the dual iteration form is used then i will be the corresponding index.

12.7 Other Operations

IsEmpty(T): Tup -> BoolElt

Returns whether the tuple has no components.

&* T: Tup -> Any

For a tuple T where each component lies in a structure that supports multiplication and such that there exists a common overstructure, return the product of the entries.

Chapter 13

Lists

13.1 Introduction

A *list* in Magma is an ordered finite collection of objects. Unlike sequences, lists are not required to consist of objects that have some common parent. Lists are not stored compactly and the operations provided for them are not extensive. They are mainly provided to enable the user to gather assorted objects temporarily together.

13.2 Construction of Lists

Lists can be constructed by expressions enclosed in special brackets `[*` and `*]`.

`[* *] -> List`

The empty list.

`[* e1, e2, ..., en *]: Elt, .., Elt -> List`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$, create the list containing $a_1, a_2, \ldots, a_n$.

13.3 Creation of New Lists

Here, S denotes the list $[* s_1, \ldots, s_n *]$, while T denotes the list $[* t_1, \ldots, t_m *]$.

`S cat T: List, List -> List`

The list formed by concatenating the terms of the list S with the terms of the list T, i.e. the list $[* s_1, \ldots, s_n, t_1, \ldots, t_m *]$.

`S cat:= T: List, List`

(Procedure.) Destructively concatenate the terms of the list T to S; i.e. so S becomes the list $[* s_1, \ldots, s_n, t_1, \ldots, t_m *]$.

`Append(S, x): List, Elt -> List`

The list formed by adding the object x to the end of the list S, i.e. the list $[* s_1, \ldots s_n, x *]$.

`Append(~S, x): List, Elt`

(Procedure.) Destructively add the object x to the end of the list S; i.e. so S becomes the list $[* s_1, \ldots s_n, x *]$.

`Insert(~S, i, x): List, RngIntElt, Any`

`Insert(S, i, x): List, RngIntElt, Any -> List`

Create the list formed by inserting the object x at position i in S and moving the terms $S[i], \ldots, S[n]$ down one place, i.e., the list $[* s_1, \ldots, s_{i-1}, x, s_i, \ldots, s_n *]$. Note that i must not be bigger than $n+1$ where n is the length of S. There are two versions of this: a procedure, where S is replaced by

the new list, and a function, which returns the new list. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the list S will not be copied.

`Prune(S): List -> List`

The list formed by removing the last term of the list S, i.e. the list $[* s_1, \ldots, s_{n-1} *]$.

`Prune(~S): List`

(Procedure.) Destructively remove the last term of the list S; i.e. so S becomes the list $[* s_1, \ldots, s_{n-1} *]$.

`SequenceToList(Q): SeqEnum -> List`

`Seqlist(Q): SeqEnum -> List`

Given a sequence Q, construct a list whose terms are the elements of Q taken in the same order.

`TupleToList(T): Tup -> List`

`Tuplist(T): Tup -> List`

Given a tuple T, construct a list whose terms are the elements of T taken in the same order.

`Reverse(L): List -> List`

Given a list L return the same list, but in reverse order.

13.4 Access Functions

`# S: List -> RngIntElt`

The length of the list S.

`IsEmpty(S): List -> BoolElt`

Return whether S is empty (has zero length).

`S[i]: List, RngIntElt -> Elt`

Return the i-th term of the list S. If either $i \leq 0$ or $i > \#S + 1$, then an error results. Here i is allowed to be a multi-index (see Section *Multi-indexing* for the interpretation).

`S[I]: List, [RngIntElt] -> List`

Return the sublist of S given by the indices in the sequence I. Each index in I must be in the range $[1..l]$, where l is the length of S.

`IsDefined(L, i): List, RngIntElt -> Elt`

Checks whether the ith item in L is defined or not, that is it returns `true` if i is at most the length of L and `false` otherwise.

13.5 Assignment Operator

`S[i] := x: List, RngIntElt, Elt`

Redefine the i-th term of the list S to be x. If $i \leq 0$, then an error results. If $i = \#S + 1$, then x is appended to S. Otherwise, if $i > \#S + 1$, an error results. Here i is allowed to be a multi-index.

13.6 Iteration

Iteration over lists is allowed. Note that, unlike most other iterators, the values returned by iteration over a list may be of differing types.

`x in L`

`i -> x in L`

The identifier x will take on the value of successive elements of the list L; if the dual iteration form is used then i will be the corresponding index.

Chapter 14

Associative Arrays

14.1 Introduction

An *associative array* in Magma is an array which may be indexed by arbitrary elements of an index structure I. The indexing may thus be by objects which are not integers. These objects are known as the *keys*. For each current key there is an associated value. The *values* associated with the keys need not lie in a fixed universe but may be of any type.

14.2 Operations

AssociativeArray() -> Assoc

```
Default: Any                    Default:
```

Create the null associative array A with no index universe. The first assignment to A will determine its index universe. The optional parameter `Default := D` allows one to associate the default value D to A (so $A[x]$ will return D if x is not in the keys of A).

AssociativeArray(I): Str -> Assoc

```
Default: Any                    Default:
```

Create the empty associative array A with index universe I. The optional parameter `Default := D` allows one to associate the default value D to A (so $A[x]$ will return D if x is not in the keys of A).

A[x] := y: Assoc, Elt, Elt

Set the value in A associated with index x to be y. If x is not coercible into the current index universe I of A, then an attempt is first made to lift the index universe of A to contain both I and x.

In a nested assignment such as `A[x][z] := y`, if x is not in the keys of A but a default value D was specified when A was created, then $A[x]$ is first set to D and the assignment then proceeds on that value. Thus arrays of arrays may be built up without initializing each level explicitly (see the example below).

A[x]: Assoc, Elt -> Elt

Given an index x coercible into the index universe I of A, return the value associated with x. If x is not in the keys of A, then: (1) if a default value D was specified when A was created, then D is returned; (2) otherwise, an error is raised.

`IsDefined(A, x): Assoc, Elt -> Bool, Elt`

Given an index x coercible into the index universe I of A, return whether $A[x]$ is currently defined, and if so, return also the value $A[x]$. This is the case if x is in the keys of A or A has a default value.

`IsInKeys(A, x): Assoc, Elt -> Bool, Elt`

Given an index x coercible into the index universe I of A, return whether x is explicitly in the keys of A and if so, return also the value $A[x]$.

`Remove(~A, x): Assoc, Elt`

(Procedure.) Destructively remove the value indexed by x from the array A. If x is not present as an index, then nothing happens (i.e., an error is not raised).

`Universe(A): Assoc -> Str`

Given an associative array A, return the index universe I of A, in which the keys of A currently lie.

`# A: Assoc -> RngIntElt`

Given an associative array A, return the number of items stored in A.

`Keys(A): Assoc -> SetEnum`

Given an associative array A, return the current keys of A as a set. Warning: this constructs a new copy of the set of keys, so should only be called when that is needed; it is not meant to be used as a quick access function.

`Values(A): Assoc -> List`

Given an associative array A, return the current values of A as an unsorted list (a list is returned since the values need not lie in a fixed universe but may be of any type). Warning: this constructs a new copy of the list of values, so should only be called when that is needed; it is not meant to be used as a quick access function.

Example: Creating an Associative Array (ex-c3ad44)

This example shows simple use of associative arrays. First we create an array indexed by rationals.

```
> A := AssociativeArray();
> A[1/2] := 7;
> A[3/8] := "abc";
> A[3] := 3/8;
> A[1/2];
7
> IsDefined(A, 3);
true 3/8
> IsDefined(A, 4);
false
> IsDefined(A, 3/8);
true abc
> Keys(A);
{ 3/8, 1/2, 3 }
> Values(A);
[* 7, abc, 3/8 *]
> for x in Keys(A) do x, A[x]; end for;
1/2 7
3/8 abc
3 3/8
> Remove(~A, 3/8);
> IsDefined(A, 3/8);
false
> Keys(A);
{ 1/2, 3 }
```

(continues on next page)

(continued from previous page)

```
> Values(A);
[* 7, 3/8 *]
> Universe(A);
Rational Field
```

We repeat that an associative array can be indexed by elements of any structure. We now index an array by elements of the symmetric group S_3.

```
> G := Sym(3);
> A := AssociativeArray(G);
> v := 1; for x in G do A[x] := v; v +:= 1; end for;
> A;
Associative Array with index universe GrpPerm: G, Degree 3, Order 2 * 3
> Keys(A);
{
    (1, 3, 2),
    (2, 3),
    (1, 3),
    (1, 2, 3),
    (1, 2),
    Id(G)
}
> A[G!(1,3,2)];
3
```

The following shows how the parameter `Default` can be used when an associative array A is created.

```
> A := AssociativeArray(: Default := []);
> x := 3; y := 5;
> Append(~A[x], y);
> assert A[x] eq [y];
> assert Keys(A) eq {x};
> assert Values(A) eq [* [ y ] *];
> IsDefined(A, x);
true [ 5 ]
> IsInKeys(A, x);
true [ 5 ]
> IsDefined(A, 4);
true []
> IsInKeys(A, 4);
false
```

Here we can append an element to $A[x]$ even when x is not yet in the keys of A; in such a case, $A[x]$ is initially taken to be [] and so can be appended to without error.

Since the default may itself be an associative array with a default, nested arrays may be built up in the same way, without initializing any of the intermediate levels.

```
> A := AssociativeArray(: Default := AssociativeArray(: Default := []));
> Append(~A[1][2], 3);
> Append(~A[1][2], 4);
> Append(~A[5][6], 7);
> A[1][2];
[ 3, 4 ]
> Keys(A);
{ 1, 5 }
> Keys(A[5]);
{ 6 }
```

An index is only added to the keys of A when something is actually assigned to it, so simply reading a value leaves A unchanged.

```
> A[9][9];
[]
> Keys(A);
{ 1, 5 }
```

14.3 Iteration

v in A

k -> v in A

Iteration over the values in an associative array is supported as usual; in the forms above, v will take the value of successive elements in the associative array. If the dual iteration form is used then additionally k will take on the value of the associated key.

Example: Iterating over an Associative Array (ex-adf920)

We revisit the first part of the above example and print the elements out using the dual iteration.

```
> A := AssociativeArray();
> A[1/2] := 7;
> A[3/8] := "abc";
> A[3] := 3/8;
> for k -> v in A do k, v; end for;
1/2 7
3/8 abc
3 3/8
```

Chapter 15

Coproducts

15.1 Introduction

Coproducts can be useful in various situations, as they may contain objects of entirely different types. Although the coproduct structure will serve as a single parent for such diverse objects, the proper parents of the elements are recorded internally and restored whenever the element is retrieved from the coproduct.

15.2 Creation Functions

There are two versions of the coproduct constructor. Ordinarily, coproducts will be constructed from a list of structures. These structures are called the *constituents* of the coproduct. A single sequence argument is allowed as well to be able to create coproducts of parameterized families of structures conveniently.

15.2.1 Creation of Coproducts

`cop< S1, S2, ..., Sk >: Str, Str, ... -> Cop, [ Map ]`

`cop< [ S1, S2, ..., Sk ] >: [ Str, Str, ... ] -> Cop, [ Map ]`

Given a list or a sequence of two or more structures S_1, S_2, ..., S_k, this function creates and returns their coproduct C as well as a sequence of maps $[m_1, m_2, \ldots, m_k]$ that provide the injections $m_i \colon S_i \to C$.

15.2.2 Creation of Coproduct Elements

Coproduct elements are usually created by the injections returned as the second return value from the `cop<>` constructor. The bang (!) operator may also be used but only if the type of the relevant constituent is unique for the particular coproduct.

`m(e): Map, Elt -> CopElt`

Given a coproduct injection map m and an element of one of the constituents of the coproduct C, create the coproduct element version of e.

`C ! e: Cop, Elt -> CopElt`

Given a coproduct C and an element e of one of the constituents of C such that the type of that constituent is unique within that coproduct, create the coproduct element version of e.

15.3 Accessing Functions

`Injections(C): Cop -> [ Map ]`

Given a coproduct C, return the sequence of injection maps returned as the second argument from the `cop<>` constructor.

`# C: Cop -> RngIntElt`

Given a coproduct C, return the length (number of constituents) of C.

`Constituent(C, i): Cop, RngIntElt -> Str`

Given a coproduct C and an integer i between 1 and the length of C, return the i-th constituent of C.

`Index(x): CopElt -> RngIntElt`

Given an element x from a coproduct C, return the constituent number of C to which x belongs.

15.4 Retrieve

The function described here restores an element of a coproduct to its original state.

`Retrieve(x): CopElt -> Elt`

Given an element x of some coproduct C, return the element as an element of the structure that formed its parent before it was mapped into C.

Example: Basic Use of Coproducts (ex-13536c)

We illustrate basic uses of the coproduct constructors and functions.

```
> C := cop<IntegerRing(), Strings()>;
> x := C ! 5;
> y := C ! "abc";
> x;
5
> y;
abc
> Parent(x);
Coproduct<Integer Ring, String structure>
> x eq 5;
true
> x eq y;
false
> Retrieve(x);
5
> Parent(Retrieve(x));
Integer Ring
```

15.5 Flattening

The function described here enables the 'concatenation' of coproducts into a single one.

`Flat(C): Cop -> Cop`

Given a coproduct C of structures which may themselves be coproducts, return the coproduct of the base structures, considered in depth-first order.

15.6 Universal Map

`UniversalMap(C, S, [ n`$_1$`, ..., n`$_m$` ]): Cop, Str, [ Map ] -> Map`

Given maps n_1, ..., n_m from structures S_1, ..., S_m that compose the coproduct C, to some structure S, this function returns the universal map $C \to S$.

Chapter 16

Records

16.1 Introduction

In a *record* several objects can be collected. The objects in a record are stored in *record fields*, and are accessed by using *fieldnames*. Records are like tuples (and unlike sets or sequences) in that the objects need not all be of the same kind. Though records and tuples are somewhat similar, there are several differences too. The components of tuples are indexed by integers, and every component must be defined. The fields of records are indexed by fieldnames, and it is possible for some (or all) of the fields of a record not to be assigned; in fact, a field of a record may be assigned or deleted at any time. A record must be constructed according to a pre-defined *record format*, whereas a tuple may be constructed without first giving the Cartesian product that is its parent, since Magma can deduce the parent from the tuple. In the definition of a record format, each field is given a fieldname. If the field is also given a parent magma or a category, then in any record created according to this format, that field must conform to this requirement. However, if the field is not given a parent magma or category, there is no restriction on the kinds of values stored in that field; different records in the format may contain disparate values in that field. By contrast, every component of a Cartesian product is a magma, and the components of all tuples in this product must be elements of the corresponding magma. Because of the flexibility of records, with respect to whether a field is assigned and what kind of value is stored in it, Boolean operators are not available for comparing records.

16.2 The Record Format Constructor

The special constructor `recformat< ... >` is used for the creation of record formats. A record format must be created before records in that format are created.

`recformat< L >: FieldnameList -> RecFormat`

Construct the record format corresponding to the non-empty fieldname list L. Each term of L must be one of the following:

(a) *fieldname* in which case there is no restriction on values that may be stored in this field of records having this format;

(b) *fieldname*:*expression* where the expression evaluates to a magma which will be the parent of values stored in this field of records having this format; or

(c) *fieldname*:*expression* where the expression evaluates to a category which will be the category of values stored in this field of records having this format;

where *fieldname* consists of characters that would form a valid identifier name. Note that it is not a string.

Example: Creating a Record Format (ex-d1148b)

We create a record format with these fields: `n`, an integer; `misc`, which has no restrictions; and `seq`, a sequence (with any universe possible).

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> RF;
recformat<n: IntegerRing(), misc, seq: SeqEnum>
> Names(RF);
[ n, misc, seq ]
```

16.3 Creating a Record

Before a record is created, its record format must be defined. A record may be created by assigning as few or as many of the record fields as desired.

`rec< F | L >: RecFormat, FieldAssignmentList -> Rec`

Given a record format F, construct the record format corresponding to the field assignment list L. Each term of L must be of the form $fieldname := expression$ where $fieldname$ is in F and the value of the expression conforms (directly or by coercion) to any restriction on it. The list L may be empty, and there is no fixed order for the fieldnames.

Example: Building Records (ex-648bdd)

We build some records having the record format `RF`.

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> r := rec< RF | >;
> r;
rec<RF | >
> s := rec< RF | misc := "adsifaj", n := 42, seq := [ GF(13) | 4, 8, 1 ]>;
> s;
rec<RF | n := 42, misc := adsifaj, seq := [ 4, 8, 1 ]>
> t := rec< RF | seq := [ 4.7, 1.9 ], n := 51/3 >;
> t;
rec<RF | n := 17, seq := [ 4.7, 1.9 ]>
> u := rec< RF | misc := RModule(PolynomialRing(Integers(7)), 4) >;
> u;
rec<RF | misc := RModule of dimension 4 with base ring Univariate Polynomial
Algebra over Integers(7)>
```

16.4 Access and Modification Functions

Fields of records may be inspected, assigned, and deleted at any time.

`Format(r): Rec -> RecFormat`

The format of record r.

`Names(F): RecFrmt -> [ MonStgElt ]`

The fieldnames of the record format F returned as a sequence of strings.

`Names(r): Rec -> [ MonStgElt ]`

The fieldnames of record r returned as a sequence of strings.

`r`fieldname: Rec, Fieldname -> Elt`

Return the field of record r with this fieldname. The format of r must include this fieldname, and the field must be assigned in r.

r`fieldname := expression;

Reassign the given field of r to be the value of the expression. The format of r must include this fieldname, and the expression's value must satisfy (directly or by coercion) any restriction on the field.

delete r`fieldname: Rec, Fieldname -> Nil

(Statement.) Delete the current value of the given field of record r.

assigned r`fieldname: Rec, Fieldname -> BoolElt

Returns true if and only if the given field of record r currently contains a value.

r`` s: Rec, String -> Elt

Given an expression s that evaluates to a string, return the field of record r with the fieldname corresponding to this string. The format of r must include this fieldname, and the field must be assigned in r. This syntax may be used anywhere that **r`fieldname** may be used, including in left hand side assignment, **assigned** and **delete**.

Example: Accessing Record Fields (ex-81cb41)

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> r := rec< RF | >;
> s := rec< RF | misc := "adsifaj", n := 42, seq := [ GF(13) | 4, 8, 1 ]>;
> t := rec< RF | seq := [ 4.7, 1.9 ], n := 51/3 >;
> u := rec< RF | misc := RModule(PolynomialRing(Integers(7)), 4) >;
> V4 := u`misc;
> assigned r`seq;
false
> r`seq := Append(t`seq, t`n); assigned r`seq;
true
> r;
rec<RF | seq := [ 4.7, 1.9, 17 ]>
> // The following produces an error:
> t``(s`misc);

>> t``(s`misc);
          ^
Runtime error in `: Field 'adsifaj' does not exist in this record
> delete u``("m" cat "isc"); u;
rec<RF | >
```

Chapter 17

Mappings

17.1 Introduction

Mappings play a fundamental role in algebra and, indeed, throughout mathematics. Reflecting this importance, mappings are one of the fundamental datatypes in our language. The most general way to define a mapping $f : A \to B$ in a programming language is to write a *function* which, given any element of A, will return its image under f in B. While this approach to the definition of mappings is completely general, it is desirable to have mappings as an independent datatype. It is then possible to provide a very compact notation for specifying important classes of mappings such as homomorphisms. Further, a range of operations peculiar to the mapping type can be provided. Mappings are created either through use of *mapping constructors* as described in this Chapter, or through use of certain standard functions that return mappings as either primary or secondary values. All mappings are objects in the Magma category `Map`.

17.1.1 The Map Constructors

There are three main mapping constructors: the general map constructor `map< >`, the homomorphism constructor `hom< >`, and the partial map constructor `pmap< >`. The general form of all constructors is the same: inside the angle brackets there are two components separated by a pipe `|`. To the left the user specifies a *domain* A and a *codomain* B, separated by `->`; to the right of the pipe the user specifies how images are obtained for elements of the domain. The latter can be done in one of several ways: one specifies either the *graph* of the map, or a *rule* describing how images are to be formed, or for homomorphisms, one specifies generator images. We will describe each in the next subsections. The result is something like `map< A -> B | expr >`.

The domain and codomain of the map can be arbitrary magmas. When a full map (as opposed to a partial map) is constructed by use of a graph, the domain is necessarily finite.

The main difference between maps and partial maps is that a partial map need not be defined for every element of the domain. The main difference between these two types of map and homomorphisms is that the latter are supposed to provide *structure-preserving* maps between algebraic structures. On the one hand this makes it possible to allow the specification of images for homomorphisms in a different fashion: homomorphism can be given via *images* for *generators* of the domain. On the other hand homomorphisms are restricted to cases where domain and (image in the) codomain have a similar structure. The generator image form only makes sense for domains that are *finitely presented*. Homomorphisms are described in more detail below.

17.1.2 The Graph of a Map

Let A and B be structures. A *subgraph* of the cartesian product $C = A \times B$ is a subset G of C such that each element of A appears at most once among the first components of the pairs $< a, b >$ of G. A subgraph having the additional property that every element of A appears as the first component of some pair $< a, b >$ of G is called a *graph* of $A \times B$.

A mapping between A and B can be identified with a graph G of $A \times B$, a partial map can be identified with a subgraph. We now describe how a graph may be represented in the context of the map constructor. An element of the graph of $A \times B$ can be given either as a *tuple* `<a, b>`, or as an *arrow pair* `a -> b`. The specification of a (sub)graph in a map constructor should then consist of either a (comma separated) list, a sequence, or a set of such tuples or arrow pairs (a mixture is permitted).

17.1.3 Rules for Maps

The specification of a rule in the map constructor involves a free variable and an expression, usually involving the free variable, separated by `:->`, for example `x :-> 3*x - 1`. The scope of the free variable is restricted to the map constructor (so the use of x does not interfere with values of x outside the constructor). A general expression is allowed in the rule, which may involve intrinsic or user functions, and even in-line definitions of such functions.

17.1.4 Homomorphisms

Probably the most useful form of the map-constructor is the version for homomorphisms. Most interesting mappings in algebra are homomorphisms, and if an algebraic structure A belongs to a family of algebraic structures which form a variety we have the fundamental result that a homomorphism is uniquely determined by the images of any generating set. This provides us with a particularly compact way of defining and representing homomorphisms. While the syntax of the homomorphism constructor is similar to that of the general mapping constructor, the semantics are sometimes different. The kind of homomorphism built by the hom-constructor is determined entirely by the domain: thus, a *group* homomorphism results from applying `hom` to a domain A that is one of the types of group in Magma, a *ring* homomorphism results when A is a ring, etc. As a consequence, the requirements on the specification of homomorphisms are dependent on the category to which A belongs. Often, the codomain of a homomorphism is required to belong to the same variety. But even within a category the specification may depend on the type of structure; for details we refer the reader to the specific chapters. A homomorphism can be specified using either a rule map or by generator images. In the latter case the processor will seek to express an element as a word in the generators of A when asked to compute its image. Thus A needs to be finitely presented.

17.1.5 Checking of Maps

It should be pointed out that checking the 'correctness' of mappings can be done to a limited extent only. If the mapping is given by means of a graph, Magma will check that no multiple images are specified, and that an image is given for every element of the domain (unless a partial map is defined). If a rule is given, it cannot be checked that it is defined on all of the domain. Also, it is in general the responsibility of the user to ensure that the images provided for a `hom` constructor do indeed define a homomorphism.

17.2 Creation Functions

In this section we describe the creation of maps, partial maps, and homomorphisms via the various forms of the constructors, as well as maps that define coercions between algebraic structures.

17.2.1 Creation of Maps

Maps between structures A and B may be specified either by providing the full graph (as defined in the previous section) or by supplying an expression rule for finding images.

`map< A -> B | G >: Str, Str -> Map`

Given a finite structure A, a structure B, and a graph G of $A \times B$, construct the mapping $f : A \to B$, as defined by G. The graph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter. Note that G must be a full graph, i.e., every element of A must occur exactly once as a first component.

`map< A -> B | x :-> e(x) >: Str, Str -> Map`

Given a set or structure A, a set or structure B, a variable x, and an expression $e(x)$, usually involving x, construct the mapping $f : A \to B$, as defined by $e(x)$. It is the user's responsibility

to ensure that a value is defined for every $x \in A$. The scope of the variable x is restricted to the map-constructor.

`map< A -> B | x :-> e(x), y :-> i(y) >: Str, Str -> Map`

Given a set or structure A, a set or structure B, a variable x, an expression $e(x)$, usually involving x, a variable y, and an expression $i(y)$, usually involving y, construct the mapping $f : A \to B$, as defined by $x \mapsto e(x)$, with corresponding inverse $f^{-1} : B \to A$, as defined by $y \mapsto i(y)$. It is the user's responsibility to ensure that a value $e(x)$ is defined for every $x \in A$, a value $i(y)$ is defined for every $y \in B$, and that $i(y)$ is the true inverse of $e(x)$. The scope of the variables x and y is restricted to the map-constructor.

17.2.2 Creation of Partial Maps

Partial mappings are quite different to both general mappings and homomorphisms, in that images need not be defined for every element of the domain.

`pmap< A -> B | G >: Str, Str -> Map`

Given a finite structure A of cardinality n, a structure B, and a subgraph G of $A \times B$, construct the partial map $f : A \to B$, as defined by G. The subgraph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

`pmap< A -> B | x :-> e(x) >: Str, Str -> Map`

Given a set A, a set B, a variable x, and an expression $e(x)$, construct the partial map $f : A \to B$, as defined by $e(x)$. This form of the map constructor is a special case of the previous one whereby the image of x can be defined using a single expression. Again the scope of x is restricted to the map-constructor.

`pmap< A -> B | x :-> e(x), y :-> i(y) >: Str, Str -> Map`

This constructor is the same as the map constructor above which allows the inverse map $i(y)$ to be specified, except that the result is marked to be a partial map.

17.2.3 Creation of Homomorphisms

The principal construction for homomorphisms consists of the generator image form, where the images of the generators of the domain are listed. Note that the kind of homomorphism and the kind and number of generators for which images are expected, depend entirely on the type of the domain. Moreover, some features of the created homomorphism, e.g. whether checking of the homomorphism is done during creation or whether computing preimages is possible, depend on the types of the domain and the codomain. We refer to the appropriate handbook chapters for further information.

`hom< A -> B | G >: Str, Str -> Map`

Given a finitely generated algebraic structure A and a structure B, as well as a graph G of $A \times B$, construct the homomorphism $f : A \to B$ defined by extending the map of the generators of A to all of A. The graph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

The detailed requirements on the specification are module-dependent, and can be found in the chapter describing the domain A.

`hom< A -> B | y`$_1$`, ..., y`$_n$` >: Str, Str -> Map`

`hom< A -> B | x`$_1$` -> y`$_1$`, ..., x`$_n$` -> y`$_n$` >: Str, Str -> Map`

This is a module-dependent constructor for homomorphisms between structures A and B; see the chapter describing the functions for A. In general after the bar the images for all generators of the structure A must be specified.

`hom< A -> B | x :-> e(x) >: Str, Str -> Map`

Given a structure A, a structure B, a variable x, and an expression $e(x)$, construct the homomorphism $f : A \to B$, as defined by $e(x)$. This form of the map constructor is a special case of the previous one whereby the image of x can be defined using a single expression. Again the scope of x is restricted to the map-constructor.

`hom< A -> B | x :-> e(x), y :-> i(y) >: Str, Str -> Map`

This constructor is the same as the map constructor above which allows the inverse map $i(y)$ to be specified, except that the result is marked to be a homomorphism.

17.2.4 Coercion Maps

Magma has a sophisticated machinery for coercion of elements into structures other than the parent. Non-automatic coercion is usually performed via the `!` operator. To obtain the coercion map corresponding to `!` in a particular instance the `Coercion` function can be used.

`Coercion(D, C): Str, Str -> Map`

`Bang(D, C): Str, Str -> Map`

Given structures D and C such that elements from D can be coerced into C, return the map m that performs this coercion. Thus the domain of m will be D and the codomain will be C.

17.3 Operations on Mappings

17.3.1 Composition

Although compatible maps can be composed by repeated application, say $g(f(x))$, it is also possible to create a composite map.

`f * g: Map, Map -> Map`

Given a mapping $f : A \to B$, and a mapping $g : B \to C$, construct the composition h of the mappings f and g as the mapping $h = g \circ f : A \to C$.

`Components(f): Map -> [ Map ]`

Returns the maps which were composed to form f.

17.3.2 (Co)Domain and (Co)Kernel

The domain and codomain of any map can simply be accessed. Only for some intrinsic maps and for maps with certain domains and codomains, also the formation of image, kernel, and cokernel is available.

`Domain(f): Map -> Str`

The domain of the mapping f.

`Codomain(f): Map -> Str`

The codomain of the mapping f.

`Image(f): Map -> Elt`

Given a mapping f with domain A and codomain B, return the image of A in B as a substructure of B. This function is currently supported only for some intrinsic maps and for maps with certain domains and codomains.

`Kernel(f): Map -> Str`

Given the homomorphism f with domain A and codomain B, return the kernel of f as a substructure of A. This function is currently supported only for some intrinsic maps and for maps with certain domains and codomains.

17.3.3 Inverse

`Inverse(m): Map -> Map`

The inverse map of the map m.

17.3.4 Function

For a map given by a rule, it is possible to get access to the rule as a user defined function.

`Function(f): Map -> UserProgram`

The function underlying the mapping f. Only available if f has been defined by the user by means of a rule map (i.e., an expression for the image under f of an arbitrary element of the domain).

17.4 Images and Preimages

The standard mathematical notation is used to denote the calculation of a map image. Some mappings defined by certain system intrinsics and constructors permit the taking of preimages. However, preimages are not available for any mapping defined by means of the mapping constructor.

`a @ f: Elt, Map -> Elt`

`f(a): Map, Elt -> Elt`

Given a mapping f with domain A and codomain B, and an element a belonging to A, return the image of a under f as an element of B.

`S @ f: SetEnum, Map -> SetEnum`

`S @ f: SetIndx, Map -> SetIndx`

`S @ f: SeqEnum, Map -> Seq`

`f(S): SetEnum, Map -> SetEnum`

`f(S): SetIndx, Map -> SetIndx`

`f(S): SeqEnum, Map -> Seq`

Given a mapping f with domain A and codomain B, and a finite enumerated set, indexed set, or sequence S of elements belonging to A, return the image of S under f as an enumerated set, indexed set, or sequence of elements of B.

`C @ f: Str, Map -> Str`

`f(C): Map, Str -> Str`

Given a homomorphism f with domain A and codomain B, and a substructure C of A, return the image of C under f as a substructure of B.

`y @@ f: Elt, Map -> Elt`

Given a mapping f with domain A and codomain B, where f supports preimages, and an element y belonging to B, return the preimage of y under f as an element of A. If the mapping f is a homomorphism, then a single element is returned as the preimage of y. In order to obtain the full preimage of y, it is necessary to form the coset $K * y@@f$, where K is the kernel of f.

`R @@ f: SetEnum, Map -> SetEnum`

`R @@ f: SetIndx, Map -> SetIndx`

`R @@ f: SeqEnum, Map -> Seq`

Given a mapping f with domain A and codomain B, where f supports preimages, and a finite enumerated set, indexed set, or sequence of elements R belonging to B, return the preimage of R under f as an enumerated set, indexed set, or sequence of elements of A.

`D @@ f: Str, Map -> Str`

Given a mapping f with domain A and codomain B, where f supports preimages and the kernel of f is known or can be computed, and a substructure D of B, return the preimage of D under f as a substructure of A.

`HasPreimage(x, f): Any, Map -> BoolElt, Any`

Return whether the preimage of x under f can be taken and the preimage as a second argument if it can.

17.5 Parents of Maps

Parents of maps are structures knowing a domain and a codomain. They are often used in automorphism group calculations where a map is returned from an automorphism group into the set of all automorphisms of some structure. Parents of maps all inherit from the type `PowMap`. The type `PowMapAut` which inherits from `PowMap` is the type which the parents of automorphisms inherit from.

There is also a power structure of maps (of type `PowStr`, similar to that of other structures) which is used as a common overstructure of the different parents.

`Parent``(m): Map -> PowMap`

The parent of m.

`Domain``(P): PowMap -> Str`

`Codomain``(P): PowMap -> Str`

The domain and codomain of the maps for which P is the parent.

`Maps``(D, C): Str, Str -> PowMap`

`Iso``(D, C): Str, Str -> PowMap`

The parent of maps (or isomorphisms) from D to C. `Iso` will only return a different structure to `Maps` if it has been specifically implemented for such maps.

`Aut``(S): Str -> PowMapAut`

The parent of automorphisms of S.

References

[Bee93] L. Beeckmans. The splitting algorithm for Egyptian fractions. *J. Number Th.*, 43:173–185, 1993.

Index

Symbols

< a_1, a_2, ..., a_k > (*literal*), 41
A (*operation*), 47
C (*operation*), 41, 50
R (*operation*), 14
S (*operation*), 29, 45
T (*operation*), 42
& o S (*operation*), 23
& ○ S (*operation*), 37
&* T (*operation*), 43
{ U | } (*literal*), 8
{ U | e(x) : x in E | P(x) } (*literal*), 9
{ U | e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) } (*literal*), 9
{ U | e_1, e_2, ..., e_n } (*literal*), 8
{ U | i .. j by k } (*literal*), 12
{ U | i..j } (*literal*), 12
{ } (*literal*), 8
{ e(x) : x in E | P(x) } (*literal*), 9
{ e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) } (*literal*), 9
{ e_1, e_2, ..., e_n } (*literal*), 8
{ i .. j by k } (*literal*), 12
{ i..j } (*literal*), 12
{* *} (*literal*), 11
{* U | *} (*literal*), 11
{* U | e(x) : x in E | P(x) *} (*literal*), 11
{* U | e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) *} (*literal*), 11
{* U | e_1, e_2, ..., e_m *} (*literal*), 11
{* e(x) : x in E | P(x) *} (*literal*), 11
{* e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) *} (*literal*), 11
{* e_1, e_2, ..., e_n *} (*literal*), 11
{@ @} (*literal*), 10
{@ U | @} (*literal*), 10
{@ U | e(x) : x in E | P(x) @} (*literal*), 10
{@ U | e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)@} (*literal*), 10
{@ U | e_1, e_2, ..., e_m @} (*literal*), 10
{@ e(x) : x in E | P(x) @} (*literal*), 10
{@ e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) @} (*literal*), 10
{@ e_1, e_2, ..., e_n @} (*literal*), 10
\[m_1, ..., m_n] (*literal*), 28
[U | e(x) : x in E | P(x)] (*literal*), 27
[U | e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)] (*literal*), 28
[U | e_1, e_2, ..., e_m] (*literal*), 27
[U | i .. j by k] (*literal*), 28
[U | i..j] (*literal*), 28
[U |] (*literal*), 27
[e(x) : x in E | P(x)] (*literal*), 27
[e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)] (*literal*), 28
[e_1, e_2, ..., e_n] (*literal*), 27
[i .. j by k] (*literal*), 28
[i..j] (*literal*), 28
[ ] (*literal*), 27
[! x in F | P(x) !] (*literal*), 27
[* *] (*literal*), 44
[* e_1, e_2, ..., e_n *] (*literal*), 44

A

a @ f (*operation*), 59
Accessing Record Fields (*example*), 54
Accessing Tuple Entries (*example*), 42
Almost Fermat Cubes (*example*), 9
Almost Fermat Cubes with Indexed Sets (*example*), 10
And(~S, T) (*procedure*), 35
And(S, T) (*function*), 35
Append(~S, x) (*procedure*), 31, 44
Append(~T, x) (*procedure*), 41
Append(S, x) (*function*), 31, 44
Append(T, x) (*function*), 41
Arithmetic Progressions in Sequences (*example*), 28
Arithmetic Progressions in Sets (*example*), 12
assigned r`fieldname (*operation*), 54
AssociativeArray() (*function*), 46
AssociativeArray(I) (*function*), 46
Aut(S) (*function*), 60
A[x] (*literal*), 46
A[x] := y (*literal*), 46

B

Bang(D, C) (*function*), 58
Basic Use of Coproducts (*example*), 51
Binary Set Operators (*example*), 21
Building Records (*example*), 53

C

C ! < a_1, a_2, ..., a_k > (*operation*), 41
C ! e (*operation*), 50
C @ f (*operation*), 59
CanChangeUniverse(S, V) (*function*), 18, 33

car< R_1, ..., R_k > (*constructor*), 40
Cartesian Product of Q and Z (*example*), 41
CartesianPower(R, k) (*function*), 40
CartesianProduct(L) (*function*), 40
CartesianProduct(R, S) (*function*), 40
Category(S) (*function*), 14
ChangeUniverse(~S, V) (*procedure*), 17
ChangeUniverse(S, V) (*function*), 17, 33
ChangeUniverse(S, V) (*procedure*), 33
Codomain(f) (*function*), 58
Codomain(P) (*function*), 60
Coercion(D, C) (*function*), 58
Component(C, i) (*function*), 40
Components(C) (*function*), 40
Components(f) (*function*), 58
Constituent(C, i) (*function*), 51
cop< S_1, S_2, ..., S_k > (*constructor*), 50
cop< [S_1, S_2, ..., S_k] > (*constructor*), 50
Creating a Record Format (*example*), 52
Creating an Associative Array (*example*), 47
Cubes Found with ExtractRep (*example*), 16
C[i] (*literal*), 40

D

D @@ f (*operation*), 59
delete r`fieldname (*operation*), 54
Domain(f) (*function*), 58
Domain(P) (*function*), 60
Dual Iteration over Sequences (*example*), 39

E

Egyptian Fractions (*example*), 34
elt< C | a_1, a_2, ..., a_k > (*constructor*), 41
Eltseq(R) (*function*), 30
Exclude(~S, x) (*procedure*), 17, 31
Exclude(S, x) (*function*), 17, 31
exists(t){ e(x): x in E | P(x) } (*literal*), 22
exists(t){ e(x_1, ..., x_k): x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)} (*literal*), 22
exists(t_1, ..., t_r){ e(x) : x in E | P(x) } (*literal*), 22
exists(t_1, ..., t_r){ e(x_1, ..., x_k) : x_1 in E_1, ..., x_k in E_k | P } (*literal*), 22
Explode(R) (*function*), 30
Explode(T) (*function*), 42
ExtractRep(~R, ~r) (*procedure*), 16

F

f * g (*operation*), 58
f(a) (*literal*), 59
f(C) (*literal*), 59
f(S) (*literal*), 59
Farey Series (*example*), 33
Flat(C) (*function*), 40, 51
Flat(T) (*function*), 41
forall(t){ e(x) : x in E | P(x) } (*literal*), 22
forall(t){ e(x_1, ..., x_k): x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)} (*literal*), 22
forall(t_1, ..., t_r){ e(x) : x in E | P(x) } (*literal*), 22
forall(t_1, ..., t_r){ e(x_1, ..., x_k) : x_1 in E_1, ..., x_k in E_k | P } (*literal*), 22
Format(r) (*function*), 53
Function(f) (*function*), 59

H

Hash(x) (*function*), 17
HasPreimage(x, f) (*function*), 59
hom< A -> B | G > (*constructor*), 57
hom< A -> B | x :-> e(x) > (*constructor*), 57
hom< A -> B | x :-> e(x), y :-> i(y) > (*constructor*), 57
hom< A -> B | x_1 -> y_1, ..., x_n -> y_n > (*constructor*), 57
hom< A -> B | y_1, ..., y_n > (*constructor*), 57

I

i -> x in L (*literal*), 45
i -> x in S (*literal*), 24, 38
i -> x in T (*literal*), 42
Image(f) (*function*), 58
Include(~S, x) (*procedure*), 17, 31
Include(S, x) (*function*), 17, 31
Index(S, x) (*function*), 14, 30
Index(S, x, f) (*function*), 30
Index(x) (*function*), 51
Indexed Set of Sets (*example*), 14
IndexedSetToSequence(S) (*function*), 18
IndexedSetToSet(S) (*function*), 18
Injections(C) (*function*), 50
Insert(~S, i, x) (*procedure*), 31, 44
Insert(~S, k, m, T) (*procedure*), 31
Insert(S, i, x) (*function*), 31, 44
Insert(S, k, m, T) (*function*), 31
Inverse(m) (*function*), 58
IsComplete(S) (*function*), 36
IsDefined(A, x) (*function*), 46
IsDefined(L, i) (*function*), 45
IsDefined(S, i) (*function*), 36
IsDisjoint(R, S) (*function*), 20
IsEmpty(R) (*function*), 19
IsEmpty(S) (*function*), 36, 45
IsEmpty(T) (*function*), 43
Isetseq(S) (*function*), 18
Isetset(S) (*function*), 18
IsInKeys(A, x) (*function*), 47
IsNull(R) (*function*), 19
IsNull(S) (*function*), 36
Iso(D, C) (*function*), 60
IsSubsequence(S, T) (*function*), 36
IsSubsequence(S, T: Kind := option) (*function*), 36
Iterating over an Associative Array (*example*), 49

K

k -> v in A (*literal*), 49
Kernel(f) (*function*), 58
Keys(A) (*function*), 47

L

Letter Frequencies in a String (*example*), 24

M

m(e) (*literal*), 50
map< A -> B | G > (*constructor*), 56
map< A -> B | x :-> e(x) > (*constructor*), 56
map< A -> B | x :-> e(x), y :-> i(y) > (*constructor*), 57
Maps(D, C) (*function*), 60
Max(S) (*function*), 17, 30
Maximum(S) (*function*), 17, 30
Min(S) (*function*), 16, 30
Minimum(S) (*function*), 16, 30
Multiplicities(S) (*function*), 21
Multiplicity(S, x) (*function*), 21
Multiset Constructors (*example*), 11
Multisets(S, k) (*function*), 21
MultisetToSet(S) (*function*), 19

N

Names(F) (*function*), 53
Names(r) (*function*), 53
Nested Iteration (*example*), 38
Nested Quantifiers and Sums of Squares (*example*), 23
Not(~S) (*procedure*), 35
Not(S) (*function*), 35
NumberOfComponents(C) (*function*), 40

O

Or(~S, T) (*procedure*), 35
Or(S, T) (*function*), 35

P

P ! S (*operation*), 13, 29
Pairs of Primes and Reciprocals (*example*), 41
ParallelSort(~S, ~T) (*procedure*), 32
Parent(m) (*function*), 60
Parent(R) (*function*), 14
Parent(S) (*function*), 29
Parent(T) (*function*), 42
Partition(S, P) (*function*), 34
Partition(S, p) (*function*), 34
Permutations(S) (*function*), 21
Permutations(S, k) (*function*), 21
pmap< A -> B | G > (*constructor*), 57
pmap< A -> B | x :-> e(x) > (*constructor*), 57
pmap< A -> B | x :-> e(x), y :-> i(y) > (*constructor*), 57
Position(S, x) (*function*), 14, 30
Position(S, x, f) (*function*), 30
Power Sequences (*example*), 29
Power Sets (*example*), 13
PowerIndexedSet(R) (*function*), 13
PowerMultiset(R) (*function*), 13
PowerSequence(R) (*function*), 29
PowerSet(R) (*function*), 13
Prune(~S) (*procedure*), 31, 45
Prune(~T) (*procedure*), 41
Prune(S) (*function*), 31, 45
Prune(T) (*function*), 41

R

R @@ f (*operation*), 59
R diff S (*operation*), 20
R eq S (*operation*), 20
R join S (*operation*), 20
R meet S (*operation*), 20
R ne S (*operation*), 20
R notsubset S (*operation*), 20
R sdiff S (*operation*), 20
R subset S (*operation*), 20
r`` s (*literal*), 54
r`fieldname (*literal*), 53
r`fieldname := expression (*literal*), 53
Random Primitive Elements (*example*), 15
Random(C) (*function*), 41
Random(R) (*function*), 15, 30
random{ e(x) : x in E | P(x) } (*literal*), 15
random{ e(x_1, ..., x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) } (*literal*), 15
RandomSubset(S, k) (*function*), 21
rec< F | L > (*constructor*), 53
recformat< L > (*constructor*), 52
Recursive Fibonacci Sequence (*example*), 37
Remove(~A, x) (*procedure*), 47
Remove(~S, i) (*procedure*), 32
Remove(S, i) (*function*), 32
Rep(C) (*function*), 41
Rep(R) (*function*), 16, 30
rep{ e(x) : x in E | P(x) } (*literal*), 16
rep{ e(x_1, ..., x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) } (*literal*), 16
Representative(R) (*function*), 16, 30
Retrieve(x) (*function*), 51
Reverse(~S) (*procedure*), 32
Reverse(L) (*function*), 45
Reverse(S) (*function*), 32
Rotate(~S, p) (*procedure*), 32
Rotate(S, p) (*function*), 32

S

S @ f (*operation*), 59
S cat T (*operation*), 34, 44
S cat:= T (*operation*), 34, 44
S eq T (*operation*), 36
S ge T (*operation*), 37
S gt T (*operation*), 37

S in P (*operation*), 13, 29
S le T (*operation*), 37
S lt T (*operation*), 37
S ne T (*operation*), 36
Self() (*function*), 37
Self(n) (*function*), 37
Seqlist(Q) (*function*), 45
Seqset(S) (*function*), 34
SequenceToList(Q) (*function*), 45
SequenceToMultiset(Q) (*function*), 19
SequenceToSet(S) (*function*), 34
Set(M) (*function*), 14
Setseq(S) (*function*), 34
SetToIndexedSet(E) (*function*), 18
SetToMultiset(E) (*function*), 19
SetToSequence(S) (*function*), 34
Sort(~S) (*procedure*), 32
Sort(~S, C) (*procedure*), 32
Sort(~S, C, ~p) (*procedure*), 32
Sort(S) (*function*), 32
Sort(S, C) (*function*), 32
Subsequences(S, k) (*function*), 21
Subsets of Given Cardinality by Reduction (*example*), 23
Subsets(S) (*function*), 21
Subsets(S, k) (*function*), 21
Sums of Cubes with Exists (*example*), 22
Sums of Two Cubes with Include (*example*), 18
S[I] (*literal*), 14, 30, 45
S[i] (*literal*), 14, 30, 45
S[i] := x (*literal*), 45

T

T eq U (*operation*), 42
T ne U (*operation*), 42
TupleToList(T) (*function*), 42, 45
Tuplist(T) (*function*), 42, 45
Type(S) (*function*), 14
T[i] (*literal*), 42

U

Undefine(~S, i) (*procedure*), 33
Undefine(S, i) (*function*), 33
UniversalMap(C, S, [n_1, ..., n_m]) (*function*), 51
Universe of an Enumerated Set (*example*), 8
Universe(A) (*function*), 47
Universe(R) (*function*), 14
Universe(S) (*function*), 29

V

v in A (*literal*), 49
Values(A) (*function*), 47

X

x -> v in M (*literal*), 24
x eq y (*operation*), 19
x in L (*literal*), 45
x in R (*operation*), 19
x in S (*literal*), 24, 38
x in S (*operation*), 36
x in T (*literal*), 42
x ne y (*operation*), 19
x notin R (*operation*), 19
x notin S (*operation*), 36
Xor(~S, T) (*procedure*), 35
Xor(S, T) (*function*), 35

Y

y @@ f (*operation*), 59