`Algebra` User Guide and Reference Manual

Manuel Bronstein and Marc Moreno Maza

Version 1.0.1 – June 21, 2002

`Algebra` Contributors

`Algebra` is the result of extracting the central subset of the Σ^{it} library[1] and adding new data structures and polynomial types written by Marc Moreno Maza, in order to provide a general-purpose computer algebra library.
Besides the principal authors, the following Σ^{it} project members have contributed code that is now part of the `Algebra` library:

Laurent Bernardin
Marco Codutti
Niklaus Mannhart
Thom Mulders
Julien Ohler
Hélène Prieto

[1] Σ^{it}, which was until 1997 a project of Manuel Bronstein's computer algebra group at the ETH Zurich, and whose development continues within the CAFÉ project at INRIA Sophia–Antipolis, is a special-purpose computer algebra library, see `http://www-sop.inria.fr/cafe/Manuel.Bronstein/sumit`.

Contents

Commutative Algebra

Finite Fields

Linear Algebra

Univariate Polynomials and Series

Arithmetic

Input/Output

Data Structures

1 Introduction

What is Algebra?

Algebra is a general-purpose computer algebra library designed to provide reusable and efficient algorithms for manipulating the standard objects of algebra, namely polynomials, series and matrices. Built as an extension of the libaldor library, it provides ALDOR programmers with an extensible computer algebra layer with a rich data type hierarchy.

How do I get and install Algebra?

You can download Algebra by anonymous ftp from the CAFÉ server at ftp-sop.inria.fr in cafe/software/algebra, or from the URL:

http://www.inria.fr/cafe/Manuel.Bronstein/algebra/

After downloading the file algebra.tar.gz, issue "gzip -dc algebra.tar.gz | tar -xvf -" in order to unpack it. This will create the following directories:

- algebra/doc: this user guide,
- algebra/lib: the library,
- algebra/include: the required include files,
- algebra/test: some test files,
- algebra/samples: Algebra programming samples,

Once the above file is unpacked, do the following:

- add the option -csys=XXX to your ALDORARGS environment variable, where XXX depends on your hardware and operating system. Common values for XXX are axposf1v4 for OSF1 V4.0 on a DEC Alpha, linux-486 for linux on a 486 or above PC, and sun4os55g-v8 for SunOS 5.5 on a SPARC v8 machine. See the file $ALDORROOT/include/aldor.conf for other values;
- go to algebra/lib and execute source makealgebra-csh or source makealgebra-bash depending on your shell;
- if you want to build the GMP version of the library, execute source makealgebra.gmp-csh or source makealgebra.gmp-bash depending on your shell. See the subsection on using GMP for more information about using the GMP version of Algebra;
- if you want to build the debug version of the library, execute source makealgebra-csh or source makealgebra-bash depending on your shell; See the subsection on debugging for more information on using the debug library.

How do I use Algebra in my programs?

Once Algebra is properly built, you need to set the following environment variables before using it:

- the variables ALDORLIBROOT and ALGEBRAROOT should be set respectively to the main aldorlib and algebra directories;
- $ALDORLIBROOT/include and $ALGEBRAROOT/include should be appended to your INCPATH variable;
- $ALDORLIBROOT/lib and $ALGEBRAROOT/lib should be appended to your LIBPATH variable;

In your Aldor programs, use #include "algebra" instead of #include "aldor". When building your final executable, add the options

```
-lalgebra -laldor -y$ALGEBRAROOT/lib -y$ALDORLIBROOT/lib
```

to your compiler command line, or

```
-lalgebrad -laldord -dDEBUG -y$ALGEBRAROOT/lib -y$ALDORLIBROOT/lib
```

to link to the debug version of Algebra. Check the subsection on using GMP for the options required if you want to use the GMP library and the GMP version of Algebra.

If you are running Algebra inside the compiler interactive loop, then type the line

```
#include "aldorinterp"
```

immediately after #include "algebra", which will import various things for interactive use and make the interpreter loop print values automatically. As with any Aldor program, do not forget the -q option in order to optimize your programs, specially if performance is an issue.

Before using Algebra for the first time, please check your installation by running make in the algebra/test directory, followed by running testall.

Please report any installation problem or bugs you encounter to sumit@sophia.inria.fr.

2 User Guide

This guide introduces the common types and categories provided by Algebra, and presupposes some familiarity with programming in Aldor and libaldor. If you are unfamiliar with Aldor or libaldor, we suggest that you first go through the tutorial in aldorlib/tutorial/, which will familiarize you both with Aldor and libaldor.

Algebra provides over 180 categories and domains, and over 400 different exported functions, which can look daunting at first. Remember however that a large part of those are low–level functionalities that make it possible to programmers to write efficient applications. Most of the high-level functionalities are found in a small subset of types and those are the ones described in this short guide. As you become more familiar with Algebra, you will find the reference guide to be more useful when browsed on line.

As you learn programming with Algebra, make sure to check the stdmath/samples directory for various programming samples.

Basic algebraic categories

The basic algebraic categories provided by `Algebra` are shown in Figure 1. While it follows quite closely the usual algebraic structures, a few points needs to be mentioned.

- The multiplication $*$ is in general not assumed to be commutative, except in the subtree starting at `CommutativeRing`, which appears in blue in Figure 1. This means that code written for the other categories should be careful and not assume commutativity of $*$. Addition is on the other hand always assumed commutative.

- The `Algebra` category tree has a unique root `ExpressionType` to which almost all the types provided by `Algebra` belong. The category `ExpressionType` inherits `PrimitiveType` from `libaldor`, which means that types in `Algebra` must export an equality. That equality however does not need to be complete, as described in the = page of the `libaldor` reference manual. `ExpressionType` plays also an important role in input/output as explained below.

- The `libaldor` categories `AdditiveType`, `ArithmeticType` and `LinearCombinationType` already export the basic arithmetic operations provided respectively by abelian groups, rings and modules. In fact `AbelianGroup`, `Ring` and `Module` do inherit their exports from them. There is however a fundamental difference: while the `libaldor` categories export the usual operations $=$, $+$, $-$, $*$,..., they make no assumption about their algebraic properties whatsoever. On the other hand, the corresponding algebraic categories in `Algebra` do assume that $=$ is a complete equality test, and that the arithmetic operations satisfy the algebraic axioms of their mathematical structure. So for example, while `Integer` is extended by `Algebra` to be a `Ring` (among other things), `SingleFloat` remains an `ArithmeticType` but is not a `Ring`. There is a similar relationship between `LinearArithmeticType` and `Algebra`, both provided by `Algebra`. Mathematical types such as `DenseMatrix`(R) or `DenseUnivariatePolynomial`(R) allow their argument R to be an `ArithmeticType` rather than a `Ring`, but those among their functions that require a complete equality on R are not exported when R is not a `Ring`. Similarly, while `DenseUnivariatePolynomial`(R) is always an `ArithmeticType`, it is a `Ring` only when R itself is a `Ring`. For example,

  ```
  DenseMatrix DenseUnivariatePolynomial SingleFloat
  ```

  is a valid type in `Algebra`, but Gaussian elimination is not available for such matrices, and the Euclidean algorithm is not available for their polynomial entries. Basic arithmetic, as provided by `ArithmeticType` is however available.

Arithmetic

All of the arithmetic types provided by `libaldor` (machine and software integers, floats, as well as `Complex`) remain available in `Algebra`, and most of them are extended to various algebraic categories. As in `libaldor`, `Integer` is actually a macro that defaults to `AldorInteger`. If integer efficiency is important for your application, we strongly recommend that you link with the GMP version of `Algebra` instead, see the section on `using GMP` for more details. Regardless of the integer implementation that you choose, we also recommended that you use `MachineInteger` whenever

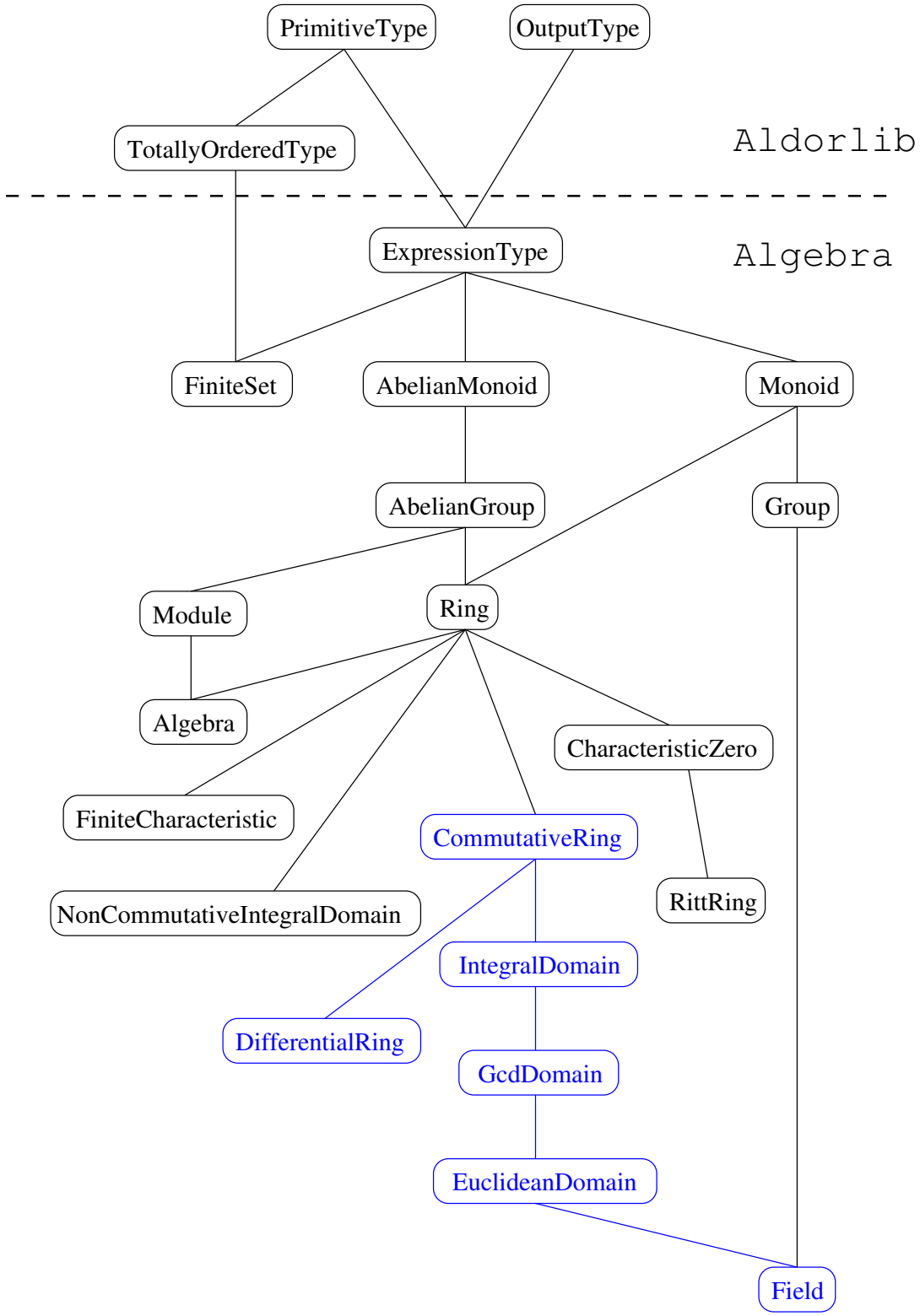

Figure 1: The `Algebra` basic algebraic category hierarchy

appropriate, in particular for loop or data structure indices. Conversions between `MachineInteger` and `Integer` are provided by `coerce` and `machine`.

In addition, `Algebra` provides two different implementations of the finite field $\mathbb{Z}/p\mathbb{Z}$ when p is a machine prime: `SmallPrimeField` provides a standard implementation, while `ZechPrimeField` provides a significantly faster implementation based on a logarithmic representation of its elements. However, `ZechPrimeField` precomputes a table of size $\mathcal{O}(p)$ so it should only be used for reasonably small values of p and when a significant amount of calculations in $\mathbb{Z}/p\mathbb{Z}$ are made following the creation of the type (we have found the precomputation time to be under one second for p around 10^6 on recent workstations).

The type `Fraction`(R) implements the fraction field of the `GcdDomain` R. Typical arguments are `Integer` (to get the rational numbers) or polynomial types. Note that fractions are automatically normalized after each arithmetic operation.

Data structures

All the data structure of `Algebra` are provided by `libaldor`. Some of them are extended by mathematical operations and take on a new name: `Sequence` corresponds to `Stream` from `libaldor` and `Vector` corresponds to `Array` from `libaldor`, in both cases with pointwise arithmetic operations added. Note however that `Array` is 0–indexed while `Vector` is 1–indexed.

Input/Output

`Algebra` inherits the stream I/O model provided by `libaldor`. In particular `ExpressionType` inherits `OutputType` from `libaldor`, which means that elements of a `ExpressionType` can be written in text format to any `TextWriter` (in particular I/O streams, strings or files). Because it is desirable to output mathematical objects such as matrices and polynomials in more than one format depending on the context, `Algebra` types do not in general define their own << function. Rather, they implement the `extree` function, that converts their elements into elements of `ExpressionTree`. Expression trees, wich are similar to Lisp's S–expressions, are the unique layer between all the types in `Algebra` and the outside world. When writing your own types, once you provide or inherit an implementation of `extree`, your elements can then be written to any `TextWriter` in any of the many formats that `ExpressionTree` understands, for example using `lisp` to produce Lisp output. In that case, the default behavior of the << is to convert your objects to expression trees and then to use `tex` to produce TEXoutput. You can override this default behaviour by providing your own implementation of << if you choose. As with `libaldor`, you can use `#include "aldorio"` *after* `#include "algebra"` in order to import the types commonly used for input and output.

Linear algebra

The basic types to use for doing linear algebra are `Vector` and `DenseMatrix`. The basic linear algebra computations are provided by the type `LinearAlgebra`. Although `Algebra` provides many specialized types for performing various sorts of triangulations and solving linear systems, `LinearAlgebra` knows about all of them and contains procedures that decide which algorithm is

best suited for your particular input. So do not call directly the more specialized packages unless you have a particular reason to use a specific algorithm rather than let `Algebra` decide for you.

The type of the entries of vectors and matrices does not need to be a `Ring`, it can be an `AdditiveType` or `ArithmeticType` respectively. This allows types that do not have a full equality such as `SingleFloat` or `DenseUnivariateTaylorSeries` to be entries of vectors and matrices. However, `LinearAlgebra` requires the entries to be from a `CommutativeRing` so its functionalities are not available for matrices of series or floating point numbers.

`Algebra` contains many fraction–free algorithms that allow you to perform some computations, such as matrix inverses or solving systems, over an `IntegralDomain` rather than over a `Field` as is usually done. Those fraction–free algorithms are significantly faster over integral domains than over their fraction fields, so consider using domains such as $\mathbb{Z}$ or $R[x]$ as matrix entries rather than fractions. See for example the description of the `inverse` function to see the effect of working over domains rather than fields on the signatures of the functions.

To write generic linear algebra code that does not depend on the implementation of matrices, use a type parameter of category `MatrixCategory`. For example, a normal form package could look like:

```
NormalForms(R:IntegralDomain, M:MatrixCategory R): with {
        frobenius: M -> M;     -- Returns the Frobenius form of its argument
        ...
} == add { ... }
```

Univariate polynomials and series

The basic types to use for computing with polynomials and series are `DenseUnivariatePolynomial` and `DenseUnivariateTaylorSeries` respectively. Both types are univariate but can be nested if needed to produce dense multivariate polynomials and series with a fixed number of variables. When the number of variables is too large for a dense representation, you can also use `SparseUnivariatePolynomial` but be aware that for univariate or bivariate use, its arithmetic is much less efficient than the one of `DenseUnivariatePolynomial`.

As for matrices, the type of the coefficients of polynomials or series does not need to be a `Ring`, it can be an `ArithmeticType` instead. This allows types that do not have a full equality such as `SingleFloat` to be used as coefficients, but some polynomial functionalities are only available when the coefficient type is a `Ring` or something stronger. The polynomial and series types take a `Symbol` as second argument. That symbol is used only for output when converting the polynomial or series to an `ExpressionTree`, so it is not necessary to give one when you use polynomials or series inside a calculation. If you insist on naming the variable, use `-` with any string as argument to create a symbol. For example, `DenseUnivariatePolynomial(Integer)` and `SparseUnivariatePolynomial(Fraction Integer, -"x")` are both valid polynomial types. Whether you give a name for a variable or not, you can override that choice using `apply` with a `Symbol` or `ExpressionTree` as argument. For example, if p is a polynomial or series, `stdout(p, -"z")` writes p to `stdout` using "z" as variable name, and `p(extree leaf(-"y"))` returns the `ExpressionTree` corresponding to p with "y" as variable name.

To write generic code for manipulating polynomials or series use a type parameter usually of category `UnivariatePolynomialCategory` or `UnivariateTaylorSeriesCategory` respectively.

Because polynomials, skew–polynomials and series share many common operations, `Algebra` provides in fact a complete category hierarchy for them, shown in Figure 2. Those categories make it

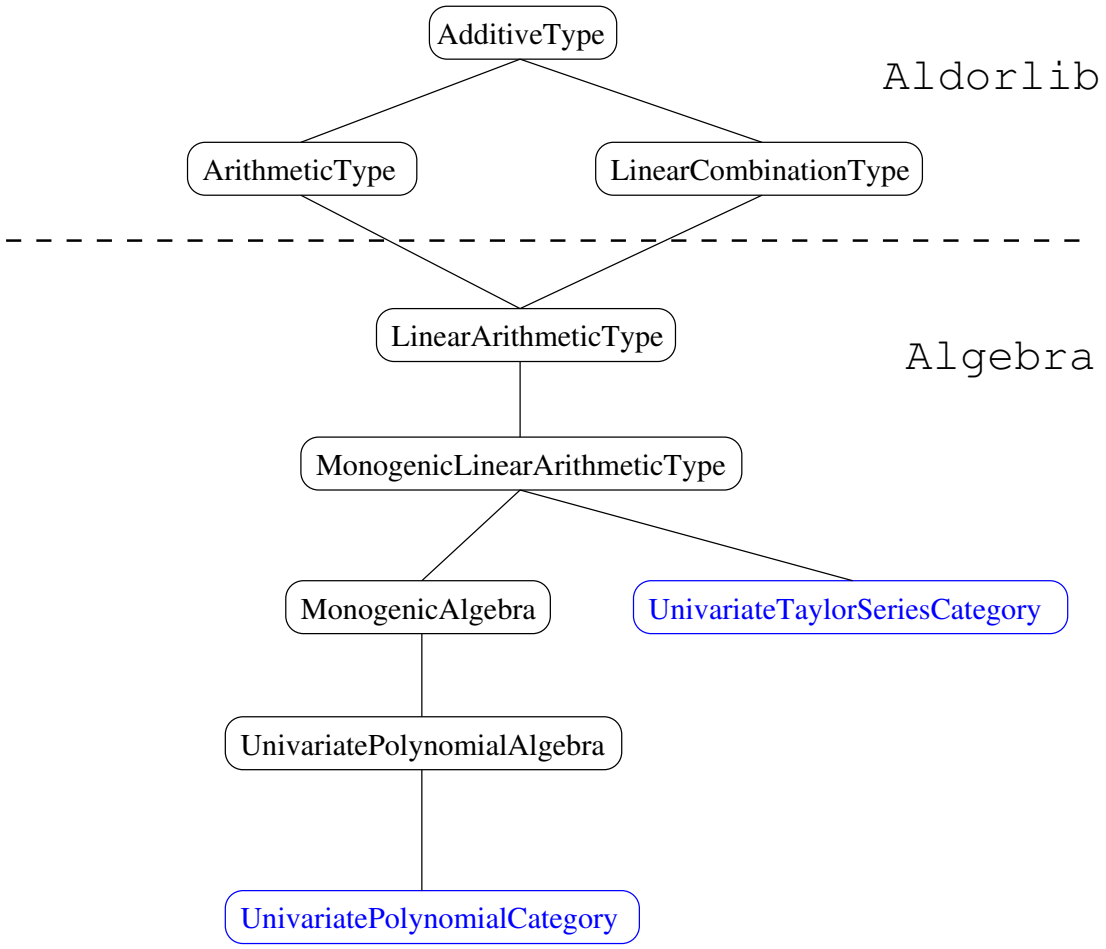

Figure 2: The `Algebra` univariate polynomial category hierarchy

possible to write functions that work for polynomials, skew–polynomials, series or any combination of them. `UnivariatePolynomialCategory`(R) is the category for types whose elements are usual polynomials of the form $\sum_n r_n x^n$ with finite support. While R is not assumed to be commutative, the generator x is assumed to commute with coefficients in R, *i.e.* $rx = xr$. Similarly, `UnivariateTaylorSeriesCategory`(R) is the category for types whose elements are series of the form $\sum_n r_n x^n$. Here also, R is not assumed to be commutative, but the generator x commutes with coefficients in R. Those two categories are shown in blue in Figure 2, as they are the only ones assuming that x commutes with R. The more general category `UnivariatePolynomialAlgebra`(R) is for types whose elements are sums of the form $\sum_n r_n x^n$ with finite support, but where x does not necessarily commute with R. Polynomials and skew-polynomials are both in that category. Even more general, `MonogenicAlgebra`(R) is for types whose elements are sums of the form $\sum_n r_n P_n$ with finite support, but where the family P_n is not necessarily the power basis x^n. An example of such a type is `UnivariateFactorialPolynomial` for which $P_n = x(x-1)\dots(x-n+1)$. Finally, `MonogenicLinearArithmeticType`(R) is for types whose elements are potentially infinite sums of

the form $\sum_n r_n P_n$. Code written at that level works for polynomials, series and skew-polynomials as well.

The elements of `DenseUnivariateTaylorSeries` are lazy series represented by their coefficient sequences, themselves of type `Sequence`. Those coefficient sequences are in turn represented as `Stream` from `libaldor`. Therefore, the usual way to write a function producing a series is to produce first its coefficient stream s, then call `sequence` on s to produce the coefficient sequence, and finally call `series` on the coefficient sequence to produce the series. Since streams are lazy, constructing a series does not compute any of its coefficients until they are specifically requested by another operation. There are several ways to create streams, all documented in the `libaldor` reference manual, and you should become familiar with them before programming with series. For example, the following function takes constants a_0 and c and produces the hypergeometric series $\sum_{n\geq 0} a_n x^n$ where $a_{n+1}/a_n = c$.

```
SeriesSample(R:CommutativeRing, Rx:UnivariateTaylorSeriesCategory R): with {
        hypergeometricSeries: (R, R) -> Rx;
} == add {
        hypergeometricSeries(a0:R, c:R):Rx == {
                import from Sequence F;
                zero? a0 => 0;
                -- the following creates the stream [a0, c a0, c^2 a0, ... ]
                coeffs:Stream R := orbit(a0, (x:R):R +-> c * x);
                series sequence coeffs;
        }
}
```

Finally, the type `UnivariateTaylorSeriesCategory2Poly` provides conversions between univariate polynomials and series.

Multivariate polynomials

The basic types for computing with multivariate polynomials are `SparseMultivariatePolynomial` and `DistributedMultivariatePolynomial1` Both types use a sparse representation. Elements of the former one are regarded as univariate polynomials with polynomial coefficients. Elements of the latter one are regarded as lists of terms where a term is the product of a coefficient by a power product of variables.

Note: the type `SparseMultivariatePolynomial` will be included in a forthcoming update of the library, in the meantime, you can only use `DistributedMultivariatePolynomial1` for manipulating multivariate polynomials, as in the following example:

```
#include "algebra"
import from String, Symbol;
macro V == OrderedVariableTuple(-"x",-"y",-"z");
import from MachineInteger, V;
macro E == MachineIntegerDegreeLexicographicalExponent(V);
macro P == DistributedMultivariatePolynomial1(Integer, V, E);
x: P := variable(1)$V :: P;
```

```
y: P := variable(2)$V :: P;
z: P := variable(3)$V :: P;
p := (y + z)*x^2 + (y^3 + z)*x + z^4;
```

Compatibility with C types

All of `libaldor`'s types that are compatible with their C counterparts remain so inside `Algebra`. In addition, if you are using the GMP version of the library, then `Integer` and `Float` are compatible with `mpz_t` and `mpf_t` respectively.

Using GMP

As in `libaldor`, the type `Integer` is actually a macro, which defaults to `AldorInteger`, the software integers provided by the Aldor runtime. For efficiency or other reasons, you may prefer to use the GMP library, which is supported by `Algebra`. The easiest way to use GMP is to compile all your source files with the option `-dGMP` and then to use the options

```
-lalgebra-gmp -laldor -cruntime=foam-gmp,gmp -y$ALGEBRAROOT/lib -y$ALDORLIBROOT/lib
```

when linking your final executable. All you need is GMP 3.0 or later installed in a file called `libgmp.a` to produce executables. Using GMP generally produces more efficient programs, but programs calling GMP cannot be interpreted by the Aldor compiler, nor can they run inside its interactive loop.

Using the `-dGMP` option allows you to compile the same sources either with or without GMP, which can be appreciable, but you must ensure that you do not mix files compiled with and without `-dGMP` since the macro `Integer` would then be expanded into two different types.

An additional advantage of using GMP, is that `GMPInteger` exports and uses internally several of the in–place or higher–level operations of GMP, which are not available with `AldorInteger`. In addition, variables of type `GMPInteger` are compatible with the C type `mpz_t` from GMP, so you can directly call C programs that use GMP from your Aldor code.

Exceptions

In addition to the exceptions thrown by `libaldor`, `Algebra` throws a `ReducibleModulusException` when a divisor of zero is discovered in a `SimpleAlgebraicExtension`. This can be used to implement algorithms based on lazy factorisation, since such an exception contains a non–trivial factor of the polynomial defining the extension.

Profiling and debugging

The macros TIME, TRACE and AGAT provided by `libaldor` remain available in `Algebra`. In addition, `Algebra` also comes with a debug version, which makes many assertions about the arguments of the functions called as well as their results. While those assertions slow down the code considerably they tend to be quite useful when chasing bugs since the release version of `Algebra`

does not make validity checks to most of its inputs. The debug version of `Algebra` must be used jointly with the debug version of `libaldor`. To use them, just compile your application with the

```
-lalgebrad -laldord -dDEBUG
```

options rather than `-lalgebra -laldor`. It is also preferable when debugging to add the `-q1` option in order to prevent inlining.

3 Reference Manual

AbelianGroup

Usage

AbelianGroup: Category

Description

AbelianGroup is the category of commutative groups.

Exports

`AdditiveType`
`AbelianMonoid`

Usage

AbelianMonoid: Category

Description

AbelianMonoid is the category of commutative monoids.

Exports

`ExpressionType`

0:	%	zero
+:	(%, %) → %	sum
∗:	(`Integer`, %) → %	product by an integer
`add!`:	(%, %) → %	In–place sum
`zero?`:	% → `Boolean`	test for 0

Usage

0

Signature

0: %

Returns

Returns the constant 0.

Usage

$x + y$

Signature

+: (%,%) → %

Parameter	Type	Description
x,y	%	elements of the monoid

Returns

Returns the sum $x + y$.

Usage

n $*$ x

Signature

$*$: (Integer, %) $\to$ %

Parameter	Type	Description
n	Integer	An integer
x	%	An element of the monoid to be multiplied by n

Returns

Returns the product nx.

Usage

add!(x, y)

Signature

add!: (%, %) → %

Parameter	Type	Description
x	%	An element of the monoid (to be destroyed)
y	%	An element of the monoid to be added to x

Returns

Returns the sum $x + y$, where the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

This function may cause x to be destroyed, so do not use it unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

zero? x

Signature

zero?: % → `Boolean`

Parameter	Type	Description
x	%	an element of the monoid

Returns

Returns the result of $x = 0$ using the semantics of $=$ of the monoid.

Usage

Algebra R:Category

Parameter	Type	Description
R	`Ring`	The coefficient ring

Description

Algebra R is the category of algebras over R.

Exports

`LinearArithmeticType` R
`Module` R
`Ring`

CharacteristicZero

Usage

CharacteristicZero: Category

Description

CharacteristicZero is the category of rings of characteristic 0.

Exports

`Ring`

Usage

CommutativeRing: Category

Description

CommutativeRing is the category of commutative rings.

Exports

`Ring`

`canonicalUnitNormal?`:	→ `Boolean`	Check if `unitNormal` is canonical
`cutoff`:	`MachineInteger` → `MachineInteger`	Cutoffs for various fast algorithms
`reciprocal`:	% → `Partial` %	Inverse
`unitNormal`:	% → (%,%,%) (%, %) → (%,%)	Representative of the associates
`unit?`:	% → `Boolean`	Test whether an element is a unit

Usage

canonicalUnitNormal

Signature

canonicalUnitNormal: `Boolean`

Returns

Returns *true* if `unitNormal` is canonical in the following sense: if $x = vx'$ for some unit v and `unitNormal`(x) returns (u, y, u^{-1}) and `unitNormal`(x') returns (u', y', u'^{-1}), then $y = y'$.

See also

`unitNormal`, `unit?`

Usage

cutoff t

Signature

cutoff: `MachineInteger` → `MachineInteger`

Returns

Returns various cutoffs for asymptotically fast algorithms, which are then used for structures (*e.g.* polynomials or matrices) over this ring when the input size is greater than the corresponding cutoff. The parameter t denotes the algorithm in question and must be one of the CUTOFF_XXX values defined in `include/algebrauid.as`

Remarks

If a cutoff is -1, then the corresponding algorithm is not used at all over this ring. The default value is always -1 so you only need to define other values if you want particular algorithms to be used over your rings.

Usage

reciprocal x

Signature

reciprocal: % → `Partial` %

Parameter	Type	Description
x	%	An element of the ring

Returns

Returns the unique y such that $x\,y = 1$ if such a y exists, *failed* otherwise.

Usage

(y, u, u1) := unitNormalx
(y, z1) := unitNormal(x, z)

Signatures

unitNormal: % → (%, %. %)
unitNormal: (%,%) → (%. %)

Parameter	Type	Description
x,y	%	Elements of the ring

Returns

unitNormal(x) returns (y, u, u^{-1}), while unitNormal(x,z) returns $(y, u^{-1}z)$. In both cases, $x = uy$ and u is a unit.

See also

`canonicalUnitNormal?`, `unit?`

Usage

unit? x

Signature

unit?: % → `Boolean`

Parameter	Type	Description
x	%	An element of the ring

Returns

Returns *true* if x is a unit, *i.e.*$x\,y = 1$ for some y, *false* otherwise.

See also

`canonicalUnitNormal?`, `unitNormal`

Usage

DecomposableRing: Category

Description

DecomposableRing is the category of commutative rings whose elements can sometimes be decomposed into products (not to be confused with true factorization).

Exports

`CommutativeRing`

`provablyIrreducible?`:	% → `Boolean`	
`someFactors`:	% → `List` %	Get some factors

Usage

provablyIrreducible? x

Signature

provablyIrreducible?: % → `Boolean`

Parameter	Type	Description
x	%	A ring element

Returns

Returns *true* if x can be proven to be irreducible, *false* if either x is reducible or the proof of irreducibility cannot be obtained quickly enough.

Remarks

This function is not meant to use factorization or catch all irreducible elements, even when those functionalities are available. It is however meant to be efficient.

Usage

someFactors x

Signature

someFactors: $\% \to$ `List` $\%$

Parameter	Type	Description
x	$\%$	A ring element

Returns

Returns $[x_1, \ldots, x_n])$ such that each x_i divides x exactly.

Remarks

This function is not meant to use factorization or return a complete decomposition, even when those functionalities are available. It is however meant to be efficient.

Usage

DifferentialExtension R: Category

Parameter	Type	Description
R	`CommutativeRing`	The base ring

Description

DifferentialExtension(R) is the category of differential extensions of R.

Exports

`CommutativeRing`

`lift`: `Derivation` R → `Derivation` % Extension of a derivation

if R has `DifferentialRing` then

`DifferentialRing`

Usage

lift D

Signature

lift: `Derivation` R → `Derivation` %

Parameter	Type	Description
D	`Derivation` R	A derivation on R

Returns

Returns the derivation *D* extended to the ring extension.

Usage

DifferentialRing: Category

Description

DifferentialRing is the category of commutative differential rings.

Exports

`CommutativeRing`

`derivation`:	$\to$ `Derivation` %	The derivation
`differentiate`:	(%, `Integer`) $\to$ %	Differentiate an element

Usage
derivation

Signature
derivation: `Derivation` %

Returns
Returns the derivation of the ring.

See also
differentiate(DifferentialRing)

Usage

differentiate x
differentiate(x, n)

Signature

differentiate: (%, `Integer`) → %

Parameter	Type	Description
x	%	The element to differentiate
n	`Integer`	The order of differentiation (optional)

Returns

differentiate x returns x', the derivative of x.
differentiate(x, n) returns $x^{(n)}$, the n^{th} derivative of x.

See also

derivation(DifferentialRing)

Usage

EuclideanDomain: Category

Description

EuclideanDomain is the category of commutative Euclidean domains.

Exports

`GcdDomain`

`diophantine`:	(%, %, %) → `Partial` %	Solve a linear diophantine equation
`divide`:	(%, %) → (%, %)	Euclidean division
`divide!`:	(%, %, %) → (%, %)	In–place Euclidean division
`euclid`:	(%, %) → %	Euclidean gcd
`euclid!`:	(%, %) → %	In–place Euclidean gcd
`euclideanSize`:	% → `Integer`	Size function of the domain
`extendedEuclidean`:	(%, %) → (%, %, %)	Extended Euclidean Algorithm
	(%, %, %) → `Partial`(%, %)	
`quo`:	(%, %) → %	Quotient
`rem`:	(%, %) → %	Remainder
`remainder!`:	(%, %) → %	In–place remainder

Usage

diophantine(a, b, m)

Signature

diophantine: (%, %, %) → `Partial` %

Parameter	Type	Description
a	%	An element of the ring
b	%	The right hand side of the equation
m	%	The nonzero modulus

Returns

If the diophantine equation $ax \equiv b \pmod{m}$ has solutions in the ring, returns a solution x such that either $x = 0$ or $|x| < |m|$. Returns *failed* if the equation has no solution.

Usage

divide(a, b)
a quo b
a rem b

Signatures

divide: (%,%) → (%, %)
quo,rem: (%,%) → %

Parameter	Type	Description
a,b	%	Element of the ring, $y \neq 0$

Returns

a rem b returns r such that either $r = 0$ or $0 \leq |r| < |b|$ and $a \equiv r \pmod{b}$, a quo b returns q such that $a - bq = 0$ or $0 \leq |a - bq| < |b|$, and divide(a, b) returns the pair (a quo b, a rem b).

Usage

divide!(x, y, z)

Signature

divide!: (%, %, %) → (%, %)

Parameter	Type	Description
x	%	An element of the ring (to be destroyed)
y	%	An element of the ring
z	%	A placeholder for the quotient (to be destroyed)

Returns

Returns (q, r) such that $x = qy + r$ and either $r = 0$ or $0 \leq |r| < |x|$, where the storage used by x and z is allowed to be destroyed or reused, so x and z is lost after this call.

Remarks

This function may cause x and z to be destroyed, so do not use it unless x and z have been locally allocated, and are guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

`remainder!`

Usage

euclid(x, y)
euclid!(x, y)

Signature

euclid: (%, %) → %

Parameter	Type	Description
x,y	%	Elements of the ring

Returns

Returns gcd(x, y) computed by the Euclidean algorithm. When using euclid!(x, y), the storage used by x and y is allowed to be destroyed or reused, so x and y are lost after this call.

Remarks

The call euclid!(x, y) may cause x and y to be destroyed, so do not use it unless x and y have been locally allocated, and are guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

`gcd`, `gcd!`

Usage

euclideanSize x

Signature

euclideanSize: % → `Integer`

Parameter	Type	Description
x	%	A nonzero element of the ring

Returns

Returns $|x|$, the euclidean size of x. It is connected to the Euclidean remainder, in that the remainder r of a by b is either 0, or satisfies $0 \leq |r| < |b|$.

Usage

extendedEuclidean(a, b)
extendedEuclidean(a, b, c)

Signatures

extendedEuclidean: (%, %) → (%, %, %)
extendedEuclidean: (%, %, %) → `Partial`(%, %)

Parameter	Type	Description
a,b,c	%	Elements of the ring

Returns

extendedEuclidean(a, b) returns (g, x, y) such that $g = \gcd(a, b) = ax + by$.
extendedEuclidean(a, b, c) returns either a solution (x, y) of the diophantine equation $ax+by = c$, or *failed* if it has no solution in the ring.
For the values returned by both calls, either $x = 0$ or $|x| < |b|$.

Usage

remainder!(x, y)

Signature

remainder!: (%, %) → %

Parameter	Type	Description
x	%	An element of the ring (to be destroyed)
y	%	An element of the ring

Returns

Returns the remainder of x by y, where the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

This function may cause x to be destroyed, so do not use it unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Field

Usage

Field: Category

Description

Field is the category of commutative fields.

Exports

`EuclideanDomain`
`Group`

Usage

FiniteField: Category

Description

FiniteField is the category of finite fields.

Exports

`Field`
`FiniteCharacteristic`
`FiniteSet`

`degree`:	`Integer`	Dimension over the prime field
`pthRoot`:	$\% \to \%$	Exponentiation to 1 over the characteristic
`pthRoot!`:	$\% \to \%$	In-place exponentiation to 1 over the characteristic

Usage

degree

Signature

degree: `Integer`

Returns

Returns the extension degree of the field, *i.e.* its dimension as a vector space over its prime fields. In other words, the size of the field is $p^d egree$ where p is the characteristic.

Usage

pthRoot x
pthRoot! x

Signature

pthRoot: $\% \to \%$

Parameter	Type	Description
x	$\%$	An element of the ring

Returns

Return y such that $y^p = x$ where p is the characteristic of the field.

Remarks

pthRoot! does not make a copy of x, which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := pthPower! x`.

Usage

FiniteCharacteristic: Category

Description

FiniteCharacteristic is the category of finite characteristic rings.

Exports

`Ring`

`pthPower`:	$\% \to \%$	Exponentiation to the characteristic
`pthPower!`:	$\% \to \%$	In–place exponentiation to the characteristic

Usage

pthPower x
pthPower! x

Signature

pthPower: $\% \to \%$

Parameter	Type	Description
x	$\%$	An element of the ring

Returns

Return x^p where p is the characteristic of the ring.

Remarks

pthPower! does not make a copy of x, which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := pthPower! x`.

Usage

FiniteSet: Category

Description

FiniteSet is the category of finite sets.

Exports

`ExpressionType`
`TotallyOrderedType`

`#`:	`Integer`	number of elements
`apply`:	(%, `ExpressionTree`) → `ExpressionTree`	Conversion to an expression tree
`index`:	% → `Integer`	Index of an element
`lookup`:	`Integer` → %	Element with a given index
`random`:	() → %	Get a random element

Usage

#

Signatures

#: `Integer`

Returns

Returns the number of elements of the type.

Usage

apply(p, x)
p x

Signature

apply: (%, `ExpressionTree`) → `ExpressionTree`

Parameter	Type	Description
p	%	An element
x	`ExpressionTree`	A name for the variables

Returns

Returns p as an expression tree, using x as root variable name.

Usage

index p

Signature

index: % → `Integer`

Parameter	Type	Description
p	%	An element

Returns

Returns the index of *p*.

See also

lookup(FiniteSet)

Usage
lookup j

Signature
lookup: `Integer` → %

Parameter	Type	Description
j	`Integer`	An index

Returns
Returns the element with index j.

See also
index(FiniteSet)

Usage
random()

Signature
random: () → %

Returns
Returns a random element.

Usage

FreeAlgebra R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

FreeAlgebra R is a category for free algebras over an arbitrary arithmetic system R and with respect to an arbitrary basis. Its elements are assumed to have finite support.

Exports

`FreeLinearArithmeticType` R
`FreeModule` R

If R has `CharacteristicZero` then
`CharacteristicZero`

If R has `FiniteCharacteristic` then
`FiniteCharacteristic`

If R has `Ring` then
`Algebra` R

if R has `RittRing` then
`RittRing`

Usage

FreeLinearArithmeticType R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

FreeLinearArithmeticType R is the category of arithmetic types containing linear combinations of their elements with coefficients in R with respect to a linearly independent generating set. Its elements are not assumed to have finite support, so this type cannot be asserted to be a R even when R is a `Ring`.

Exports

`FreeLinearCombinationType` R
`LinearArithmeticType` R

Usage

FreeLinearCombinationType R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

FreeLinearCombinationType R is the category of types containing linear combinations of their elements with coefficients in R with respect to a linearly independent generating set. Its elements are not assumed to have finite support, so this type cannot be asserted to be a R even when R is a `Ring`.

Exports

`ExpressionType`
`LinearCombinationType` R

`map`:	$(\mathrm{R} \to \mathrm{R}) \to \% \to \%$	Lift a mapping
`map!`:	$(\mathrm{R} \to \mathrm{R}) \to \% \to \%$	Lift a mapping

Usage

map f
map! f
map(f)(m)
map!(f)(m)

Signature

map: (R → R) → % → %

Parameter	Type	Description
f	R → R	A map
m	%	An element of the module

Description

map(f)(m) returns

$$f(m) = \sum_i f(r_i)e_i$$

where $m = \sum_i r_i e_i$, while map(f) returns the mapping $m \to f(m)$. In both cases, map! does not make a copy of m but modifies it in place.

Usage

FreeModule R: Category

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	

Description

FreeModule is a category for free modules over an arbitrary arithmetic system R and with respect to an arbitrary basis. Its elements are assumed to have finite support.

Exports

`FreeLinearCombinationType` R

`ground?`:	% → `Boolean`	Test for a ground element
`leadingCoefficient`:	% → R	The leading coefficient
`nonZeroCoefficients`:	% → `Generator` R	Iterate over the coefficients
`reductum`:	% → %	All terms except the leading one
`reductum!`:	% → %	All terms except the leading one
`support`:	% → `Generator Cross`(R, %)	Make an iterator
`term?`:	% → Boolean	Test for a monomial
`trailingCoefficient`:	% → R	The trailing coefficient

If R has `Ring` then
`Module` R

If R has `GcdDomain` then

`content`:	% → R	Content
`primitive`:	% → (R, %)	Content and primitive part
`primitive!`:	% → (R, %)	Content and primitive part
`primitivePart`:	% → %	Primitive part
`primitivePart!`:	% → %	Primitive part

If R has `Field` then

`monic`:	% → %	Make monic
`monic!`:	% → %	Make monic

Usage

content m
content! m
primitive m
primitive! m
primitivePart m
primitivePart! m

Signatures

content,content!:	% → R
primitive,primitive!:	% → (R, %)
primitivePart,primitivePart!:	% → %

Parameter	Type	Description
m	%	An element of the module

Description

content(m) returns the gcd of the coefficients of m, while primitive(m) and primitivePart(m) return respectively $(c, c^{-1}m)$ and $c^{-1}m$ where $c = \text{content}(p)$.

Remarks

The storage used by m is allowed to be destroyed or reused if content!, primitive! or primitivePart! is used, so m is lost after those calls. This may cause m to be destroyed, so do not use this unless m has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

ground? m
term? m

Signature

ground?,term?: % → `Boolean`

Parameter	Type	Description
m	%	An element of the module

Returns

ground?(m) returns *true* if m is an element of R, *false* otherwise, while term?(m) returns *true* if $m = re$ for $a \in R$ and e an element of the basis, *false* otherwise.

Remarks

term?(0) returns *true*.

Usage

leadingCoefficient m
trailingCoefficient p

Signature

trailingCoefficient: $\% \to R$

Parameter	Type	Description
m	$\%$	An element of the module

Returns

leadingCoefficient(m) and trailingCoefficient(m) return respectively the leading and trailing coefficient of m. Both return 0 when $p = 0$.

See also

`coefficients`, `nonZeroCoefficients`

Usage

monic m
monic! m

Signature

monic: $\% \to \%$

Parameter	Type	Description
m	$\%$	An element of the module

Returns

Returns $r^{-1}m$ where r is the leading coefficient of m, returns 0 if $m = 0$.

Remarks

The storage used by m is allowed to be destroyed or reused if monic! is used, so m is lost after this call. This may cause m to be destroyed, so do not use this unless m has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

for c in nonZeroCoefficients m repeat { ... }

Signature

nonZeroCoefficients: % → `Generator` R

Parameter	Type	Description
m	%	An element of the module

Returns

Returns a generator that produces all the nonzero coefficients of m.

Usage

reductum m
reductum! m

Signature

reductum: $\% \to \%$

Parameter	Type	Description
m	$\%$	An element of the module

Returns

Returns the reductum of m, *i.e.* $p - re$ where r is the leadingCoefficient of p and e the corresponding element. Returns 0 if $m = 0$.

Remarks

The storage used by m is allowed to be destroyed or reused if reductum! is used, so m is lost after this call. This may cause m to be destroyed, so do not use this unless m has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

support m

Signature

support: % → `Generator Cross`(R, %)

Parameter	Type	Description
m	%	An element of the module

Description

support(m) generates the terms of m, *i.e.* the smallest number of pairs (r, e) where e lies in the basis and whose sum is m.

GcdDomain

Usage

GcdDomain: Category

Description

GcdDomain is the category of commutative Gcd domains.

Exports

`IntegralDomain`

`gcd`:	(%, %) → %	Gcd of 2 elements
`gcd`:	`Generator` % → %	Gcd of several elements
`gcd!`:	(%, %) → %	In-place gcd of 2 elements
`gcdquo`:	(%, %) → (%, %, %)	Gcd with quotients
`gcdquo`:	`List` % → (%, `List` %)	Gcd with quotients
`lcm`:	(%, %) → %	Lcm of 2 elements
`lcm`:	`List` % → %	Lcm of several elements

Usage

gcd(x_1, x_2)
gcd!(x_1, x_2)
gcd g
lcm(x_1, x_2)
lcm $[x_1, \ldots, x_n]$

Signatures

gcd,lcm:	(%, %) → %
gcd!:	(%, %) → %
gcd:	`Generator` % → %
lcm:	`List` % → %

Parameter	Type	Description
x_i	%	Elements of the ring
g	`Generator` %	Generates elements of the ring

Returns

gcd(x_1, x_2) and lcm(x_1, x_2) return respectively a greatest common divisor and least common multiple of x_1 and x_2, while gcd(g) return a greatest common divisor of all the elements generated by g and lcm($[x_1, \ldots, x_n]$) return a least common multiple of the x_i.

Remarks

With certain types, for example polynomials, the generator version can be more efficient than iterating the binary version. The function gcd! may cause x_1 and x_2 to be destroyed, so do not use it unless x_1 and x_2 have been locally allocated, and are guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

`gcdquo`

Usage

gcdquo(x_1, x_2)
gcdquo $[x_1, \ldots, x_n]$

Signatures

gcdquo: (%,%) → (%,%,%)
gcdquo: `List` % → (%, `List` %)

Parameter	Type	Description
x_i	%	Elements of the ring

Returns

gcdquo(x_1, x_2) returns (g, y_1, y_2) where $g = \gcd(x_1, x_2)$, $x_1 = gy_1$ and $x_2 = gy_2$,
gcdquo$([x_1, \ldots, x_n])$ returns $(g, [y_1, \ldots, y_n])$ where $g = \gcd(x_1, \ldots, x_n)$ and $x_i = gy_i$.

Remarks

With certain types, for example polynomials, the n-ary version can be more efficient than iterating the binary version.

See also

`gcd`

Group

Usage

Group: Category

Description

Group is the category of groups, not necessarily commutative.

Exports

`Monoid`

`/`:	(%, %) → %	quotient
`inv`:	% → %	inverse

Usage

x/y

Signature

/: $\% \to \%$

Parameter	Type	Description
x,y	$\%$	Elements of the group

Returns

Returns xy^{-1}.

Usage

inv x

Signature

inv: $\% \to \%$

Parameter	Type	Description
x	$\%$	An element of the group

Returns

Returns x^{-1}.

Usage

IndexedFreeAlgebra R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain
E	`ExpressionType` `TotallyOrderedType`	The index domain

Description

IndexedFreeAlgebra(R, E) is a category for free algebras over an arbitrary arithmetic system R, with respect to a linearly independent generating set E. Its elements are assumed to have finite support.

Exports

`FreeAlgebra` R
`IndexedFreeModule`(R, E)
`IndexedFreeLinearArithmeticType`(R, E)

Usage

IndexedFreeLinearArithmeticType R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain
E	`ExpressionType`	The index domain

Description

IndexedFreeLinearArithmeticType R is the category of arithmetic types containing linear combinations of their elements with coefficients in R with respect to a linearly independent generating set. Its elements are not assumed to have finite support, so this type cannot be asserted to be a `Algebra` R even when R is a `Ring`.

Exports

`IndexedFreeLinearCombinationType` R
`FreeLinearArithmeticType` R

Usage

add!(p, c, e, q)

Signature

add!: (%, R, E, %) → %

Parameter	Type	Description
p	%	An element of the module (to be destroyed)
c	R	A scalar
e	E	The degree of the term to add
q	%	An element of the module

Returns

add!(p, c, e, q) computes the sum $p + ceq$.

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other polynomials.

Usage

IndexedFreeLinearCombinationType R: Category

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	
E	`ExpressionType`	The index domain

Description

IndexedFreeLinearCombinationType(R, E) is the category of types containing linear combinations of their elements with coefficients in R with respect to a linearly independent generating set. Its elements are not assumed to have finite support, so this type cannot be asserted to be a `Module` R even when R is a `Ring`.

Exports

`FreeLinearCombinationType` R

`add!`:	(%, R, E) → %	In–place addition of a term
`coefficient`:	(%, Z) → R	Extraction of a coefficient
`monomial`:	E → %	Creation of a monic term
`term`:	(R, E) → %	Creation of a term

Usage

add!(p, c, e)

Signature

add!: (%, R, E) → %

Parameter	Type	Description
p	%	An element of the module (to be destroyed)
c	R	A scalar
e	E	The degree of the term to add

Returns

add!(p, c, e) computes the sum $p + ce$.

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other polynomials. Some functions, like `reductum` are not necessarily copying their arguments and can thus create memory aliases.

Usage

coefficient(p, e)

Signature

coefficient: (%, E) → R

Parameter	Type	Description
p	%	An element of the module
e	E	An exponent

Returns

Returns the coefficient of e in p.

Usage

monomial(e)
term(r, e)

Signatures

monomial: E → %
term: (R, E)→ %

Parameter	Type	Description
r	R	A scalar
e	E	An exponent

Returns

monomial(e) and term(r, e) return respectively the monomial e and the term re.

Usage

IndexedFreeModule R: Category

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	
E	`ExpressionType`	The index domain
	`TtotallyOrderedType`	

Description

IndexedFreeModule(R, E) is a category for free modules over an arbitrary arithmetic system R, with respect to a linearly independent generating set E. Its elements are assumed to have finite support.

Exports

`FreeModule` R
`IndexedFreeLinearCombinationType`(R, E)

`degree`:	% → E	degree
`generator`:	% → `Generator Cross`(R, E)	iterate through all the terms
`leadingMonomial`:	% → %	leading monomial
`leadingTerm`:	% → (R, E)	leading term
`terms`:	% → `Generator Cross`(R, E)	iterate through all the terms
`trailingDegree`:	% → E	trailing degree
`trailingMonomial`:	% → %	trailing monomial
`trailingTerm`:	% → (R, E)	trailing term

if R has `HashType` and E has `HashType` then
`HashType`

if R has `SerializableType` and E has `SerializableType` then
`SerializableType`

Usage

degree p
trailingDegree p

Signature

degree,trailingDegree: % → E

Parameter	Type	Description
p	%	A nonzero element of the module

Returns

degree(p) and trailingDegree(p) return respectively the degree and trailing degree of p, *i.e.*e_n and e_m where $p = \sum_{i=m}^{n} a_i e_i$ and $a_n \neq 0 \neq a_m$. Both functions are undefined when $p = 0$.

See also

`leadingCoefficient`,`leadingMonomial`, `trailingCoefficient`,`trailingMonomial`

Usage

for term in p repeat { (c, n) := term; ... }
for term in generator p repeat { (c, n) := term; ... }
for term in terms p repeat { (c, n) := term; ... }

Signature

generator,terms: % → `Generator Cross`(R, E)

Parameter	Type	Description
p	%	An element of the module

Description

Both functions allow an element of the module to be iterated independently of its representation. Both generators yield pairs of the form (a, e), with $a \neq 0$. The difference between the two is that generator(p) yields the terms in decreasing exponents, while terms(p) yields the terms in increasing exponents.

Example

```
import from Integer, DenseUnivariatePolynomial Integer;

x := monom;
p := x^3 + 2*x - 1;
for term in p repeat { (c, n) := term; print << c << "," << n << newline }
```

writes

```
1,3
2,1
-1,0
```

to the standard stream print.

See also

`coefficients`, `revert`

Usage

leadingMonomial p
(r, e) := leadingTerm p
trailingMonomial p
(r, e) := trailingTerm p

Signatures

leadingMonomial, trailingMonomial: % → %
leadingTerm, trailingTerm: % → (R, E)

Parameter	Type	Description
p	%	An element of the module

Returns

When $p = \sum_{i=m}^{n} a_i e_i$ and $a_n \neq 0 \neq a_m$, then leadingMonomial(p) and trailingMonomial(p) return respectively e_n and e_m, while then leadingTerm(p) and trailingTerm(p) return respectively (a_n, e_n) and (a_m, e_m). When $p = 0$, leadingMonomial(0) and trailingMonomial(0) both return 0, while leadingTerm(p) and trailingTerm(p) are undefined.

See also

`degree`,`leadingCoefficient`, `trailingCoefficient`,`trailingDegree`

Usage

IntegerCategory: Category

Description

IntegerCategory is the category of integer-like rings.

Exports

`CharacteristicZero`
`EuclideanDomain`
`IntegerType`
`Parsable`
`Specializable`
`integer`: $\% \to$ `Integer` Conversion to an integer

Usage

binomial(n, m)

Signature

binomial: (%,%) → %

Parameter	Type	Description
n,m	%	Integers

Returns

Returns

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}\,.$$

Returns 0 if either $m < 0$ or $n < m$.

Usage

integer n

Signature

integer: % → `Integer`

Parameter	Type	Description
n	%	An integer

Returns

Returns n as an integer.

Usage

IntegralDomain: Category

Description

IntegralDomain is the category of commutative integral domains.

Exports

`CommutativeRing`

`divDiffProd`:	(%, %, %, %, %) → %	Combined product and quotient
`divSumProd`:	(%, %, %, %, %, %, %) → %	Combined product and quotient
`exactQuotient`:	(%, %) → `Partial` %	Exact quotient
`order`:	% → % → `Integer`	Order of divisibility
`orderquo`:	% → % → (`Integer`, %)	Order of divisibility and quotient
`quotient`:	(%, %) → %	Exact quotient
`quotient!`:	(%, %) → %	In-place exact quotient
`quotientBy`:	% → (% → %)	Exact quotient procedure
`quotientBy!`:	% → (% → %)	In-place exact quotient procedure

Usage

divDiffProd(a1, a2, b1, b2, q)

Signature

divDiffProd: (%, %, %, %, %) → %

Parameter	Type	Description
a1, a2, b1, b2, q	%	Elements of the ring

Returns

divDiffProd(a1, a2, b1, b2, q) returns $(a1\,a2 - b1\,b2)/q$.

Usage

divSumProd(a1, a2, b1, b2, c1, c2, q)

Signature

divSumProd: (%, %, %, %, %, %, %) → %

Parameter	Type	Description
a1, a2, b1, b2, c1, c2, q	%	Elements of the ring

Returns

divSumProd(a1, a2, b1, b2, c1, c2, q) returns $(a1\,a2 + b1\,b2 + c1\,c2)/q$.

Usage

exactQuotient(x, y)

Signature

exactQuotient: (%, %) → `Partial` %

Parameter	Type	Description
x	%	The numerator
y	%	The denominator

Returns

Returns the unique q such that $x = q\,y$ if such a q exists, *failed* otherwise.

See also

`quotient`

Usage

order a
order(a)(b)

Signature

order: $\% \to \% \to$ `Integer`

Parameter	Type	Description
a	$\%$	A nonunit element of the domain
b	$\%$	A nonzero element of the domain

Returns

order(a)(b) returns the unique nonnegative integer $n = \nu_a(b)$ such that $a^n \mid b$ and $a^{n+1} \not\mid b$, while order(a) returns the map $b \to \nu_a(b)$.

Remarks

order(a) makes some precalculations based on a, so if you need to use order(a)(b) several times with the same a and different b's, it is more efficient to compute order(a) once and assign it, as in the example below. In addition, if you need to compute $b/a^{\nu_a(b)}$, then it is more efficient to use the `orderquo` function.

Example

The following function computes the orders at $x^2 + 1$ of a list of polynomials $l := [p_1, \ldots, p_n]$:

```
orders(l:List DenseUnivariatePolynomial Integer):List Integer == {
    import from DenseUnivariatePolynomial Integer;
    nu := order(monom * monom + 1);    -- order function at x^2 + 1
    [nu(p) for p in l];
}
```

See also

`orderquo`

Usage

orderquo a
(n, c) := orderquo(a)(b);

Signature

orderquo: % → % → (`Integer`, %)

Parameter	Type	Description
a	%	A nonunit element of the domain
b	%	A nonzero element of the domain

Returns

orderquo(a)(b) returns (n, c) such that $b = ca^n$ and $a \not| \ c$, while orderquo(a) returns the map $b \to$ orderquo(a)(b).

Remarks

orderquo(a) makes some precalculations based on a, so if you need to use orderquo(a)(b) several times with the same a and different b's, it is more efficient to compute orderquo(a) once and assign it.

See also

`order`

Usage

quotient(x, y)
quotient!(x, y)
quotientBy(y)
quotientBy(y)(x)
quotientBy!(y)
quotientBy!(y)(x)

Signatures

quotient,quotient!: (%, %) → %
quotientBy, quotientBy!: % → % → %

Parameter	Type	Description
x	%	The numerator
y	%	The denominator

Returns

quotient(x,y) and quotient!(x,y) return the unique q such that $x = q\,y$ if such a q exists, while quotientBy(y) returns the map $x \to \text{quotient}(x, y)$ and quotientBy!(y) returns the map $x \to \text{quotient!}(x, y)$. When using quotient! or quotientBy!, the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

Use those functions only when it is guaranteed that y divides x exactly in the ring. If there is a nonzero remainder, this function may produce a wrong quotient instead of an error, so use `exactQuotient` when there is the possibility of a nonzero remainder. When however y is known to divide x exactly, then `quotient` can have better efficiency than the other divisions.

See also

`exactQuotient`

Usage

LinearArithmeticType R:Category

Parameter	Type	Description
R	`ExpressionType` `AdditiveType`	The coefficient domain

Description

LinearArithmeticType R is the category of arithmetic types containing linear combinations of their elements with coefficients in R.

Remarks

Use `Algebra` instead if R is always meant to be a `Ring`.

Exports

`ArithmeticType`
`ExpressionType`
`LinearCombinationType` R

^:	(%, `Integer`) → %	exponentiation
`coerce`:	R → %	Natural embedding

Usage

$x \hat{} n$

Signature

**: (%,`Integer`) → %

Parameter	Type	Description
x	%	an element of the type
n	`Integer`	an exponent

Returns

Returns x^n.

Usage

coerce r

Signature

coerce: R → %

Parameter	Type	Description
r	R	An element of the base type

Returns

Returns $r \cdot 1$.

Module

Usage

Module R:Category

Parameter	Type	Description
R	`Ring`	The coefficient ring

Description

Module R is the category of modules over R.

Exports

`AbelianGroup`
`ExpressionType`
`LinearCombinationType` R

Monoid

Usage

Monoid: Category

Description

Monoid is the category of monoids, not necessarily commutative.

Exports

`ExpressionType`

1:	%	one
$*$:	(%, %) → %	product
$\hat{}$:	(%, `Integer`) → %	exponentiation
`one?`:	% → `Boolean`	test for 1
`times!`:	(%, %) → %	In–place product

Usage

1

Signature

1: %

Returns

Returns the constant 1.

Usage

$x * y$

Signature

$*$: (%,%) $\to$ %

Parameter	Type	Description
x,y	%	elements of the monoid

Returns

Returns the product xy.

Usage

$x \hat{} n$

Signature

**: (%,`Integer`) → %

Parameter	Type	Description
x	%	an element of the monoid
n	`Integer`	an exponent

Returns

Returns x^n.

Usage

one? x

Signature

one?: % → `Boolean`

Parameter	Type	Description
x	%	An element of the monoid

Returns

Returns *true* if $x = 1$, *false* otherwise.

Usage

times!(x, y)

Signature

times!: (%, %) → %

Parameter	Type	Description
x	%	An element of the monoid (to be destroyed)
y	%	An element of the monoid to be multiplied by x

Returns

Returns the product xy, where the storage used by x is allowed to be destroyed or reused, so x can be lost after this call.

Remarks

This function may cause x to be destroyed, so do not use it unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

NonCommutativeIntegralDomain: Category

Description

NonCommutativeIntegralDomain is the category of non–commutative integral domains.

Exports

`Ring`

`leftExactQuotient`:	(%, %) → `Partial` %	Left exact quotient
`rightExactQuotient`:	(%, %) → `Partial` %	Right exact quotient

Usage

leftExactQuotient(x, y)

Signature

leftExactQuotient: (%, %) → `Partial` %

Parameter	Type	Description
x	%	The numerator
y	%	The denominator

Returns

Returns either q such that $x = y\,q$ if such a q exists, *failed* otherwise.

See also

rightExactQuotient(NonCommutativeIntegralDomain)

Usage

rightExactQuotient(x, y)

Signature

rightExactQuotient: (%, %) → `Partial` %

Parameter	Type	Description
x	%	The numerator
y	%	The denominator

Returns

Returns either q such that $x = q\,y$ if such a q exists, *failed* otherwise.

See also

leftExactQuotient(NonCommutativeIntegralDomain)

Usage

Ring: Category

Description

Ring is the category of rings, not necessarily commutative.

Exports

`AbelianGroup`
`ArithmeticType`
`Monoid`

`characteristic`:	`Integer`	characteristic
`coerce`:	`Integer` → %	embedding of the integers
`factorial`:	(%, %, `MachineInteger`) → %	Generalized factorial
`random`:	() → %	Get a random element

Usage
characteristic

Signature
characteristic: `Integer`

Returns
Returns the characteristic of the ring.

Usage

coerce n
n::%

Signature

coerce: `Integer` → %

Parameter	Type	Description
n	`Integer`	an integer

Returns

Returns *n* seen as an element of the ring.

Usage

factorial(a, s, n)

Signature

factorial: (%, %, `MachineInteger`) → %

Parameter	Type	Description
a, s	%	Elements of the ring
n	`MachineInteger`	A nonnegative integer

Returns

Returns the generalized factorial $\prod_{i=0}^{n-1}(a + is)$.

Usage
random()

Signature
random: () → %

Returns
Returns a random element.

Usage

RittRing: Category

Description

RittRing is the category of rings of characteristic *0* in which all nonzero integers are invertible. The center of such a ring contains an isomorphic image of the rational numbers.

Exports

`CharacteristicZero`

/:	(%, `Integer`) → %	Division by a nonzero integer
`inv`:	`Integer` → %	Inversion of a nonzero integer

Usage

x /n

Signature

/: (%,`Integer`) → %

Parameter	Type	Description
x	%	An element of the ring
n	`Integer`	A nonzero integer

Returns

Returns x/n as an element of the ring.

Usage

inv n

Signature

inv: `Integer` → %

Parameter	Type	Description
n	`Integer`	A nonzero integer

Returns

Returns $1/n$ as an element of the ring.

Specializable

Usage

Specializable: Category

Description

Specializable is the category of specializable types.

Exports

`specialization`: (Image:`CommutativeRing`) → `PartialFunction`(%, Image) Morphism

Usage

specialization R

Signature

specialization: (Image:`CommutativeRing`) → `PartialFunction`(%,Image)

Parameter	Type	Description
R	`CommutativeRing`	A ring

Returns

Returns a partial map from % into R.

Usage

ExpressionType: Category

Description

ExpressionType is the category of types whose elements can be converted to `ExpressionTree`.

Exports

`OutputType`
`PrimitiveType`

`extree:`	% → `ExpressionTree`	Conversion to an expression tree
`relativeSize:`	% → `MachineInteger`	Complexity measure

Usage

extree x

Signature

extree: % → `ExpressionTree`

Parameter	Type	Description
x	%	The element to convert

Description

Converts x to an expression tree.

Usage
relativeSize x

Signature
relativeSize: % → `MachineInteger`

Parameter	Type	Description
x	%	An element of the type

Returns
Returns some measure of the complexity of x.

Remarks
This measure does not have to be absolute, but it should be usable to compare 2 elements of the type and decide which one is the "cheapest" for calculations. This measure has no mathematical meaning in general, but can be used for selection strategies, for example in Gaussian elimination.

Automorphism

Usage

import from Automorphism R

Parameter	Type	Description
R	`Ring`	The ring on which the automorphisms operate

Description

Automorphism R provides automorphims on R.

Exports

`Group`

`apply`:	(%, R, `Integer`) → R	Apply a morphism to an element of R
`function`:	% → (R → R)	Action of a morphism
`morphism`:	(R → R) → %	Create a morphism
	(R → R, R → R) → %	
	((R, `Integer`) → R) → %	

Usage

σx
$\sigma(x, n)$
apply(σ, x)
apply(σ, x, n)

Signature

apply: (%, R, .:`Integer` == 1) → R

Parameter	Type	Description
σ	%	An automorphism of R
x	R	An element of R
n	`Integer`	The number of times to apply (optional)

Returns

Returns $\sigma^n x$.

Usage

function σ

Signature

function: $\% \to (\mathrm{R} \to \mathrm{R})$

Parameter	Type	Description
σ	%	An automorphism of R

Returns

Returns the map corresponding to the action of σ on the ring.

Usage

morphism f
morphism(f, f^{-1})
morphism g

Signatures

morphism: (R → R) → %
morphism: (R → R, R → R) → %
morphism: ((R, `Integer`) → R) → %

Parameter	Type	Description
f	R → R	A function
f^{-1}	R → R	The inverse function of f
g	(R, `Integer`) → R	A function

Description

morphism f creates the morphism σ on R given by

$$\sigma x = f(x)$$

for any $x \in R$. The morphism is not necessarily invertible, so any attempt to use its inverse causes an error.
morphism(f, f^{-1}) creates the invertible morphism σ on R given by

$$\sigma x = f(x) \qquad \sigma^{-1} x = f^{-1}(x)$$

for any $x \in R$.
morphism g creates the morphism σ on R given by

$$\sigma^n x = g(x, n)$$

for any $x \in R$. This morphism is considered invertible, so g must also be defined for negative integers.

Remarks

The maps passed as arguments must be ring morphisms, and the maps f and f^{-1} must be inverses of each other. When an efficient algorithm for computing σ^n is known, for example for $\sigma = 1_R$, then the form morphism g with g: (R, Integer) → R should be used to avoid repeated iterations of σ, which is the default behavior.

Usage

import from ChineseRemainderingR

Parameter	Type	Description
R	`EuclideanDomain`	an Euclidean Domain.

Description

ChineseRemaindering provides Garner's Chinese Remaindering Algorithm implemented over an arbitrary `EuclideanDomain`.

Exports

`combine`:	(R,R) → (R,R) → R	combine interpolated result with new modulus.
`interpolate`:	(`List` R,`List` R) → R	interpolate given residues and moduli.

if R has `IntegerCategory` then

`combine`:	(R,`MachineInteger`) → (R,`MachineInteger`) → R	combine with new modulus.

Usage

combine(M,m)
combine(M,m)(A,a)

Signatures

combine: (R,R) → (R,R) → R
combine: (R,`MachineInteger`) → (R,`MachineInteger`) → R

Parameter	Type	Description
M	R	A product of primes.
m	R `MachineInteger`	A new modulus.
A	R	The interpolated value modulo M.
a	R `MachineInteger`	The residue modulo m.

Returns

Returns the unique X in $R/(mM)$ such that $X = A \pmod{M}$ and $X = a \pmod{m}$.

Usage

interpolate(p,m)

Signature

interpolate: (`List` R,`List` R) → R

Parameter	Type	Description
p	`List` R	A list of residues.
m	`List` R	A list of moduli.

Returns

Returns the interpolated value from the residues and the corresponding moduli.

Usage

import from Complex R

Parameter	Type	Description
R	`ArithmeticType`	Type to be extended
	`ExpressionType`	

Description

Complex R implements the algebraic extension of R generated by a root of $X^2 + 1 = 0$. This type, already provided by `libaldor`, is extended by `Algebra`. Only the additional exports are documented here, see the `libaldor` reference manual for the basic exports and assumptions on the parameter R.

Exports

`LinearArithmeticType` R

if R has `CharacteristicZero` then
`CharacteristicZero`

if R has `CommutativeRing` then
`CommutativeRing`

if R has `IntegralDomain` then
`IntegralDomain`

if R has `Field` then
`Field`

if R has `FiniteCharacteristic` then
`FiniteCharacteristic`

if R has `FiniteField` then
`FiniteField`

if R has `Parsable` then
`Parsable`

if R has `Ring` then
`Algebra` R

if R has `RittRing` then
`RittRing`

Usage

import from Derivation R

Parameter	Type	Description
R	`Ring`	The ring on which the derivations operate

Description

Derivation R provides derivations on R.

Exports

`Module` R

`apply`:	(%, R, `Integer`) → R	Differentiate an element of R
`derivation`:	(R → R) → %	Create a derivation
`function`:	% → (R → R)	Action of a derivation

Usage

apply(D, x)
apply(D, x, n)
D x
D(x,n)

Signature

apply: (%, R, .:`Integer` == 1) → R

Parameter	Type	Description
D	%	A derivation
x	R	An element to differentiate
n	`Integer`	The number of times to differentiate (optional)

Returns

Returns $D^n x$, *i.e.* the result of applying D to x n times.

Usage

derivation f

Signature

derivation: $(\mathrm{R} \to \mathrm{R}) \to \%$

Parameter	Type	Description
f	$\mathrm{R} \to \mathrm{R}$	A map

Returns

Returns the derivation D on R given by

$$Dx = f(x)$$

for any $x \in R$.

Remarks

f must satisfy the rules of a derivation, namely:

$$f(x+y) = f(x) + f(y), \quad \text{and} \quad f(xy) = xf(y) + f(x)y$$

for any $x, y \in R$.

Usage

function D

Signature

function: $\% \to (\mathrm{R} \to \mathrm{R})$

Parameter	Type	Description
D	$\%$	A derivation

Returns

Returns the map corresponding to the action of D on the ring.

Usage

import from Fraction R

Parameter	Type	Description
R	`GcdDomain`	a gcd domain

Description

Fraction R forms the fraction field of the gcd domain R. Fractions are automatically normalized in this field.

Exports

`FractionFieldCategory` R

Usage

import from FractionalRoot

Description

FractionalRoot(R) provides fractions of R with multiplicities.

Parameter	Type	Description
R	`CommutativeRing`	A ring

Exports

ExpressionType

`fractionalRoot`:	(R, R, `Integer`) → %	Create a root
`integral?`:	% → `Boolean`	Test whether root is integral
`integralRoot`:	(R, `Integer`) → %	Create a root
`integralValue`:	% → R	Value of an integral root
`multiplicity`:	% → `Integer`	Multiplicity of a root
`setMultiplicity!`:	(%, `Integer`) → %	Change a multiplicity
`value`:	% → (R, R)	Value of a root

Signature

integral?: % → `Boolean`

Usage

integral? r

Parameter	Type	Description
r	%	A root

Returns

Return *true* if r is in R, *false* otherwise.

Usage

fractionalRoot(a, b, n)
integralRoot(a, n)

Signatures

fractionalRoot: (R, R, `Integer`) → %
integralRoot: (R, `Integer`) → %

Parameter	Type	Description
a	R	A numerator
b	R	A denominator
n	`Integer`	A multiplicity

Returns

Return the root a or a/b with multiplicity n.

Usage

integralValue r

Signature

integralValue: % → `Integer`

Parameter	Type	Description
r	%	A root

Returns

Returns the value of the integral root r, ignoring its multiplicity.

See also

`value`

Usage

multiplicity r

Signature

multiplicity: $\% \to$ `Integer`

Parameter	Type	Description
r	$\%$	A root

Returns

Return the multiplicity of r.

Usage

setMultiplicity!(r, m)

Signature

setMultiplicity!: (%, `Integer`) → `Integer`

Parameter	Type	Description
r	%	A root
m	`Integer`	Its new multiplicity

Description

Sets the multiplicity of r to m and returns r.

Usage

(n, d) := value r

Signature

value: % → (`Integer`, `Integer`)

Parameter	Type	Description
r	%	A root

Returns

Return (n, d) such that the value of r is n/d.

See also

`integralValue`

Usage

import from FractionBy(R, p, irr?)

Parameter	Type	Description
R	`IntegralDomain`	an integral domain
p	R	a nonzero nonunit of R
irr?	`Boolean`	Indicates whether p is known to be irreducible in R

Description

FractionBy(R, p, irr?) forms the fractions of the integral domain *R* by the nonzero nonunit *p*, *i.e.*the set of all fractions whose denominator is a power of *p*. Fractions are normalized in the sense that *p* does not divide the numerators. Indicating whether *p* is irreducible if for efficiency purposes only, always use *false* when it is unknown.

Exports

`FractionByCategory` R

Usage

FractionByCategory R: Category

Parameter	Type	Description
R	`IntegralDomain`	an integral domain

Description

FractionByCategory(R) is the category of fractions of the integral domain R by some nonzero nonunit $p \in R$, *i.e.*the set of all fractions whose denominator is a power of p.

Exports

`FractionByCategory0` R

Usage

FractionByCategory0 R: Category

Parameter	Type	Description
R	`IntegralDomain`	an integral domain

Description

FractionByCategory0(R) is the category of fractions of the integral domain R by some nonzero nonunit $p \in R$, *i.e.*the set of all fractions whose denominator is a power of p.

Exports

`FractionCategory` R

`order`:	% → `Integer`	Valuation at p
`shift`:	(%, `Integer`) → %	Multiplication by a power of p

Usage

order x

Signature

order: $\% \to$ `Integer`

Parameter	Type	Description
x	$\%$	A fraction whose denominator is a power of p

Returns

Returns n such that $x = ap^n$ and $a \in R$, $p \not| \ a$.

Usage

shift(x, n)

Signature

shift: (%, `Integer`) → %

Parameter	Type	Description
x	%	A fraction whose denominator is a power of p
n	`Integer`	An exponent

Returns

Returns xp^n.

Usage

FractionCategory R: Category

Parameter	Type	Description
R	`IntegralDomain`	an integral domain

Description

FractionCategory R is the category of subrings of the fraction field of R.

Exports

`Algebra` R
`DifferentialExtension` R
`IntegralDomain`
`LinearAlgebraRing`

`denominator`:	$\% \to$ R	Denominator of a fraction
`normalize`:	$\% \to \%$	Normalize a fraction
`numerator`:	$\% \to$ R	Numerator of a fraction

if R has `CharacteristicZero` then
`CharacteristicZero`

if R has `FactorizationRing` then
`FactorizationRing`

if R has `FiniteCharacteristic` then
`FiniteCharacteristic`

if R has `RationalRootRing` then
`RationalRootRing`

if R has `Specializable` then
`Specializable`

if R has `UnivariateGcdRing` then
`UnivariateGcdRing`

Usage

denominator x
numerator x

Signature

denominator,numerator: $\% \to \mathrm{R}$

Parameter	Type	Description
x	$\%$	A fraction

Returns

Returns respectively the denominator and the numerator of a fraction.

Usage

normalize x

Signature

normalize: $\% \to \%$

Parameter	Type	Description
x	$\%$	A fraction

Description

Normalize x as much as possible given the category of R.

Usage

FractionFieldCategory R: Category

Parameter	Type	Description
R	`IntegralDomain`	an integral domain

Description

FractionFieldCategory R is the category of the fraction fields of R.

Exports

`FractionFieldCategory0` R

Usage

FractionFieldCategory0 R: Category

Parameter	Type	Description
R	`IntegralDomain`	an integral domain

Description

FractionFieldCategory0 R is the category of the fraction fields of R.

Exports

`Field`
`FractionCategory` R
`/`: (R,R) → % Quotient of two ring elements

if R has `CharacteristicZero` then
`RittRing`

if R has `Parsable` then
`Parsable`

if R has `SerializableType` then
`SerializableType`

if R has `TotallyOrderedType` then
`TotallyOrderedType`

Usage

n / d

Signature

/: (R,R) → %

Parameter	Type	Description
n	R	An element of the ring.
d	R	A nonzero element of the ring.

Returns

Returns the quotient *n* over *d*.

Usage

LinearCombinationFraction(R, LR, Q, LQ): Category

Parameter	Type	Description
R	`IntegralDomain`	an integral domain
LR	`LinearCombinationType` R	a type over R
Q	`FractionCategory` R	a fraction domain of R
LQ	`LinearCombinationType` Q	a type over Q

Description

LinearCombinationFraction(R, LR, Q, LQ) is a category for domains providing conversions between types integral and rational coefficients.

Exports

$*$:	(Q, LR) → LQ	Product of a fraction by an integral element
`makeIntegral`:	LQ → (R, LR)	Conversion to an integral element
`makeRational`:	LR → LQ	Conversion to a rational element

if Q has `FractionByCategory0` R

`makeIntegralBy`:	LQ → (`Integer`, LR)	Conversion to an integral element

if Q has `FractionFieldCategory0` R

`normalize`:	LR → (R, LQ)	Conversion to a monic rational element

Usage

$c * A$

Signature

$*$: (Q, LR) $\rightarrow$ LQ

Parameter	Type	Description
c	Q	A fraction
A	LR	An integral element

Returns

Returns cA as a rational element.

Usage

(a, A) := makeIntegral B

Signature

makeIntegral: LQ → (R, LR)

Parameter	Type	Description
B	LQ	A rational element

Returns

Returns (a, A) such that $A = aB$ is integral. If R is a `GcdDomain`, then A is primitive.

Usage

(μ, A) := makeIntegralBy B

Signature

makeIntegralBy: LQ $\to$ (`Integer`, LR)

Parameter	Type	Description
B	LQ	A rational element

Returns

Returns (μ, A) such that $A = \text{shift}(B, \mu)$ is integral.

Usage

makeRational A

Signature

makeRational: LR → LQ

Parameter	Type	Description
A	LR	An integral element

Returns

Returns *A* as a rational element.

Usage

(a, B) := normalize A

Signature

normalize: LR → (R, LQ)

Parameter	Type	Description
A	LR	An integral element

Returns

Returns (a, B) such that $A = aB$, and B is a normalized rational element.

Remarks

The normalization depends on the actual type LQ. For polynomial, it means that the leading coefficient if 1, for vectors that the first coordinate is 1, etc.

Product

Usage

import from Product R

Parameter	Type	Description
R	`CommutativeRing`	A commutative ring

Description

Product R provides finite products of elements of R, *i.e.*elements of the type $\prod_{i=1}^{n} r_i^{e_i}$ where the r_i's are in R and the e_i's are integers.

Exports

`CopyableType`

`Monoid`

`#`:	% → `MachineInteger`	Number of terms
`divisors`:	% → `Generator` R	Iterate through all the divisors
`expand`:	% → R	Multiply-out a product
`expandFraction`:	% → (R, R)	Multiply-out a product
`generator`:	% → `Generator Cross`(R, `Integer`)	Make an iterator
`log`:	(M:`AbelianMonoid`, R → M) → (% → M)	Lift a logarithm
`term`:	(R, `Integer`) → %	Create a single term r^e
`times!`:	(%, R, `Integer`) → %	In-place multiplication

Usage

p

Signatures

#: % → `MachineInteger`

Parameter	Type	Description
p	%	A product

Returns

Returns the number of terms in the product p.

Usage

for d in divisors p repeat { ... }

Signature

divisors: % → `Generator` R

Parameter	Type	Description
p	%	A product

Description

This generator yields all the products of the form $\prod_{i=1}^{n} r_i^{f_i}$ where $p = \prod_{i=1}^{n} r_i^{e_i}$ and $0 \leq f_i \leq e_i$.

Example

```
import from Integer, Product Integer, List Integer;

p := term(3, 1) * term(2, 2) * term(5, 2)       -- p = 3^1 2^2 5^2 = 300
l := sort! [divisors p];
```

creates the list

[1,2,3,4,5,6,10,12,15,20,25,30,50,60,75,100,150,300]

of all the divisors of 300.

Usage

expand p

Signature

expand: % → R

Parameter	Type	Description
p	%	A product

Returns

Returns the product of all the terms in p.

Remarks

When R is not a field, expand(p) only works when p has only nonnegative exponents. Use `expandFraction` when p can have negative exponents and R is not a field.

Usage

expandFraction p

Signature

expandFraction: % → (R, R)

Parameter	Type	Description
p	%	A product

Returns

Returns (n, d) where n is the product of all the terms in p having positive exponents and d is the product of all terms in p having negative exponents.

See also

`expand`

Usage

for term in p repeat { (c, n) := term; ... }
for term in generator p repeat { (c, n) := term; ... }

Signature

generator: % → `Generator Cross`(R, `Integer`)

Parameter	Type	Description
p	%	A product

Description

This function allows a product to be iterated independently of its representation. The generator yields pairs of the form (a, n) where a^n is a term in p.

Example

```
import from Integer, Product Integer;

p := term(3, 1) * term(2, 11) * term(5, 2)       -- p = 3^1 2^11 5^2
for term in p repeat { (c, n) := term; stdout << c << "," << n << newline; }
```

writes

3,1
2,11
5,2

to the standard stream `stdout`.

Usage

log(M,f)
log(M,f)(p)

Signature

log: (M:`AbelianMonoid`, R → M) → % → M

Parameter	Type	Description
M	`AbelianMonoid`	the image monoid
f	R → M	a logarithmic function on R
p	%	A product

Description

log(M,f)(p) returns $\sum_n nf(a_n)$ where $p = \prod_n a_n^n$, while log(M,f) returns the map $\prod_n a_n^n \to \sum_n nf(a_n)$.

Usage

term(r, n)

Signature

term: (R, `Integer`) → %

Parameter	Type	Description
r	R	A ring element
n	`Integer`	An exponent

Returns

Returns r^e as a product.

Usage

times!(p, r, n)

Signature

times!: (%, R, `Integer`) → %

Parameter	Type	Description
p	%	A product
r	R	A ring element
n	`Integer`	An exponent

Returns

Returns $p\, r^e$ as a product, where the storage used by p is allowed to be destroyed or reused, so p is lost after this call.

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other polynomials. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

throw ReducibleModulusException(R, m, a)
try . . . catch E in { E has `ReducibleModulusExceptionType` R =¿ . . . }

Parameter	Type	Description
R	`CommutativeRing`	A commutative ring
m	R	The modulus
a	R	A proper factor of m

Description

ReducibleModulusException(R, m, a) is an exception type thrown by inversion modulo a reducible element of R.

Usage

ReducibleModulusExceptionType R: Category

Parameter	Type	Description
R	`CommutativeRing`	A commutative ring

Description

ReducibleModulusExceptionType R is the category of exceptions thrown by inversion modulo a reducible element of R. The constants `modulus` and `factor` contain the modulus and a proper factor respectively.

Usage

import from SimpleAlgebraicExtension(R, Rx, p)
import from SimpleAlgebraicExtension(R, Rx, p, x)

Parameter	Type	Description
R	`IntegralDomain`	The coefficient ring of the polynomials
Rx	`UnivariatePolynomialCategory` R	The type of the modulus
p	Rx	The modulus
x	`Symbol`	The generator name (optional)

Description

SimpleAlgebraicExtension(R, Rx, p) implements the algebraic extension $Rx/(p)$, where p is expected to be irreducible. It is possible to use this type with reducible p, but then divisions can throw the exception `ReducibleModulusException`. Use `UnivariatePolynomialMod` if you want to prevent divisions from being available.

Exports

`SimpleAlgebraicExtensionCategory`(R, Rx)

Usage

SimpleAlgebraicExtensionCategory(R, Rx):Category

Parameter	Type	Description
R	`IntegralDomain`	The coefficient ring of the polynomials
Rx	`UnivariatePolynomialCategory` R	The type of the modulus

Description

SimpleAlgebraicExtensionCategory(R, Rx) is the category of extensions of R of the form $Rx/(p)$ for some irreducible $p \in Rx$. Use `UnivariatePolynomialQuotient` for extensions where the modulus is known to be reducible.

Remarks

It is possible to use this category with a reducible modulus, but then divisions can throw the exception `ReducibleModulusException`.

Exports

`IntegralDomain`
`UnivariatePolynomialQuotient`(R, Rx)

if R has `Field` then
`Field`

if R has `Field` and R has `CharacteristicZero` and R has `FactorizationRing` then
FactorizationRing

if R has `FiniteField` then
`FiniteField`

Usage

import from UnivariatePolynomialMod(R, Rx, p)
import from UnivariatePolynomialMod(R, Rx, p, x)

Parameter	Type	Description
R	`CommutativeRing`	The coefficient ring of the polynomials
Rx	`UnivariatePolynomialCategory` R	The type of the modulus
p	Rx	The modulus
x	`Symbol`	The generator name (optional)

Description

UnivariatePolynomialMod(R, Rx, p) implements the univariate polynomials modulo p, *i.e.*the ring $Rx/(p)$, where p is not necessarily irreducible. However, the leading coefficient of p must be a unit in R. Use `SimpleAlgebraicExtension` for extensions where the modulus is known to be irreducible.

Exports

`UnivariatePolynomialQuotient`(R, Rx)

Usage

UnivariatePolynomialQuotient(R, Rx):Category

Parameter	Type	Description
R	`CommutativeRing`	The coefficient ring of the polynomials
Rx	`UnivariatePolynomialCategory` R	The type of the modulus

Description

UnivariatePolynomialQuotient(R, Rx) is the category of extensions of R of the form $Rx/(p)$ for some $p \in Rx$, not necessarily irreducible. Use `SimpleAlgebraicExtensionCategory` for extensions where the modulus is known to be irreducible.

Exports

`Algebra` R
`CommutativeRing`

`compose`:	% → % → %	Modular composition
`definingPolynomial`:	→ Rx	Defining polynomial
`lift`:	% → Rx	Conversion to a polynomial
`monom`:	→ %	Generator of the algebra
`norm`:	% → R	Norm
	(P:POLY %) → P → Rx	
`reduce`:	Rx → %	Reduction of a polynomial
`trace`:	% → R	Trace
	(P:POLY %) → P → Rx	
`value`:	(P:POLY %) → (P, Z, Z) → %	Evaluation at a rational number

where

Z == `Integer`
POLY == `UnivariatePolynomialCategory`

if R has `FiniteCharacteristic` then
`FiniteCharacteristic`

if R has `FiniteSet` then
`FiniteSet`

if R has `RationalRootRing` then
`RationalRootRing`

Usage

compose(p)(q)

Parameter	Type	Description
p, q	%	Polynomials

Returns

Returns

$$q(p) = \sum_{i=0}^{n} a_i p^i$$

where $q = \sum_{i=0}^{n} a_i x^i$.

Remarks

If you want to compute $q_1(p), \ldots, q_k(p)$ for several q_i's, use the curried version as follows: `f := compose p; for i in 1..k repeat r.i := f(q.i);` , since the various calls to `f` will share a table of powers of p.

Usage
definingPolynomial

Signature
definingPolynomial: Rx

Returns
Returns the polynomial p such that the extension is $Rx/(p)$.

Usage

lift q

Signature

lift: % → Rx

Parameter	Type	Description
q	%	An element of the algebraic extension

Returns

Returns *q* as an element of *Rx*.

Usage
monom

Signature
monom: %

Returns
Returns the image in the quotient of the term x from Rx. That element generates this type as an algebra over R.

See also
`monom`

Usage

norm q
trace q
norm P
trace P
norm(P)(p)
trace(P)(p)

Signatures

norm,trace: % → R
norm,trace: (P:`UnivariatePolynomialCategory` %) → P → Rx

Parameter	Type	Description
q	%	An element of the algebraic extension
P	`UnivariatePolynomialCategory` %	A polynomial type
p	P	A polynomial

Description

norm($q(\alpha)$) and trace($q(\alpha)$) return respectively the product and sum of the $q(\alpha)$ over all the roots of the polynomial defining the extension, while norm(P)($p(\alpha, x)$) and trace(P)($p(\alpha, x)$) return respectively the product and sum of the $p(\alpha, x)$ over all the roots of the polynomial defining the extension.

Usage

reduce q

Signature

reduce: Rx → %

Parameter	Type	Description
q	Rx	A polynomial

Returns

Returns the remainder of q modulo p as an element of $Rx/(p)$.

Usage

value(P)(p, n, d)

Signature

value: (P:`UnivariatePolynomialCategory` %) → (P, `Integer`, `Integer`) → %

Parameter	Type	Description
P	`UnivariatePolynomialCategory` %	A polynomial type
p	P	A polynomial
n	`Integer`	A numerator
d	`Integer`	A nonzero denominator

Returns

Returns $d^e p(n/d)$ where e is the smallest nonnegative exponent such that $d^e p(n/d)$ is an element of the extension.

Usage

import from SmallPrimes

Description

SmallPrimes implements functionalities to obtain and manipulate small odd primes. Only a specific set $\mathcal{P}$ of small primes is available.

Exports

`PrimeCollection`

Usage

import from LazyHalfWordSizePrimes

Description

LazyHalfWordSizePrimes implements functionalities to obtain and manipulate half-word-size odd primes for lazy algorithms. Those primes are a few bits less that half-word-size, which allows for accumulation before reduction. Only a specific set $\mathcal{P}$ of half-word-size primes is available.

Exports

`PrimeCollection`

Usage

import from HalfWordSizePrimes

Description

HalfWordSizePrimes implements functionalities to obtain and manipulate half-word-size odd primes. Only a specific set $\mathcal{P}$ of half-word-size primes is available.

Exports

`PrimeCollection`

Usage

import from WordSizePrimes

Description

WordSizePrimes implements functionalities to obtain and manipulate word-size primes. Only a specific set $\mathcal{P}$ of word-size primes is available.

Exports

`PrimeCollection`

Usage

PrimeCollection: Category

Description

PrimeCollection is the category of collections of primes with various properties and various sizes.

Exports

`allPrimes`:	() → `Generator` Z	Generate all the primes
`fourierPrime`:	Z → (Z, Z)	Fourier prime
`maxPrime`:	→ Z	Largest prime
`nextPrime`:	Z → Z	First prime above a given number
`previousPrime`:	Z → Z	First prime below a given number
`primRoot`:	Z → Z	Modular primitive root
`randomPrime`:	() → Z	Random prime

where

Z == `MachineInteger`

Usage

for p in allPrimes() repeat { ... }

Signature

allPrimes: () → `Generator MachineInteger`

Description

This function allows a loop to iterate over all the primes provided by the collection.

Usage

fourierPrime n

Signature

fourierPrime: `MachineInteger` → (`MachineInteger`,`MachineInteger`)

Returns

Returns (p, ω) such that p is a prime of the form $p = 2^n k + 1$ with k odd, and ω is a primitive $2^{n\text{th}}$ root of unity in $\mathbb{F}_p$. Returns $(0, 0)$ if n is too large.

Usage
maxPrime

Signature
maxPrime: `MachineInteger`

Returns
Returns the largest prime in the collection.

Usage

nextPrime n

Signature

nextPrime: `MachineInteger` → `MachineInteger`

Parameter	Type	Description
n	`MachineInteger`	An integer

Returns

Returns the smallest prime p in the collection with $n < p$, 0 if there are none.

See also

previousPrime(PrimeCollection), randomPrime(PrimeCollection)

Usage

previousPrime n

Signature

previousPrime: `MachineInteger` → `MachineInteger`

Parameter	Type	Description
n	`MachineInteger`	An integer

Returns

Returns the largest prime p in the collection with $n > p$, 0 if there are none.

See also

nextPrime(PrimeCollection), randomPrime(PrimeCollection)

Usage
primRoot p

Signature
primRoot: `MachineInteger` → `MachineInteger`

Parameter	Type	Description
p	`MachineInteger`	A prime

Returns
Returns a generator of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$ or 0 if p is not a prime in the table, or is no primitive root is stored.

See also
primitiveRoot(`PrimitiveRoots`)

Usage
randomPrime()

Signature
randomPrime: () → `MachineInteger`

Returns
Returns a random prime in the collection.

See also
nextPrime(PrimeCollection), previousPrime(PrimeCollection)

PrimeField2

Usage
import from PrimeField2

Description
PrimeField2implements the finite field with two elements.

Exports
`SmallPrimeFieldCategory`

Usage

PrimeFieldCategory: Category

Description

PrimeFieldCategory is the category for prime fields,*i.e.*fields of the form $\mathbb{Z}/p\mathbb{Z}$ where $p \in \mathbb{Z}$ is a prime.

Exports

`PrimeFieldCategory0`
`FactorizationRing`

`roots`:	(P:POLY %) → P → `List Cross`(%, `Integer`)	In–field roots
`rootsSqfr`:	(P:POLY %) → P → `List` %	In–field roots

where

POLY == `UnivariatePolynomialCategory0`

Usage

rootsSqfr P
rootsSqfr(P)(p)

Signature

rootsSqfr: (P:`UnivariatePolynomialCategory0` %) → P → `Generator` %

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	a polynomial type
p	P	a squarefree polynomial

Returns

Returns a generator that produces all the roots of *p*, which must be squarefree, in its coefficient field.

Usage

PrimeFieldCategory0: Category

Description

PrimeFieldCategory0 is the category for prime fields,*i.e.*fields of the form $\mathbb{Z}/p\mathbb{Z}$ where $p \in \mathbb{Z}$ is a prime.

Exports

`FiniteCharacteristic`
`FiniteField`
`SerializableType`
`lift`: $\% \to$ `Integer` Conversion to an integer

Usage

lift x

Signature

lift: % → `Integer`

Parameter	Type	Description
x	%	an element of the field

Returns

Return x seen as an integer.

Usage

import from PrimitiveRoots

Description

PrimitiveRoots implements functionalities needed to compute generators of small cyclic groups.

Exports

`factors:`	Z → `List` Z	List of prime factors
`primitiveRoot:`	Z → Z	Modular primitive root

where

Z == `MachineInteger`

Usage

factors n

Signature

factors: `MachineInteger` → `List MachineInteger`

Parameter	Type	Description
n	`MachineInteger`	An integer

Returns

Returns all the prime factors p of n with $1 < d < |n|$.

Usage

primitiveRoot p

Signature

primitiveRoot: `MachineInteger` → `MachineInteger`

Parameter	Type	Description
p	`MachineInteger`	A prime

Returns

Returns a generator of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$.

Remarks

The argument p must be a prime number, otherwise this function returns any integer with no significance.

PthPowering

Usage

import from PthPowering R

Parameter	Type	Description
R	`FiniteCharacteristic`	A finite characteristic ring

Description

PthPowering provides efficient exponentiation of elements of R.

Exports

`pExponentiation`:	(T, `Integer`) → T	p^{th}–powering
`pExponentiation!`:	(T, `Integer`) → T	In–place p^{th}–powering

Usage

pExponentiation(a, n)
pExponentiation!(a, n)

Signature

pExponentiation: (R, `Integer`) → R

Parameter	Type	Description
a	R	The element to exponentiate
n	`Integer`	The exponent

Returns

Returns a^n. The exponent n must be nonnegative. When using pExponentiation!(a, n), the storage used by a is allowed to be destroyed or reused, so a is lost after this call.

Remarks

A call to pExponentiation!(a, n) may cause a to be destroyed, so do not use it unless a has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

import from SmallPrimeField p

Parameter	Type	Description
p	`MachineInteger`	The characteristic

Description

SmallPrimeField p implements the finite field $\mathbb{Z}/p\mathbb{Z}$ where $p \in \mathbb{Z}$ is a word–size prime.

Exports

`SmallPrimeFieldCategory`

Usage

import from SmallPrimeField0 p

Parameter	Type	Description
p	`MachineInteger`	The characteristic

Description

SmallPrimeField0 p is an internal implementation of the finite field $\mathbb{Z}/p\mathbb{Z}$ where $p \in \mathbb{Z}$ is a word–size prime. Use `SmallPrimeField` instead.

Exports

`SmallPrimeFieldCategory0`

Usage

SmallPrimeFieldCategory: Category

Description

SmallPrimeFieldCategory is the category for prime fields of the form $\mathbb{Z}/p\mathbb{Z}$ where $p \in \mathbb{Z}$ is a machine prime.

Exports

`PrimeFieldCategory`
`SmallPrimeFieldCategory0`
`UnivariateGcdRing`

Usage

SmallPrimeFieldCategory0: Category

Description

SmallPrimeFieldCategory0 is the category for prime fields of the form $\mathbb{Z}/p\mathbb{Z}$ where $p \in \mathbb{Z}$ is a machine prime.

Exports

`PrimeFieldCategory0`
`SerializableType`

`coerce`:	`MachineInteger` $\to$ %	conversion to a field element
`discreteLogTable`:	`Cross`(A, A, `Boolean`)	discrete log table if available
`machine`:	% $\to$ `MachineInteger`	conversion to a machine integer

where

A == `PrimitiveArray MachineInteger`

Usage

n::%
machine m

Signatures

coerce: `MachineInteger` → %
machine: % → `MachineInteger`

Parameter	Type	Description
m	%	an element of the field
n	`MachineInteger`	a machine integer

Returns

n::% returns n as an element of the field and machine(m) returns m as a `MachineInteger`.

Usage

((log, exp, ok?) := discreteLogTable

Signature

discreteLogTable: `Cross`(A, A, `Boolean`)

where

A == `PrimitiveArray MachineInteger`

Returns

Returns (log, exp, ok?) such that if ok? is *true*, then exp.i is g^i and log.i is $\log_g(i)$ where g is a generator of the multiplicative of the group (g is stored in exp.1).

Usage

import from ZechPrimeField p

Parameter	Type	Description
p	`MachineInteger`	The characteristic

Description

ZechPrimeField p implements the finite field $\mathbb{Z}/p\mathbb{Z}$, using discrete logarithm and exponential tables for multiplication, where $p \in \mathbb{Z}$ is a word–size prime.

Exports

`SmallPrimeFieldCategory`

Usage

import from Backsolve(R,M)

Parameter	Type	Description
R	`IntegralDomain`	A coefficient ring
M	`MatrixCategory` R	A matrix type over R

Description

(R,M) provides operations for backsolving triangular systems

Exports

`backsolve`:	(M,Z-¿Z,ARR Z,Z,Z,R) → V R	Backsolve a triangular system
`backsolve`:	(M,Z-¿Z,ARR Z,Z,M,R) → (M, V R)	Backsolve a triangular system

if R has `GcdDomain` then

`backsolve`:	(M,Z-¿Z,ARR Z,Z,Z) → V R	Backsolve a triangular system
`backsolve`:	(M,Z-¿Z,ARR Z,Z,M) → (M, V R)	Backsolve a triangular system
`backsolve2`:	(M,Z-¿Z,ARR Z,Z,Z,R) → V R	Backsolve a triangular system
`backsolve2`:	(M,Z-¿Z,ARR Z,Z,M,R) → (M, V R)	Backsolve a triangular system

where

Z	==	`MachineInteger`
ARR	==	`PrimitiveArray`
V	==	`Vector`

Usage

backsolve(a,p,st,r,c)
backsolve(a,p,st,r,b)
backsolve(a,p,st,r,c,d)
backsolve(a,p,st,r,b,d)

Signatures

backsolve: (M, Z → Z,`PrimitiveArray` Z,Z,Z) → `Vector` R
backsolve: (M,Z→ Z,`PrimitiveArray` Z,Z,M) → (M,`Vector` R)
backsolve: (M,Z → Z,`PrimitiveArray` Z,Z,Z,R) → `Vector` R
backsolve: (M,Z→ Z,`PrimitiveArray` Z,Z,M,R)→(M,`Vector` R)

Parameter	Type	Description
a	M	A matrix representing a Row Echelon Form (REF)
p	Z → Z	A permutation of the rows of a
st	`PrimitiveArray` Z	The stairs of the REF
r	Z	The number of leading columns (before the c^{th} column)
c	Z	The column to be backsolved
b	M	A matrix whose columns have to be backsolved
d	R	A maximal denominator needed for a dependence relation

where

Z == `MachineInteger`

Description

Backsolves a triangular system. The triple (a, p, st) represents a matrix in REF: for $j \geq st(i)$ entry (i,j) of the REF is stored in $a(p(i), j)$. The parameter d may be omitted if R is has `GcdDomain`, in which case the system is solved in a minimal way. Otherwise, d must be such that d times the c-th column of the REF (resp. d times the columns of b) is a linear combination of the first r leading columns of the REF (the j^{th} column is called leading if $j = st(i)$ for some i).

Returns

backsolve(a,p,st,r,c) returns a primitive vector v such that $av = 0$.
backsolve(a,p,st,r,c,d) returns a vector v such that $av = 0$, the c^{th} coordinate of v is d and otherwise $v(j) \neq 0$ only if $j \leq r$ and the j-th column is leading.
backsolve(a,p,st,r,b) returns a matrix s and a vector t such that when $t(l) \neq 0$, then a times the l^{th} column of s equals $t(l)$ times the l^{th} column of b, and when $t(l) = 0$, then the l^{th} column of b is not a linear combination of the columns of a. $s(j, l) \neq 0$ only if the j^{th} column is leading. Furthermore the gcd of all the entries in the l^{th} column of s and $t(l)$ is 1.
backsolve(a,p,st,r,b,d) returns a matrix s and a vector t such that when $t(l) \neq 0$, then a times the l^{th} column of s equals d times the l^{th} column of b, and when $t(l) = 0$, then the l^{th} column of b is not a linear combination of the columns of a. $s(j, l) \neq 0$ only if the j^{th} column is leading.

Usage

backsolve2(a,p,st,r,c,d)
backsolve2(a,p,st,r,b,d)

Signatures

backsolve2: (M,Z → Z,`PrimitiveArray` Z,Z,Z,R) → `Vector` R
backsolve2: (M,Z→ Z,`PrimitiveArray` Z,Z,M,R)→(M,`Vector` R)

Parameter	Type	Description
a	M	A matrix representing a Row Echelon Form (REF)
p	Z → Z	A permutation of the rows of a
st	`PrimitiveArray` Z	The stairs of the REF
r	Z	The number of leading columns (before the c^{th} column)
c	Z	The column to be backsolved
b	M	A matrix whose columns have to be backsolved
d	R	A maximal denominator needed for a dependence relation

where

Z == `MachineInteger`

Description

Backsolves a triangular system in a minimal way. The triple (a, p, st) represents a matrix in REF: for $j \geq st(i)$ entry (i,j) of the REF is stored in $a(p(i), j)$. The parameter d must be such that d times the c-th column of the REF (resp. d times the columns of b) is a linear combination of the first r leading columns of the REF (the j^{th} column is called leading if $j = st(i)$ for some i).

Returns

backsolve2(a,p,st,r,c,d) returns a primitive vector v such that $av = 0$, and $v(j) \neq 0$ only if $j = c$ or $j \leq r$ and the j^{th} column is leading.
backsolve2(a,p,st,r,b,d) returns a matrix s and a vector t such that when $t(l) \neq 0$, then a times the l^{th} column of s equals $t(l)$ times the l^{th} column of b, and when $t(l) = 0$, then the l^{th} column of b is not a linear combination of the columns of a. $s(j, l) \neq 0$ only if the j^{th} column is leading.

Usage

import from DenseMatrix R

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	

Description

DenseMatrix R provides dense mutable matrices with entries in R. They are 1–indexed and do not bound check.

Exports

`MatrixCategory` R

Usage

import from DivisionFreeGaussElimination(R,M)

Parameter	Type	Description
R	`CommutativeRing`	A coefficient ring
M	`MatrixCategory` R	A matrix type over R

Exports

`LinearEliminationCategory`(R,M)

Description

This domain implements division–free Gaussian elimination on matrices.

Usage

import from FractionFreeGaussElimination(R,M)

Parameter	Type	Description
R	`IntegralDomain`	A coefficient ring
M	`MatrixCategory` R	A matrix type over R

Exports

`LinearEliminationCategory`(R,M)

Description

This domain implements fraction–free Gaussian elimination on matrices.

Usage

import from HermiteGaussElimination(R,M)

Parameter	Type	Description
R	`EuclideanDomain`	A coefficient ring
M	`MatrixCategory` R	A matrix type over R

Exports

`LinearEliminationCategory`(R,M)

Description

This domain implements Hermite reduction on matrices.

Usage

import from LinearAlgebra(R, M)

Parameter	Type	Description
R	`CommutativeRing`	The coefficient domain
M	`MatrixCategory` R	A matrix type

Description

LinearAlgebra(R, M) provides basic linear algebra functionalities for matrices over R.

Exports

`invertibleSubmatrix`:	M → (`Boolean`, AZ, AZ)	Probable maximal minor
`maxInvertibleSubmatrix`:	M → (AZ, AZ)	Maximal minor
`span`:	M → AZ	Span

if R has `IntegralDomain` then

`cycle`:	(V → V, V) → (V, M)	First dependence relation
`cycle`:	(M, V) → (V, M)	First dependence relation
`determinant`:	M → R	Determinant
`factorOfDeterminant`:	M → (`Boolean`, R)	Probable determinant
`firstDependence`:	(`Generator` V, `MachineInteger`) → V	First dependence relation
`inverse`:	M → (M, V)	Inverse
`kernel`:	M → M	Kernel
`particularSolution`:	(M, M) → (M, V)	A solution
`rank`:	M → `MachineInteger`	Rank
`rankLowerBound`:	M → (`Boolean`, `MachineInteger`)	Probable rank
`solve`:	(M, M) → (M, M, V)	All solutions
`subKernel`:	M → (`Boolean`, M)	Subspace of the kernel

where

AZ == `Array MachineInteger`
V == `Vector` R

Usage

cycle(f,v)
cycle(A,v)

Signatures

cycle: (`Vector` R → `Vector` R, `Vector` R) → (`Vector` R, M)
cycle: (M, `Vector` R) → (`Vector` R, M)

Parameter	Type	Description
f	`Vector` R → `Vector` R	A map
A	M	A matrix
v	`Vector` R	A vector whose cycle is wanted

Returns

Returns $([a_0, \ldots, a_n], m)$ where

$$\sum_{i=0}^{n} a_i A^i v = 0 \quad \left(\text{resp. } \sum_{i=0}^{n} a_i f^i(v) = 0\right),$$

and the columns of m are $v, f(v), \ldots, f^n(v)$ (resp. $v, Av, \ldots, A^n v$).

Remarks

The relation is as small as possible, meaning that $v, f(v), \ldots, f^{n-1}(v)$ (resp. $v, Av, \ldots, A^{n-1}v$) are linearly independent over R. The iterates of v under f must all have the same dimension.

See also

`firstDependence`

Usage

determinant a

Signature

determinant: M → R

Parameter	Type	Description
a	M	A matrix

Returns

Returns the determinant of *a*.

See also

`factorOfDeterminant`

Usage

factorOfDeterminant a

Signature

factorOfDeterminant: M → (`Boolean`, R)

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(det?, d)$ such that d is always a factor of the determinant of a, and d is exactly the determinant of a if *det*? is *true*.

Remarks

d can also happen to be the determinant of a when *det*? is *false*, but the algorithm was unable to prove it.

See also

`determinant`

Usage

firstDependence(gen,n)

Signature

firstDependence: (`Generator Vector` R, `MachineInteger`) → `Vector` R

Parameter	Type	Description
gen	`Generator Vector` R	A generator of vectors
n	`MachineInteger`	The dimension of the vectors generated

Description

Returns a vector v which contains the coefficients of a dependence relation among the vectors generated by *gen*. The relation is as small as possible, meaning that if v has dimension m then the first $m-1$ vectors generated are linearly independent over R. The dimension of the vectors generated by *gen* must be n. There must be a relation between the vectors generated.

See also

`cycle`

Usage

inverse a

Signature

inverse: M → (M, `Vector` R)

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(b, [d_1, \ldots, d_n])$ such that

$$ab = \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

Remarks

$\prod_{i=1}^{n} d_i \neq 0$ if and only if a is invertible. In that case, $a^{-1} = bd^{-1}$ where d is the diagonal matrix with diagonal $d_1, \ldots, d_n$. To compute the inverse of a as a product of a diagonal matrix on the left rather than the right, let (b, d) be the result of calling inverse on `transpose`(a). Then, $(a^t)^{-1} = bd^{-1}$, so $a^{-1} = d^{-1}b^t$. When R is a `Field`, then $d_i \in \{0, 1\}$ for $1 \leq i \leq n$, so $b = a^{-1}$ when R is a `Field` and a is invertible.

Usage

invertibleSubmatrix a

Signature

invertibleSubmatrix: M → (`Boolean`, `Array MachineInteger`, `Array MachineInteger`)

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(max?, [r_1, \ldots, r_r], [c_1, \ldots, c_r])$ where $r \leq \text{rank}(a)$ and the submatrix of a formed by the intersections of the rows r_i and c_i is always invertible. If $max?$ is *true*, then r is exactly the rank of a and the given minor is of maximal size.

Remarks

r can also happen to be the rank of a when $max?$ is *false*, but the algorithm was unable to prove it.

See also

`maxInvertibleSubmatrix`

Usage
kernel a

Signature
kernel: M → M

Parameter	Type	Description
a	M	A matrix

Returns
Returns a matrix whose columns form a basis of the kernel of *a*.

See also
`solve`,`subKernel`

Usage

maxInvertibleSubmatrix a

Signature

maxInvertibleSubmatrix: M → (`Array MachineInteger`, `Array MachineInteger`)

Parameter	Type	Description
a	M	A matrix

Returns

Returns $([r_1, \ldots, r_r], [c_1, \ldots, c_r])$ where r is the rank of a and the submatrix of a formed by the intersections of the rows r_i and c_i is invertible.

See also

`invertibleSubmatrix`

Usage

particularSolution(a, b)

Signature

particularSolution: (M, M) → (M, `Vector` R)

Parameter	Type	Description
a, b	M	Matrices

Returns

Returns $(m, [d_1, \ldots, d_n])$ such that

$$am = b \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

Remarks

For each i, $d_i \neq 0$ if and only if the system $ax = i^{\text{th}}$ column of b has a solution, which is then d_i^{-1} times the i^{th} column of m. When R is a `Field`, then $d_i \in \{0, 1\}$ for $1 \leq i \leq n$, so m is a solution of $ax = b$ when R is a `Field` and all the d_i's are nonzero.

See also

`solve`

Usage
rank a

Signature
rank: M → `MachineInteger`

Parameter	Type	Description
a	M	A matrix

Returns
Returns the rank of *a*.

See also
`rankLowerBound`,`span`

Usage

rankLowerBound a

Signature

rankLowerBound: M → (`Boolean`, `MachineInteger`)

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(rank?, r)$ such that $r \leq \text{rank}(a)$, and r is exactly the rank of a if *rank*? is *true*.

Remarks

r can also happen to be the rank of a when *rank*? is *false*, but the algorithm was unable to prove it.

See also

`rank`,`span`

Usage

span a

Signature

span: M → `Array MachineInteger`

Parameter	Type	Description
a	M	A matrix

Returns

Returns $[c_1, \ldots, c_r]$ where r is the rank of a and the span of a is generated by its columns $c_1, \ldots, c_r$.

See also

`rank`

Usage

solve(a, b)

Signature

solve: (M, M) → (M, M, `Vector` R)

Parameter	Type	Description
a, b	M	Matrices

Returns

Returns $(w, m, [d_1, \ldots, d_n])$ such that the columns of w form a basis of the kernel of a and

$$am = b \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

Remarks

For each i, $d_i \neq 0$ if and only if the system $ax = i^{\text{th}}$ column of b has a solution, which is then d_i^{-1} times the i^{th} column of m. When R is a `Field`, then $d_i \in \{0, 1\}$ for $1 \leq i \leq n$, so the general solution of $ax = b$ when R is a `Field` and all the d_i's are nonzero is $x = m + \sum_j r_j w_j$ where w_j is the j^{th} column of w.

See also

`kernel`,`particularSolution`

Usage

subKernel a

Signature

subKernel: M → (`Boolean`, M)

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(ker?, m)$ such that the columns of m, which are always linearly independent over R, generate a subspace of the kernel of a, and generate the full kernel if $ker?$ is *true*.

Remarks

m can also happen to generate the full kernel of a when $ker?$ is *false*, but the algorithm was unable to prove it.

See also

`kernel`

Usage

this: Category

Description

LinearAlgebraRing is the category of rings that export algorithms for linear algebra for matrices over themselves.

Exports

`IntegralDomain`

`determinant`:	(M:MC) → M → %	Determinant
`factorOfDeterminant`:	(M:MC) → M → (B, %)	Probable determinant
`invertibleSubmatrix`:	(M:MC) → M → (B, AZ, AZ)	Probable maximal minor
`inverse`:	(M:MC) → M → (M, V)	Inverse
`kernel`:	(M:MC) → M → M	Kernel
`linearDependence`:	(`Generator` V, Z) → V	First dependence relation
`maxInvertibleSubmatrix`:	(M:MC) → M → (AZ, AZ)	Maximal minor
`particularSolution`:	(M:MC) → (M, M) → (M, V)	A solution
`rank`:	(M:MC) → M → Z	Rank
`rankLowerBound`:	(M:MC) → M → (B, Z)	Probable rank
`solve`:	(M:MC) → (M, M) → (M, M, V)	All solutions
`span`:	(M:MC) → M → AZ	Span
`subKernel`:	(M:MC) → M → (B, M)	Subspace of the kernel

where

AZ == `Array MachineInteger`
B == `Boolean`
MC == `MatrixCategory` %
V == `Vector` %
Z == `MachineInteger`

Usage

determinant(M)(a)

Signature

determinant: (M:`MatrixCategory` %) → M → R

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns the determinant of *a*.

See also

`factorOfDeterminant`

Usage

factorOfDeterminant(M)(a)

Signature

factorOfDeterminant: (M:`MatrixCategory` %) → M → (`Boolean`, R)

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns $(det?, d)$ such that d is always a factor of the determinant of a, and d is exactly the determinant of a if *det*? is *true*.

Remarks

d can also happen to be the determinant of a when *det*? is *false*, but the algorithm was unable to prove it.

See also

`determinant`

Usage

inverse(M)(a)

Signature

inverse: (M:`MatrixCategory` %) → M → (M, `Vector` R)

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns $(b, [d_1, \ldots, d_n])$ such that

$$ab = \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

Remarks

$\prod_{i=1}^{n} d_i \neq 0$ if and only if a is invertible. When R is a `Field`, then $d_i \in \{0, 1\}$ for $1 \leq i \leq n$, so $b = a^{-1}$ when R is a `Field` and a is invertible.

Usage

invertibleSubmatrix(M)(a)

Signature

invertibleSubmatrix: (M:`MatrixCategory` %) → M → (`Boolean`, AZ, AZ)

where

AZ == `Array MachineInteger`

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns $(max?, [r_1, \ldots, r_r], [c_1, \ldots, c_r])$ where $r \leq \text{rank}(a)$ and the submatrix of a formed by the intersections of the rows r_i and c_i is always invertible. If *max*? is *true*, then r is exactly the rank of a and the given minor is of maximal size.

Remarks

r can also happen to be the rank of a when *max*? is *false*, but the algorithm was unable to prove it.

See also

`maxInvertibleSubmatrix`

Usage

kernel(M)(a)

Signature

kernel: (M:`MatrixCategory` %) → M → M

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns a matrix whose columns form a basis of the kernel of *a*.

See also

`solve`,`subKernel`

Usage

linearDependence(gen,n)

Signature

linearDependence: (`Generator Vector` R, `MachineInteger`) → `Vector` R

Parameter	Type	Description
gen	`Generator Vector` R	A generator of vectors
n	`MachineInteger`	The dimension of the vectors generated

Description

Returns a vector v which contains the coefficients of a dependence relation among the vectors generated by *gen*. The relation is as small as possible, meaning that if v has dimension m then the first $m-1$ vectors generated are independent. The dimension of the vectors generated by *gen* must be n. There must be a relation between the vectors generated.

Usage

maxInvertibleSubmatrix(M)(a)

Signature

maxInvertibleSubmatrix: (M:`MatrixCategory` %) → M → (AZ, AZ)

where

AZ == `Array MachineInteger`

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns $([r_1, \ldots, r_r], [c_1, \ldots, c_r])$ where r is the rank of a and the submatrix of a formed by the intersections of the rows r_i and c_i is invertible.

See also

`invertibleSubmatrix`

Usage

particularSolution(M)(a, b)

Signature

particularSolution: (M:`MatrixCategory` %) → (M, M) → (M, `Vector` R)

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a, b	M	Matrices

Returns

Returns $(m, [d_1, \ldots, d_n])$ such that

$$am = b \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

Remarks

For each i, $d_i \neq 0$ if and only if the system $ax = i^{\text{th}}$ column of b has a solution, which is then d_i^{-1} times the i^{th} column of m. When R is a `Field`, then $d_i \in \{0, 1\}$ for $1 \leq i \leq n$, so m is a solution of $ax = b$ when R is a `Field` and all the d_i's are nonzero.

See also

`solve`

Usage

rank(M)(a)

Signature

rank: (M:`MatrixCategory` %) → M → `MachineInteger`

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns the rank of *a*.

See also

`rankLowerBound`,`span`

Usage
rankLowerBound(M)(a)

Signature
rankLowerBound: (M:`MatrixCategory` %) → M → (`Boolean`, `MachineInteger`)

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns
Returns $(rank?, r)$ such that $r \leq \text{rank}(a)$, and r is exactly the rank of a if *rank*? is *true*.

Remarks
r can also happen to be the rank of a when *rank*? is *false*, but the algorithm was unable to prove it.

See also
`rank`,`span`

Usage

solve(M)(a, b)

Signature

solve: (M:`MatrixCategory` %) → (M, M) → (M, M, `Vector` R)

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a, b	M	Matrices

Returns

Returns $(w, m, [d_1, \ldots, d_n])$ such that the columns of w form a basis of the kernel of a and

$$am = b \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

Remarks

For each i, $d_i \neq 0$ if and only if the system $ax = i^{\text{th}}$ column of b has a solution, which is then d_i^{-1} times the i^{th} column of m. When R is a `Field`, then $d_i \in \{0, 1\}$ for $1 \leq i \leq n$, so the general solution of $ax = b$ when R is a `Field` and all the d_i's are nonzero is $x = m + \sum_j r_j w_j$ where w_j is the j^{th} column of w.

See also

`kernel`,`particularSolution`

Usage

span(M)(a)

Signature

span: (M:`MatrixCategory` %) → M → `Array` `MachineInteger`

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns $[c_1, \ldots, c_r]$ where r is the rank of a and the span of a is generated by its columns $c_1, \ldots, c_r$.

See also

`rank`

Usage

subKernel(M)(a)

Signature

subKernel: (M:`MatrixCategory` %) → M → (`Boolean`, M)

Parameter	Type	Description
M	`MatrixCategory` %	A matrix type
a	M	A matrix

Returns

Returns $(ker?, m)$ such that the columns of m, which are always linearly independent over R, generate a subspace of the kernel of a, and generate the full kernel if *ker*? is *true*.

Remarks

m can also happen to generate the full kernel of a when *ker*? is *false*, but the algorithm was unable to prove it.

See also

`kernel`

Usage

LinearEliminationCategory(R,M): Category

Parameter	Type	Description
R	`CommutativeRing`	A coefficient ring
M	`MatrixCategory` R	A matrix type over R

Description

LinearEliminationCategory is a common category for linear elimination computations. The category provides operations for computating a row echelon form (REF) of a matrix and the first dependence relation among vectors.

Exports

`extendedRowEchelon`:	M → (M,Z → Z,Z,ARR Z,Z,M)	REF of a matrix
`extendedRowEchelon!`:	M → (Z → Z,Z,ARR Z,Z,M)	REF of a matrix
`extendedRowEchelonForm`:	M → (M,M)	REF of a matrix
`maxInvertibleSubmatrix`:	M → (`Array` Z, `Array` Z)	Maximal minor
`maxInvertibleSubmatrix!`:	M → (`Array` Z, `Array` Z)	Maximal minor
`pivot`:	(M, Z → Z, Z, Z) → Z	Select a pivot
`rowEchelon`:	M → (M,Z → Z,Z,ARR Z,Z)	REF of a matrix
`rowEchelon!`:	M → (Z → Z,Z,ARR Z,Z)	REF of a matrix
	(M, M) → (Z → Z,Z,ARR Z,Z)	
`rowEchelonForm`:	M → M	REF of a matrix
`span`:	M → `Array` Z	Span of a matrix
`span!`:	M → `Array` Z	Span of a matrix

if *R* has `IntegralDomain` then

`denominators`:	(M,Z → Z,Z,ARR Z) → ARR R	Maximal denominators
`dependence`:	(Generator V,Z) → (M,Z → Z,Z,R)	First linear dependence
`deter`:	(M,Z → Z,Z,Z) → R	Determinant
`determinant`:	M → R	Determinant
`determinant!`:	M → R	Determinant
`rank!`:	M → Z	Rank of a matrix
`rank`:	M → Z	Rank of a matrix

where

Z	==	`MachineInteger`
ARR	==	`PrimitiveArray`
V	==	`Vector` R

Usage

denominators(a,p,r,st)

Signature

denominators: (M,Z → Z,Z, `PrimitiveArray` Z) → `PrimitiveArray` R

where

Z == `MachineInteger`

Parameter	Type	Description
a	M	A matrix in REF form
p	`MachineInteger` → `MachineInteger`	A permutation of the rows of *a*
r	`MachineInteger`	The number of stairs of the REF
st	`PrimitiveArray MachineInteger`	The stairs of the REF

Description

(a,p,r,st) must be the representation of a REF computed by `rowEchelon`, `extendedRowEchelon` or their bang-versions. When d is returned then for $st(i) < j < st(i+1)$ we have that $d(i)$ times the j-th column of the REF is a linear combination of the first i leading columns of the REF. (The i-th column is called leading if i is a stair)

Usage

dependence(gen,n)

Signature

dependence: (`Generator Vector` R,Z) → (M,Z → Z,Z,R)

where

Z == `MachineInteger`

Parameter	Type	Description
gen	`Generator Vector` R	A generator of vectors
n	`MachineInteger`	The dimension of the vectors generated

Description

dependence(gen,n) computes the first dependence among the vectors generated. First means that dependence(gen,n) stops as soon as a dependence relation exists among the vectors generated so far. dependence(gen,n) returns a matrix a, a permutation p, the length r of a relation and the maximal denominator d needed for a dependence relation. After applying p to the rows of a one gets a matrix whose first $r-1$ columns form an upper-triangular matrix. d times the last column of a is a linear combination of the first $r-1$ columns of a. A dependence relation between the columns of a is also a dependence relation between the vectors generated.

Usage

deter(a,p,r,d)

Signature

deter: (M,Z → Z,Z,Z,Z) → R

where

Z == `MachineInteger`

Parameter	Type	Description
a	M	A matrix in REF form
p	`MachineInteger` → `MachineInteger`	A permutation of the rows of *a*
r	`MachineInteger`	The number of stairs of the REF
d	`MachineInteger`	The sign of p

Description

(a,p,r,d) must be the representation of a REF computed by `rowEchelon`, `extendedRowEchelon` or their bang-versions. The determinant of the original matrix is returned.

Usage

determinant a
determinant! a

Signature

determinant: M → R

Parameter	Type	Description
a	M	A matrix whose determinant has to be computed

Returns

Returns the determinant of *a*.

Remarks

determinant! does not make a copy of *a*, but performs all the computations in–place, modifying the entries of *a*.

Usage

extendedRowEchelon a
extendedRowEchelon! a

Signatures

extendedRowEchelon: M → (M,Z → Z,Z,`PrimitiveArray` Z,Z,M)
extendedRowEchelon!: M → (Z → Z,Z,`PrimitiveArray` Z,Z,M)

where

Z == `MachineInteger`

Parameter	Type	Description
a	M	A matrix whose REF and corresponding transformation matrix have to be computed

Description

We say that a matrix a is in REF if there are r (the rank) and $j_1 < j_2 < \cdots < j_r$ (the stairs) such that $a(i, j_i) \neq 0$, $a(i, j) = 0$ for $j < j_i$ and $a(i, j) = 0$ for $i > r$.
We say that a matrix b is a REF of the matrix a if b is in REF and there exists a non-singular matrix u such that $ua = b$.

extendedRowEchelon a computes a REF of a in a. It returns *(c,p,r,st,d,w)* where c is a matrix, p is a permutation, r is the number of stairs, st are the stairs, d is the sign of p and w is a matrix. For $i > r$, *st(i)* is set to $m + 1$ where m is the number of columns of a. For $j \geq j_i$ the entry (i, j) of the REF is stored as entry $(p(i), j)$ in c. The other entries of c may have random values. The entry (i, j) of the transformation matrix u is stored as entry $(p(i), j)$ in w.

Remarks

extendedRowEchelon! does not make a copy of a, but performs all the computations in–place, storing the final result in a.

See also

`extendedRowEchelonForm`

Usage

extendedRowEchelonForm a

Signature

extendedRowEchelonForm: M → (M,M)

Parameter	Type	Description
a	M	A matrix whose REF has to be computed

Description

We say that a matrix a is in REF if there are r (the rank) and $j_1 < j_2 < \cdots < j_r$ (the stairs) such that $a(i, j_i) \neq 0$, $a(i, j) = 0$ for $j < j_i$ and $a(i, j) = 0$ for $i > r$.

We say that a matrix b is a REF of the matrix a if b is in REF and there exists a non-singular matrix u such that $ua = b$.

Returns

Returns a REF b of a and the transformation matrix u such that $ua = b$.

See also

`extendedRowEchelon`

Usage

maxInvertibleSubmatrix a
maxInvertibleSubmatrix! a

Signature

maxInvertibleSubmatrix: M → (`Array MachineInteger`, `Array MachineInteger`)

Parameter	Type	Description
a	M	A matrix

Returns

Returns $([r_1, \ldots, r_r], [c_1, \ldots, c_r])$ where r is the rank of a and the submatrix of a formed by the intersections of the rows r_i and c_i is invertible.

Remarks

maxInvertibleSubmatrix! does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

See also

`rank`,`span`

Usage

pivot(a, p, c, r)

Signature

pivot: (M,Z → Z, Z, Z) → Z

where

Z == `MachineInteger`

Parameter	Type	Description
a	M	A matrix
p	`MachineInteger` → `MachineInteger`	A permutation
c	`MachineInteger`	A column index
r	`MachineInteger`	A row index

Returns

Returns the row index of the an appropriate pivot for column c at row r or below. The matrix considered is a with its rows permuted by p.

Usage

rank a
rank! a

Signature

rank: M → `MachineInteger`

Parameter	Type	Description
a	M	A matrix whose rank has to be computed

Returns

Returns the rank of *a*.

Remarks

rank! does not make a copy of *a*, but performs all the computations in–place, modifying the entries of *a*.

See also

`span`

Usage

rowEchelon a
rowEchelon! a
rowEchelon!(a, b)

Signatures

rowEchelon: M → (M,Z → Z,Z,`PrimitiveArray` Z,Z)
rowEchelon!: M → (Z → Z,Z,`PrimitiveArray` Z,Z)
rowEchelon!: (M, M) → (Z → Z,Z,`PrimitiveArray` Z,Z)

where

Z == `MachineInteger`

Parameter	Type	Description
a	M	A matrix whose REF has to be computed
b	M	A matrix to transform in the same way than a

Description

We say that a matrix a is in REF if there are r (the rank) and $j_1 < j_2 < \cdots < j_r$ (the stairs) such that $a(i, j_i) \neq 0$, $a(i, j) = 0$ for $j < j_i$ and $a(i, j) = 0$ for $i > r$.
We say that a matrix b is a REF of the matrix a if b is in REF and there exists a non-singular matrix u such that $ua = b$.

rowEchelon a computes a REF of a. It returns *(c,p,r,st,d)* where c is a matrix, p is a permutation, r is the number of stairs, st are the stairs and d is the sign of p. For $i > r$, *st(i)* is set to $m + 1$ where m is the number of columns of a. For $j \geq j_i$ the entry (i, j) of the REF is stored as entry $(p(i), j)$ in c. The other entries of c may have random values.
rowEchelon!(a, b) does the same than rowEchelon!(a) but applies all the elementary transformations applied to a also to b.

Remarks

rowEchelon! does not make a copy of a, but performs all the computations in–place, storing the final result in a. In addition, rowEchelon!(a, b) performs all the computations relative to b in b.

See also

`rowEchelonForm`

Usage

rowEchelonForm a

Signature

rowEchelonForm: M → M

Parameter	Type	Description
a	M	A matrix whose REF has to be computed

Description

We say that a matrix a is in REF if there are r (the rank) and $j_1 < j_2 < \cdots < j_r$ (the stairs) such that $a(i, j_i) \neq 0$, $a(i, j) = 0$ for $j < j_i$ and $a(i, j) = 0$ for $i > r$.

We say that a matrix b is a REF of the matrix a if b is in REF and there exists a non-singular matrix u such that $ua = b$.

Returns

Returns a REF of a.

See also

`rowEchelon!`

Usage

span a
span! a

Signature

span: M → `Array MachineInteger`

Parameter	Type	Description
a	M	A matrix whose span has to be computed

Returns

Returns $[c_1, \ldots, c_r]$ where r is the rank of a and the span of a is generated by its columns $c_1, \ldots, c_r$.

Remarks

span! does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

See also

`maxInvertibleSubmatrix`,`rank`

Usage

MatrixCategory R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

MatrixCategory R is a common category for matrices of arbitrary sizes with coefficients in R. They are 1–indexed and whether they do bound checking depends on each particular matrix type.

Exports

`CopyableType`

`LinearArithmeticType` R

∗:	(%,V) → V	multiplication by a vector
[]:	`Tuple` V → %	create a matrix
	`Generator` V → %	
`apply`:	(%,I,I) → R	extract an entry
`apply`:	(%,I,I,I,I) → %	extract a submatrix
`colCombine!`:	(%,R,I,R,I) → %	In–place linear combination of columns
	(%,R,I,R,I,I,I) → %	
	(%,(R,R)→ R,I,I) → %	
	(%,(R,R)→ R,I,I,I,I) → %	
`colSwap!`:	(%,I,I) *to* %	Swap columns in–place
	(%,I,I,I,I) *to* %	
`column`:	(%,I) → V	extraction of a column
`columns`:	% → `Generator` V	iteration over the columns
`companion`:	V → %	creates a companion matrix
	(V, R) → %	
`diagonal`:	V → %	creates a diagonal matrix
`diagonal?`:	% → `Boolean`	test for a diagonal matrix
`dimensions`:	% → (I,I)	get row and column dimensions
`map`:	(R → R) → V → %	lift a mapping
`map`:	(R → R) → % → %	lift a mapping
`map!`:	(R → R) → % → %	lift a mapping
`numberOfColumns`:	% → I	number of columns of the matrix
`numberOfRows`:	% → I	number of rows of the matrix
`one`:	I → %	identity matrix
`one?`:	% → `Boolean`	test for an identity matrix
`row`:	(%,I) → V	extraction of a row

`rowCombine!`:	(%,R,I,R,I) → %	In–place linear combination of rows
	(%,R,I,R,I,I,I) → %	
	(%,(R,R)→ R,I,I) → %	
	(%,(R,R)→ R,I,I,I,I) → %	
`rowSwap!`:	(%,I,I) *to* %	Swap rows in–place
	(%,I,I,I,I) *to* %	
`rows`:	% → `Generator` V	iteration over the rows
`set!`:	(%,I,I,R) → R	set an entry in the matrix
`setMatrix!`:	(%,I,I,%) → %	modify a submatrix of a matrix
`square?`:	% → `Boolean`	test for a square matrix
`tensor`:	(%, %) → %	tensor product
`transpose`:	V → %	transpose a vector
`transpose`:	% → %	transpose a matrix
`transpose!`:	% → %	transpose a matrix in–place
`zero`:	(I,I) → %	create a zero matrix
`zero!`:	% → ()	make all the entries zero
`zero?`:	% → `Boolean`	test if all entries are zero

if R has `Ring` then

`random`:	() → %	create a random matrix
	(I,I) → %	

if R has `DifferentialRing` then

`wronskian`: V → % Wronskian matrix

where

I == `MachineInteger`

V == `Vector` R

if R has `SerializableType` then

`SerializableType`

Usage

A $*$ v

Signature

$*$: (%,`Vector` R) $\to$ `Vector` R

Parameter	Type	Description
A	%	a matrix
v	`Vector` R	a vector

Returns

Returns the vector Av.

Usage

$[v_1, \ldots, v_n]$
[v for v in g]

Signatures

[]: `Tuple Vector` R → %
[]: `Generator Vector` R → %

Parameter	Type	Description
$v_1, \ldots, v_n$	`Vector` R	vectors
g	`Generator Vector` R	an iterator producing vectors

Returns

Returns the matrix whose i^{th} column is v_i, respectively the i^{th} vector generated by g.

Usage

apply(A,n,m)
apply(A,n,m,r,c)
A(n, m)
A(n, m, r, c)

Signatures

apply: (%,`MachineInteger`,`MachineInteger`) → R
apply: (%,`MachineInteger`,`MachineInteger`, `MachineInteger`,`MachineInteger`) → %

Parameter	Type	Description
A	%	A matrix
n,m,r,c	`MachineInteger`	indices

Returns

A(n,m) returns the entry of A at its n^{th} row and m^{th} column, while A(n,m,r,c) returns the submatrix of A having $A(m, n)$ in its top–left corner and r rows and c columns.

Usage

colCombine!(A, c_1, j_1, c_2, j_2)
colCombine!($A, c_1, j_1, c_2, j_2, i_1, i_2$)
colCombine!(A, f, j_1, j_2)
colCombine!(A, f, j_1, j_2, i_1, i_2)
rowCombine!(A, c_1, i_1, c_2, i_2)
rowCombine!($A, c_1, i_1, c_2, i_2, j_1, j_2$)
rowCombine!(A, f, i_1, i_2)
rowCombine!(A, f, i_1, i_2, j_1, j_2)

Signatures

colCombine!,rowCombine!: (%, R, I, R, I) → %
colCombine!,rowCombine!: (%, R, I, R, I, I, I) → %
colCombine!,rowCombine!: (%,(R,R) → R, I, I) → %
colCombine!,rowCombine!: (%, (R,R) → R, I, I, I, I) → %

where

I == `MachineInteger`

Parameter	**Type**	**Description**
A	%	A matrix
f	(R,R) → R	An binary operation on R
c1, *c2*	R	coefficients from R
j1, *j2*	`MachineInteger`	column indices
i1, *i2*	`MachineInteger`	row indices

Description

The $j_1{}^{\text{th}}$ column (resp. $i_1{}^{\text{th}}$ row) of A is replaced by the result of applying f pointwise to its $j_1{}^{\text{th}}$ and $j_2{}^{\text{th}}$ columns (resp. $i_1{}^{\text{th}}$ and $i_2{}^{\text{th}}$ rows). If the last 2 arguments i_1, i_2 (resp. j_1, j_2) are present, then this operation is applied only for rows i_1 to i_2 (resp. columns j_1 to j_2) inclusive. The form with c_1 and c_2 is equivalent to the first one with the function f defined by $f(x_1, x_2) = c_1x_1+c_2x_2$.

Usage

colSwap!(A, j_1, j_2)
colSwap!(A, j_1, j_2, i_1, i_2)
rowSwap!(A, i_1, i_2)
rowSwap!(A, i_1, i_2, j_1, j_2)

Signatures

colSwap!,rowSwap!: (%, I, I) → %
colSwap!,rowSwap!: (%, I, I, I, I) → %

where

I == `MachineInteger`

Parameter	Type	Description
A	%	A matrix
j1, *j2*	`MachineInteger`	column indices
i1, *i2*	`MachineInteger`	row indices

Description

The $j_1{}^{\text{th}}$ and $j_2{}^{\text{th}}$ columns (resp. $i_1{}^{\text{th}}$ and $i_2{}^{\text{th}}$ rows) of A are exchanged in–place. If the last 2 arguments i_1, i_2 (resp. j_1, j_2) are present, then this operation is applied only for rows i_1 to i_2 (resp. columns j_1 to j_2) inclusive.

Usage

column(A,n)
row(A,n)

Signature

colSwap!,rowSwap!: (%,`MachineInteger`) → `Vector` R

Parameter	Type	Description
A	%	A matrix
n	`MachineInteger`	An index

Returns

Returns the n^{th} column (resp. row) of A as a vector.

Usage

for v in columns A repeat { ... }
for v in rows A repeat { ... }

Signature

columns,rows: % → `Generator Vector` R

Parameter	Type	Description
A	%	A matrix

Description

This generator yields the columns (resp. rows) of A in succession.

Usage

companion $[r_1, \ldots, r_n]$
companion$([r_1, \ldots, r_n], a)$

Signature

companion: (`Vector` R, R) $\to$ %

Parameter	Type	Description
$r_1, \ldots, r_n$	R	Entries
a	R	A subdiagonal entry (optional, default is 1)

Returns

Returns the companion matrix

$$\begin{pmatrix} 0 & & & r_1 \\ a & \ddots & & r_2 \\ & \ddots & & r_3 \\ & & & \vdots \\ & & a & r_n \end{pmatrix}$$

See also

`diagonal`

Usage

diagonal $[r_1, \ldots, r_n]$
diagonal(r, n)
diagonal? A
square? A

Signatures

diagonal:	`Vector` R $\to$ %
diagonal:	(R, `MachineInteger`) $\to$ %
diagonal?,square?:	% $\to$ `Boolean`

Parameter	Type	Description
$r, r_1, \ldots, r_n$	R	Entries
n	`MachineInteger`	A size
A	%	A matrix

Description

diagonal($[r_1, \ldots, r_n]$) returns a diagonal matrix whose diagonal elements are $r_1, \ldots, r_n$, while square?(A) (resp. diagonal A) return *true* if A is a square (resp. diagonal) matrix, *false* otherwise.

See also

`companion`,`one?`

Usage

dimensions A

Signature

dimensions: % → (`MachineInteger`,`MachineInteger`)

Parameter	Type	Description
A	%	A matrix

Returns

The number of rows and columns in A

See also

`numberOfColumns`,`numberOfRows`

Usage

map f
map! f
map(f)($[v_1, \ldots, v_n]$)
map(f)(A)
map!(f)(A)

Signatures

map: (R → R) → `Vector` R → %
map: (R → R) → % → %
map!: (R → R) → % → %

Parameter	Type	Description
f	R → R	a map
v_i	R	Entries of a vector
A	%	A matrix

Description

map(f)($[v_1, \ldots, v_n]$) returns the square matrix

$$\begin{pmatrix} v_1 & \ldots & v_n \\ f(v_1) & \ldots & f(v_n) \\ \vdots & & \vdots \\ f^{n-1}(v_1) & \ldots & f^{n-1}(v_n) \end{pmatrix}$$

while map(f)(A) returns $f(A)$, *i.e.* f applied to A pointwise, and map(f) returns either the mapping $v \to f(v)$ or $A \to f(A)$. For matrices, map! does not make a copy of A but modifies it in place.

Usage

numberOfColumns A
numberOfRows A

Signature

numberOfColumns,numberOfRows: % → `MachineInteger`

Parameter	Type	Description
A	%	A matrix

Returns

The number of columns (resp. rows) in *A*.

See also

`dimensions`

Usage

one n
one? A

Signatures

one: `MachineInteger` → %
one?: % → `Boolean`

Parameter	Type	Description
n	`MachineInteger`	an integer
A	%	A matrix

Description

one(n) returns an $n \times n$ identity matrix, while one?(A) returns *true* if A is an identity matrix, *false* otherwise.

See also

`diagonal?`,`zero`,`zero?`

Usage

random()
random(n,m)

Signatures

random: () → %
random: (`MachineInteger`, `MachineInteger`) → %

Parameter	Type	Description
n,m	`MachineInteger`	The dimensions of the new matrix.

Returns

random() returns a random matrix with random size, while random(n, m) returns a random matrix with n rows and m columns.

Usage

set!(A, n, m, x)
A(n,m) := x

Signature

set!: (%,`MachineInteger`,`MachineInteger`,R) → R

Parameter	Type	Description
A	%	A matrix
n, m	`MachineInteger`	Indices
c	R	An entry

Description

Sets $A(n, m)$ to c and returns c.

Usage

setMatrix!(A, n, m, B)

Signature

setMatrix!: (%,`MachineInteger`,`MachineInteger`,%) → %

Parameter	Type	Description
A, B	%	Matrices
n, m	`MachineInteger`	Indices

Description

Inserts B as a submatrix of A starting at $A(n, m)$ and returns B.

Usage

tensor(A,B)

Signature

tensor: (%,%) → %

Parameter	Type	Description
A, B	%	Matrices

Returns

Returns $A \otimes B$, *i.e.* the matrix satisfying $(A \otimes B)(u \otimes v) = Au \otimes Bv$ for all vectors u, v.

Usage

transpose v
transpose A
transpose! A

Signatures

transpose:	`Vector` R → %
transpose,transpose!:	% → %

Parameter	Type	Description
v	`Vector` R	A vector
A	%	A matrix

Returns

Return the transpose of v (resp. A).

Remarks

transpose! does not make a copy of A, which is therefore replaced by its transpose. It is only applicable to square matrices.

Usage

wronskian $[v_1, \ldots, v_n]$

Signature

wronskian: `Vector` R $\to$ %

Parameter	Type	Description
v_i	R	Entries of a vector

Description

wronskian($[v_1, \ldots, v_n]$) returns the square matrix

$$\begin{pmatrix} v_1 & \ldots & v_n \\ v_1' & \ldots & v_n' \\ \vdots & & \vdots \\ v_1^{(n-1)} & \ldots & v_n^{(n-1)} \end{pmatrix}$$

Usage

zero(n,m)
zero! A
zero? A

Signatures

zero: (`MachineInteger`, `MachineInteger`) → %
zero!: % → ()
zero?: % → `Boolean`

Parameter	Type	Description
n,m	`MachineInteger`	integers
A	%	A matrix

Description

zero(n,m) returns an n by m zero matrix, while zero!(A) fills A with 0's and zero?(A) returns *true* if all the entries of A are 0, *false* otherwise.

See also

`one`,`one?`

Usage

import from MatrixCategoryOverFraction(R, MR, Q, MQ)

Parameter	Type	Description
R	`IntegralDomain`	an integral domain
MR	`MatrixCategory` R	a matrix type over R
Q	`FractionCategory` R	a fraction domain of R
MQ	`MatrixCategory` Q	a matrix type over Q

Description

MatrixCategoryOverFraction(R, MR, Q, MQ) provides useful conversions between matrices with integral and rational coefficients.

Exports

`LinearCombinationFraction`(R, MR, Q, MQ)

`makeColIntegral`:	(MQ, I) → (R, V R)	Convert to an integral column
	MQ → (V R, MR)	Convert to an integral matrix column by column
`makeRowIntegral`:	(MQ, I) → (R, V R)	Convert to an integral row
	MQ → (V R, MR)	Convert to an integral matrix row by row

if Q has `FractionByCategory0` R

`makeRowIntegralBy`:	MQ → (V Z, MR)	Convert to an integral matrix row by row

where

I == `MachineInteger`
Z == `Integer`
V == `Vector`

Usage

(v, A) := makeColIntegral B
(a, v) := makeColIntegral(B, i)

Signatures

makeColIntegral: MQ → (`Vector` R, MR)
makeColIntegral: (MQ, `MachineInteger`) → (R, `Vector` R)

Parameter	Type	Description
B	MQ	A matrix with rational coefficients
i	`MachineInteger`	A column index

Returns

makeColIntegral(B) returns (v, A) such that A has integral coefficients and the j^{th} column of A is equal to $v.j$ times the j^{th} column of B for each j, *i.e.*

$$A = B \begin{pmatrix} v_1 & & \\ & \ddots & \\ & & v_n \end{pmatrix}$$

makeColIntegral(B, i) returns (a, v) such that v has integral coefficients and v equals a times the i^{th} column of B. If R is a `GcdDomain`, then each column of A (resp. v) is primitive.

See also

`makeRowIntegral`

Usage

(v, A) := makeRowIntegral B
(a, v) := makeRowIntegral(B, i)

Signatures

makeRowIntegral: MQ → (`Vector` R, MR)
makeRowIntegral: (MQ, `MachineInteger`) → (R, `Vector` R)

Parameter	Type	Description
B	MQ	A matrix with rational coefficients
i	`MachineInteger`	A row index

Returns

makeRowIntegral(B) returns (v, A) such that A has integral coefficients and the j^{th} row of A is equal to $v.j$ times the j^{th} row of B for each j, *i.e.*

$$A = \begin{pmatrix} v_1 & & \\ & \ddots & \\ & & v_n \end{pmatrix} B$$

makeRowIntegral(B, i) returns (a, v) such that v has integral coefficients and v equals a times the i^{th} row of B. If R is a `GcdDomain`, then each row of A (resp. v) is primitive.

See also

`makeColIntegral`

Usage

(v, A) := makeRowIntegralBy B

Parameter	Type	Description
B	MQ	A matrix with rational coefficients

Returns

Returns (v, A) such that A has integral coefficients and the j^{th} row of A is equal to the j^{th} row of B shifted by $v.j$.

Usage

import from ModulopGaussElimination

Exports

`deter`:	(M, Z, V, Z, Z, Z) → Z	Determinant
`determinant!`:	(M, Z, Z) → Z	Determinant
`extendedRowEchelon!`:	(M, Z, Z, Z) → (V, Z, V, M)	REF of a matrix
`firstDependence!`:	(`Generator` V, Z, Z, M, M) → Z	First dependence relation
`inverse!`:	(M, Z, Z, M, V) → ()	Inverse
`kernel!`:	(M, Z, Z, M) → Z	Kernel
`maxInvertibleSubmatrix!`:	(M, Z, Z) → (V, V)	Maximal minor
`particularSolution!`:	(M, Z, Z, M, Z, Z, M, V) → ()	A solution
`rank!`:	(M, Z, Z, Z) → Z	Rank
`rowEchelon!`:	(M, Z, Z, Z) → (V, Z, V)	REF of a matrix
`solve!`:	(M, Z, Z, M, Z, Z, M, V, M) → Z	All solutions
`span!`:	(M, Z, Z) → V	Span

where

Z == `MachineInteger`
V == `PrimitiveArray MachineInteger`
M == `PrimitiveArray PrimitiveArray MachineInteger`

Description

This domain implements ordinary Gaussian elimination on dense matrices over the integers modulo a machine prime.

Usage

deter(a,n,σ,r,d,p)

Signature

deter: (`PrimitiveArray` `PrimitiveArray` Z,Z, `PrimitiveArray` Z,Z,Z) $\rightarrow$ Z

where

Z == `MachineInteger`

Parameter	Type	Description
a	A A `MachineInteger`	A square matrix in REF form
n	`MachineInteger`	The size of a
σ	A `MachineInteger`	A permutation of the rows of a
r	`MachineInteger`	The number of stairs of the REF
d	`MachineInteger`	The sign of σ
p	`MachineInteger`	a prime

where

A == `PrimitiveArray`

Description

(a,σ,r,d) must be the representation of a REF computed by either `extendedRowEchelon!` or `rowEchelon!`. The determinant of the original matrix over $\mathbb{Z}/p\mathbb{Z}$ is returned.

Usage

determinant!(a,n,p)

Signature

determinant!: (`PrimitiveArray PrimitiveArray` Z,Z,Z) → Z

where

Z == `MachineInteger`

Parameter	Type	Description
a	`PrimitiveArray PrimitiveArray MachineInteger`	A square matrix
n	`MachineInteger`	The size of a
p	`MachineInteger`	a prime

Returns

Returns the determinant of a over $\mathbb{Z}/p\mathbb{Z}$.

Remarks

Does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

Usage

extendedRowEchelon!(a, r, c, p)

Signature

extendedRowEchelon!: (A A Z,Z,Z) → (A Z, Z, A Z, Z, A A Z)

where

Z == `MachineInteger`
A == `PrimitiveArray`

Parameter	Type	Description
a	`PrimitiveArray PrimitiveArray MachineInteger`	A matrix
r	`MachineInteger`	Number of rows of a
c	`MachineInteger`	Number of columns of a
p	`MachineInteger`	A prime

Description

We say that a matrix a is in REF if there are r (the rank) and $j_1 < j_2 < \cdots < j_r$ (the stairs) such that $a(i, j_i) \neq 0$, $a(i, j) = 0$ for $j < j_i$ and $a(i, j) = 0$ for $i > r$.
We say that a matrix b is a REF of the matrix a if b is in REF and there exists a non-singular matrix u such that $ua = b$.

extendedRowEchelon!(a,r,c,p) computes a REF of a over $\mathbb{Z}/p\mathbb{Z}$. It returns *(σ,r,st,d,w)* where σ is a permutation, r is the number of stairs, st are the stairs, d is the sign of σ and w is a matrix. For $i > r$, *st(i)* is set to $m + 1$ where m is the number of columns of a. For $j \geq j_i$ the entry (i, j) of the REF is stored as entry $(p(i), j)$ in a. The other entries of a may have random values. The entry (i, j) of the transformation matrix u is stored as entry $(p(i), j)$ in w.

Remarks

Does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

See also

`rowEchelon!`

Usage

firstDependence!(gen, n, p, a, d)

Signature

firstDependence!: (`Generator` A Z, Z, Z, A A Z, A A Z) → Z

where

A == `PrimitiveArray`
Z == `MachineInteger`

Parameter	Type	Description
gen	`Generator` A `MachineInteger`	A generator of vectors
n	`MachineInteger`	The dimension of the vectors generated
p	`MachineInteger`	a prime
a	A A `MachineInteger`	A work matrix
d	A A `MachineInteger`	A matrix for the dependence

where

A == `PrimitiveArray`

Description

firstDependence!(gen,n,p,a,d) computes the first dependence over $\mathbb{Z}/p\mathbb{Z}$ among the vectors generated by *gen*. It returns the number r of vectors generated. This number is as small as possible, meaning that the first $r-1$ vectors are linearly independent over $\mathbb{Z}/p\mathbb{Z}$. The coefficients of the dependence are stored in the first column of d, which must have at least r rows, upon return. The work matrix a must have n rows and at least r columns (note that $r \leq n+1$). The dimension of the vectors generated by *gen* must be n. There must be a relation between the vectors generated.

Usage

inverse!(a,n,p,b,d)

Signature

inverse!: (A A Z, Z, Z, A A Z, A Z) → Z

where

Z == `MachineInteger`
A == `PrimitiveArray`

Parameter	Type	Description
a, b	`PrimitiveArray PrimitiveArray MachineInteger`	Square matrices
n	`MachineInteger`	The size of a, b and d
p	`MachineInteger`	a prime
d	`PrimitiveArray MachineInteger`	a vector

Returns

Fills b and d such that $d_i \in \{0, 1\}$ for each i and

$$ab = \begin{pmatrix} d_0 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & d_{n-1} \end{pmatrix}.$$

Remarks

$\prod_{i=0}^{n-1} d_i = 1$ if and only if a is invertible, in which case $b = a^{-1}$. Does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

Usage

kernel(a,r,c,p,w)

Signature

kernel: (A A Z,Z,Z,A A Z) → Z

where

Z == `MachineInteger`
A == `PrimitiveArray`

Parameter	Type	Description
a, w	`PrimitiveArray PrimitiveArray MachineInteger`	Matrices
r	`MachineInteger`	Number of rows of *a*
c	`MachineInteger`	Number of columns of *a*
p	`MachineInteger`	a prime

Returns

Stores a basis of the kernel of *a* in the columns of *w*, which must be large enough, and returns the dimension of the kernel.

Remarks

Does not make a copy of *a*, but performs all the computations in–place, modifying the entries of *a*.

Usage

maxInvertibleSubmatrix!(a,r,c,p)

Signature

maxInvertibleSubmatrix!: (A A Z,Z,Z) → (`Array` Z, `Array` Z)

where

Z == `MachineInteger`
A == `PrimitiveArray`

Parameter	Type	Description
a	`PrimitiveArray PrimitiveArray MachineInteger`	A matrix
r	`MachineInteger`	Number of rows of a
c	`MachineInteger`	Number of columns of a
p	`MachineInteger`	a prime

Returns

Returns $([r_1, \ldots, r_r], [c_1, \ldots, c_r])$ where r is the rank of a over $\mathbb{Z}/p\mathbb{Z}$ and the submatrix of a formed by the intersections of the rows r_i and c_i is invertible over $\mathbb{Z}/p\mathbb{Z}$.

Remarks

Does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

Usage

particularSolution!(a,ra,ca,b,cb,p,w,d)

Signature

particularSolution!: (A A Z,Z,Z,A A Z,Z,Z,A A Z,A Z) → ()

where

Z == `MachineInteger`
A == `PrimitiveArray`

Parameter	Type	Description
a,b,w	`PrimitiveArray PrimitiveArray MachineInteger`	Matrices
ra	`MachineInteger`	Number of rows of a
ca	`MachineInteger`	Number of columns of a
cb	`MachineInteger`	Number of columns of b
p	`MachineInteger`	a prime
d	`PrimitiveArray MachineInteger`	a vector

Returns

Fills w and d such that $d_i \in \{0, 1\}$ for each i and

$$aw = b \begin{pmatrix} d_0 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & d_{n-1} \end{pmatrix}.$$

Remarks

For each i, $d_i = 1$ if and only if the system $ax = (i+1)^{\text{th}}$ column of b has a solution, which is then the $(i+1)^{\text{th}}$ column of w, which must have the same dimensions than b, which must have the same number of rows that a. Does not make a copy of a, but performs all the computations in–place, modifying the entries of a, while b is not modified.

Usage

rank!(a,r,c,p)

Signature

rank!: (`PrimitiveArray` `PrimitiveArray` Z,Z,Z) → Z

where

Z == `MachineInteger`

Parameter	Type	Description
a	`PrimitiveArray` `PrimitiveArray` `MachineInteger`	A matrix
r	`MachineInteger`	Number of rows of a
c	`MachineInteger`	Number of columns of a
p	`MachineInteger`	a prime

Returns

Returns the rank of a over $\mathbb{Z}/p\mathbb{Z}$.

Remarks

Does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

Usage

rowEchelon!(a, r, c, p)

Signature

rowEchelon!: (A A Z,Z,Z) → (A Z, Z, A Z, Z)

where

Z == `MachineInteger`
A == `PrimitiveArray`

Parameter	Type	Description
a	`PrimitiveArray PrimitiveArray MachineInteger`	A matrix
r	`MachineInteger`	Number of rows of a
c	`MachineInteger`	Number of columns of a
p	`MachineInteger`	A prime

Description

We say that a matrix a is in REF if there are r (the rank) and $j_1 < j_2 < \cdots < j_r$ (the stairs) such that $a(i, j_i) \neq 0$, $a(i, j) = 0$ for $j < j_i$ and $a(i, j) = 0$ for $i > r$.
We say that a matrix b is a REF of the matrix a if b is in REF and there exists a non-singular matrix u such that $ua = b$.

rowEchelon!(a,r,c,p) computes a REF of a over $\mathbb{Z}/p\mathbb{Z}$. It returns *(σ,r,st,d)* where σ is a permutation, r is the number of stairs, *st* are the stairs and d is the sign of p. For $i > r$, *st(i)* is set to $m + 1$ where m is the number of columns of a. For $j \geq j_i$ the entry (i, j) of the REF is stored as entry $(p(i), j)$ in a. The other entries of a may have random values.

Remarks

Does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

See also

`extendedRowEchelon!`

Usage

solve!(a,ra,ca,b,cb,p,w,d,k)

Signature

solve!: (A A Z,Z,Z,A A Z,Z,Z,A A Z,A Z,A A Z) → Z

where

Z == `MachineInteger`
A == `PrimitiveArray`

Parameter	Type	Description
a,b,w,k	`PrimitiveArray PrimitiveArray MachineInteger`	Matrices
ra	`MachineInteger`	Number of rows of a
ca	`MachineInteger`	Number of columns of a
cb	`MachineInteger`	Number of columns of b
p	`MachineInteger`	a prime
d	`PrimitiveArray MachineInteger`	a vector

Returns

Fills w and d such that $d_i \in \{0, 1\}$ for each i and

$$aw = b \begin{pmatrix} d_0 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & d_{n-1} \end{pmatrix}.$$

Furthermore, solve! stores a basis of the kernel of a in the columns of k, which must be large enough, and returns the dimension of the kernel.

Remarks

For each i, $d_i = 1$ if and only if the system $ax = (i+1)^{\text{th}}$ column of b has a solution, which is then the $(i+1)^{\text{th}}$ column of w, which must have the same dimensions than b, which must have the same number of rows that a. Does not make a copy of a, but performs all the computations in–place, modifying the entries of a, while b is not modified.

Usage

span!(a,r,c,p)

Signature

span!: (`PrimitiveArray PrimitiveArray` Z,Z,Z) → `Array` Z

where

Z == `MachineInteger`

Parameter	Type	Description
a	`PrimitiveArray PrimitiveArray MachineInteger`	A matrix
r	`MachineInteger`	Number of rows of a
c	`MachineInteger`	Number of columns of a
p	`MachineInteger`	a prime

Returns

Returns $[c_1, \ldots, c_r]$ where r is the rank of a over $\mathbb{Z}/p\mathbb{Z}$ and the span of a is generated by its columns $c_1, \ldots, c_r$.

Remarks

Does not make a copy of a, but performs all the computations in–place, modifying the entries of a.

Usage

import from OrdinaryGaussElimination(F,M)

Parameter	Type	Description
F	`Field`	A coefficient field
M	`MatrixCategory` F	A matrix type over F

Exports

`LinearEliminationCategory`(R,M)

Description

This domain implements ordinary Gaussian elimination on matrices.

Usage

import from OverdeterminedLinearSystemSolver(R, M)

Parameter	Type	Description
R	`IntegralDomain`	A domain
M	`MatrixCategory` R	A matrix type over R

Description

OverdeterminedLinearSystemSolver(R, M) provides a solver for overdetermined linear algebraic equations with coefficients in R.

Exports

`kernel!`	(M, M → M) → M	Solve an overdetermined system

Usage

kernel!(m, f)

Signature

kernel!: (M, M → M) → M

Parameter	Type	Description
m	M	A matrix
f	M → M	Computes a kernel

Returns

Returns a basis for the kernel of m, uses a specialized algorithm if m is overdetermined, uses f when the system is no longer overdetermined.

Remarks

Can destroy m.

Usage

import from SpecializationLinearAlgebra(R, M)

Parameter	Type	Description
R	`CommutativeRing` `Specializable`	The coefficient domain
M	`MatrixCategory` R	A matrix type

Description

SpecializationLinearAlgebra(R, M) provides basic linear algebra functionalities using specializations for matrices over R.

Exports

`rankLowerBound`: M → `Partial Cross` (`Boolean`, `MachineInteger`) Probable rank

Usage

rankLowerBound a

Signature

rankLowerBound: M → `Partial Cross`(`Boolean`, `MachineInteger`)

Parameter	Type	Description
a	M	A matrix

Returns

Returns *failed* if specialization failed, otherwise $(rank?, r)$ such that $r \leq \text{rank}(a)$, and r is exactly the rank of a if $rank?$ is *true*.

Remarks

r can also happen to be the rank of a when $rank?$ is *false*, but the algorithm was unable to prove it.

Usage

import from TwoStepFractionFreeGaussElimination(R,M)

Parameter	Type	Description
R	`IntegralDomain`	A coefficient ring
M	`MatrixCategory` R	A matrix type over R

Exports

`LinearEliminationCategory`(R,M)

Description

This domain implements two-step fraction–free Gaussian elimination on matrices.

Usage

import from UnivariatePolynomialCRTLinearAlgebra(R, RX, M)

Parameter	Type	Description
R	`IntegralDomain`	The coefficient ring
RX	`UnivariatePolynomialCategory` R	Polynomials over R
M	`MatrixCategory` RX	A matrix type over RX

Description

UnivariatePolynomialCRTLinearAlgebra(F, FX, M) provides basic linear algebra functionalities using the Chinese Remainder Theorem from RX to R for matrices over RX.

Exports

`degreeBound`:	M → `Integer`)	Degree bound for the determinant
`determinant`:	M → RX	Determinant
	(M, RX) → RX	
	(M, RX, `Integer`) → RX	

Usage

degreeBound a

Signature

degreeBound: M → `Integer`

Parameter	Type	Description
a	M	A matrix

Returns

Returns n such that $\deg |a| \leq n$.

Usage

determinant a
determinant(a, d)
determinant(a, d, n)

Signatures

determinant: M → RX
determinant: (M, RX) → RX
determinant: (M, RX, `Integer`) → RX

Parameter	Type	Description
a	M	A matrix
d	RX	A known factor of $\|a\|$ (optional)
n	`Integer`	A known degree bound on $\|a\|$ (optional)

Returns

All calls to determinantreturn the determinant of a.

Remarks

The extra parameters d and n are optional. If they are provided, then d must divide $|a|$ exactly, and n must be such that $\deg |a| \leq n$.

Usage

import from UnivariatePolynomialPopovLinearAlgebra(F, FX, M)

Parameter	Type	Description
F	`Field`	The coefficient field
FX	`UnivariatePolynomialCategory` F	Polynomials over F
M	`MatrixCategory` FX	A matrix type over FX

Description

UnivariatePolynomialPopovLinearAlgebra(F, FX, M) provides basic linear algebra functionalities using weak Popov forms for matrices over FX.

Exports

`determinant`:	M → FX	Determinant
`hermite`:	M → M	Hermite form
`kernel`:	M → M	Kernel
`maxInvertibleSubmatrix`:	M → (AZ, AZ)	Maximal minor
`popov`:	M → M	weak Popov form
`rank`:	M → `MachineInteger`	Rank
`span`:	M → AZ	Span

where

AZ == `Array MachineInteger`

Usage

hermite a

Signature

hermite: M → M

Parameter	Type	Description
a	M	A matrix

Returns

Returns the Hermite form of *a*.

Usage

import from Vector R

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`AdditiveType`	

Description

Vector R provides vectors of arbitrary size with entries in R. They are 1–indexed and without bound checking.

Exports

`AdditiveType`
`BoundedFiniteLinearStructureType` R
`ExpressionType`

`zero`:	`MachineInteger` → %	zero vector
`zero!`:	% → ()	make all the entries zero
`zero?`:	% → `Boolean`	test if all entries are zero

if R has `ArithmeticType` then
`LinearCombinationType` R

`dot`:	(%, %) → R	dot product
`tensor`:	(%, %) → %	tensor product

if R has `Ring` then

`random`:	() → %	random vector
	`MachineInteger` → %	

Usage

dot(u,v)

Signature

dot: (%,%) → R

Parameter	Type	Description
u, v	%	Vectors of the same size

Returns

Returns $u \cdot v = \sum_i u_i v_i$.

Usage

random()
random n

Signatures

random: () → %
random: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	The dimension of the new vector.

Returns

random() returns a random vector of size at most 100, while random(n) returns a random vector of size n.

Usage

tensor(u,v)

Signature

tensor: (%,%) → %

Parameter	Type	Description
u, v	%	Vectors with coefficients from R.

Returns

Returns $u \otimes v = (u_1v_1, u_1v_2, \ldots, u_1v_m, u_2v_1, \ldots, u_nv_m)$.

Usage

zero n
zero! v
zero? v

Signatures

zero: `MachineInteger`) → %
zero!: % → ()
zero?: % → `Boolean`

Parameter	Type	Description
n	`MachineInteger`	The dimension of the new vector
v	%	A vector with coefficients from R

Description

zero(n) returns a zero vector of size n, while zero!(v) fills v with 0's and zero?(v) returns *true* if all the entries of v are 0, *false* otherwise.

Usage

import from VectorOverFraction(R, Q)

Parameter	Type	Description
R	`IntegralDomain`	an integral domain
Q	`FractionCategory` R	a fraction domain over R

Description

VectorOverFraction(R, Q) provides useful conversions between vectors with integral and rational coefficients.

Exports

`LinearCombinationFraction`(R,`Vector` R,Q,`Vector` Q)

Usage

import from DenseUnivariatePolynomial R
import from DenseUnivariatePolynomial(R, x)

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain
x	`Symbol`	The variable name (optional)

Description

DenseUnivariatePolynomial(R, x) implements dense univariate polynomials with coefficients in R.

Exports

`UnivariatePolynomialCategory` R

Usage

import from DenseUnivariateTaylorSeries R
import from DenseUnivariateTaylorSeries(R, x)

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain
x	`Symbol`	The variable name (optional)

Description

DenseUnivariateTaylorSeries(R, x) implements univariate Taylor series with coefficients in R.

Exports

`UnivariateTaylorSeriesCategory` R

Usage

FactorizationRing: Category

Description

FactorizationRing is the category of rings that export an algorithm for factoring univariate polynomials over themselves into irreducibles.

Exports

`GcdDomain`
`RationalRootRing`

`factor`:	(P:POL %) → P → (%, Product P)	Factorization into irreducibles
`fractionalRoots`:	(P:POL %) → P → `Generator` FR %	Roots in the fraction field
`roots`:	(P:POL %) → P → `Generator` FR %	Roots in the coefficient ring

where

FR == `FractionalRoot`
POL == `UnivariatePolynomialCategory0`

Usage

factor(P)(p)

Signature

factor: (P: `UnivariatePolynomialCategory0` %) → P → (%, `Product` P)

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	A polynomial type
p	P	A polynomial

Returns

Returns $(c, p_1^{e_1} \cdots p_n^{e_n})$ such that each p_i is irreducible, the p_i's have no common factors, and

$$p = c \prod_{i=1}^{n} p_i^{e_i} .$$

Usage

fractionalRoots(P)(p)
roots(P)(p)

Signature

fractionalRoots,roots: (P: POL %) → P → `Generator FractionalRoot` %

where

POL == `UnivariatePolynomialCategory0`

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	A polynomial type
p	P	A polynomial

Returns

Returns a generator that produces all the roots *p* either in the ring or in its fraction field.

Usage

FFTRing: Category

Description

FFTRing is the category of rings that export an algorithm for the FFT product of univariate polynomials over themselves.

Exports

`CommutativeRing`

`fft`:	(P:POL %) → (P, P) → `Partial` P	FFT product
`fft!`:	(P:POL %) → (P, P, P) → `Boolean`	FFT product
`fftCutoff`:	→ `MachineInteger`	Cutoff for FFT in $R[x]$

where

POL == `UnivariatePolynomialCategory0`

Usage

fft(P)(p,q)

Signature

fft: (P: `UnivariatePolynomialCategory0` %) → (P, P) → `Partial` P

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	A polynomial type
p,q	P	Polynomials

Returns

Returns the product *pq* computed using FFT.

See also

`fft!`

Usage

fft!(P)(r,p,q)

Signature

fft!: (P: `UnivariatePolynomialCategory0` %) → (P, P, P) → `Boolean`

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	A polynomial type
r,p,q	P	Polynomials

Returns

Replaces r by $r + pq$ where the product is computed using FFT. The space occupied by the first argument r is allowed to be reused. Returns *true* if the product could not be computed by the FFT method, *false* otherwise.

See also

`fft`

Usage
fftCutoff

Signature
fftCutoff: `MachineInteger`

Returns
Returns n such that the FFT multiplication is used in $R[x]$ for polynomials of degree greater than or equal to n.

Remarks
If this constant is 0, then FFT multiplication is not used at all in $R[x]$.

Usage

import from HeuristicGcd(Z, P)

Parameter	Type	Description
Z	`IntegerCategory`	An integer–like ring
P	`UnivariatePolynomialCategory0` Z	Polynomials over Z

Description

HeuristicGcd provides an implementation of the HEUGCD algorithm for univariate polynomials over the integers.

Exports

`balancedRemainder`:	(Z, Z) → Z	Symmetric Remainder
`heuristicGcd`:	(P, P) → (`Partial` P, P, P)	the Heuristic gcd algorithm
`radixInterpolate`:	(Z, Z) → P	Radix interpolation

Usage

balancedRemainder(n, m)

Signature

balancedRemainder: (Z, Z) → Z

Parameter	Type	Description
n	Z	An integer
m	Z	An integer

Returns

Returns $-m/2 \leq r < m/2$ such that $n \equiv r \pmod{m}$.

Usage

heuristicGcd(p_1, p_2)

Signature

heuristicGcd: (P, P) → (`Partial` P, P, P)

Parameter	Type	Description
p_1, p_2	P	Polynomials

Returns

Returns (g, q_1, q_2) such that $g = \gcd(p_1, p_2)$, $p_1 = gq_1$ and $p_2 = gq_2$.

Remarks

This heuristic can fail in theory, in which case (*failed*, p_1, p_2) is returned, although this has never been reported.

Usage

radixInterpolate(n, m)

Signature

radixInterpolate: (Z, Z) → P

Parameter	Type	Description
n	Z	A point
m	R	The value of the desired polynomial at n

Returns

Returns the unique polynomial p such that $p(n) = m$ and

$$||p||_\infty \leq \frac{n-1}{2}\,.$$

Remarks

The above bound is useful when an upper bound M on $||p||_\infty$ is known a priori, since it is then sufficient to take $n \geq 2M + 1$.

Usage

import from ModularUnivariateGcd(Z, P)

Parameter	Type	Description
Z	`IntegerCategory`	An integer-like ring
P	`UnivariatePolynomialCategory` Z	Polynomials over Z

Description

ModularUnivariateGcd provides an implementation of a modular GCD algorithm for univariate polynomials over the integer, using the Chinese Remainder Theorem.

Exports

`modularGcd`: (P, P) → (`Partial` P, P, P) The Modular Gcd algorithm

Usage

modularGcd(p_1, p_2)

Signatures

modularGcd: (P, P) → (`Partial` P, P, P)

Parameter	Type	Description
p_1, p_2	P	Polynomials over Z

Returns

Returns (g, y, z) such that $g = \gcd(p_1, p_2)$ or *failed*, and if g is not *failed*, then $p = yg$ and $q = zg$.

Remarks

This algorithm can fail because it runs out of primes. This will happen if the gcd has coefficients with more than around 3000 digits.

Usage

import from ModulopUnivariateGcd

Description

ModulopUnivariateGcd provides an implementation of an inplace gcd for polynomials modulo a machine prime.

Exports

`gcd!`: (ARR Z, Z, ARR Z, Z, Z, Z) → (ARR Z, Z, Z) in-place gcd

where

Z == `MachineInteger`
ARR == `PrimitiveArray`

Usage

gcd!(a, n, b, m, α, p)
gcd!(a, n, b, m, α, p, $[l_1, \ldots, l_{p-1}], [e_1, \ldots, e_{p-1}]$)

Signatures

gcd!: (ARR Z, Z, ARR Z, Z, Z) → (ARR Z, Z, Z)
gcd!: (ARR Z, Z, ARR Z, Z, Z, ARR Z, ARR Z) → (ARR Z, Z, Z)

where

Z == `MachineInteger`
ARR == `Array`

Parameter	Type	Description
a,b	`PrimitiveArray MachineInteger`	polynomials modulo p
n,m	`MachineInteger`	their degrees
α	`MachineInteger`	a leading coefficient
p	`MachineInteger`	a prime
$[l_0, \ldots, l_{p-1}]$	`PrimitiveArray MachineInteger`	log table
$[e_0, \ldots, e_{p-1}]$	`PrimitiveArray MachineInteger`	exp table

Description

Given 2 polynomials stored in a and b of degrees n and m respectively, computes a gcd(a, b) in $F_p[x]$ with leading coefficient α. Requires $n \geq m$ and that the polynomials are stored leading coefficient first. Returns the degree of the gcd and its starting index in the array.

Remarks

The result can be stored in either a or b, so the function also returns the appropriate array. Note that both a and b are destroyed. The last 2 optional arrays are such that $g^{l_i} = i$ and $e_i = g^i$ where g is a primitive root for the multiplicative group modulo p. They are used for fast multiplication if provided.

Usage

MonogenicAlgebra R: Category

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	

Description

MonogenicAlgebra is a common category for commutative and noncommutative univariate polynomials with coefficients in an arbitrary arithmetic system R and with respect to an arbitrary basis $(P_n)_{n\geq 0}$.

Exports

`CopyableType`
`IndexedFreeAlgebra`(R, `Integer`)
`MonogenicLinearArithmeticType` R

`coerce`:	`Vector` R → %	Create a polynomial
`companion`:	% → `DenseMatrix` R	Companion matrix
`monomial!`:	(%, R, Z) → %	In-place monomial
`random`:	(`Integer`, () → R, `Integer`) → %	Creation of a random polynomial
`revert`:	% → %	Left–Right reversion
`revert!`:	% → %	In–place left–right reversion
`vectorize!`:	`Vector` R → % → `Vector` R	Conversion to a vector

where

Z == `Integer`

if R has `GcdDomain` then

`content`:	(%, `Automorphism` R) → R	Polynomial content
`primitive`:	(%, `Automorphism` R) → (R, %)	Content and primitive part
`primitivePart`:	(%, `Automorphism` R) → %	

Primitive part

if R has `HashType` then
`HashType`

if R has `Ring` then

`random`:	(`Integer`, `Integer`) → %	Creation of a random polynomial

if R has `SerializableType` then
`SerializableType`

Usage

v::%
vectorize! v
vectorize!(v)(p)

Signatures

coerce: `Vector` R → %
vectorize!: `Vector` R → % → `Vector` R

Parameter	Type	Description
p	%	A polynomial
v	`Vector` R	A coefficient vector

Description

$[v_1, \ldots, v_n]$::% returns the polynomial $\sum_{i=0}^{n-1} v_{i+1} P_i$, while vectorize!$(v)(\sum_{j=0}^{d} a_j P_j)$ fills v with $[a_0, \ldots, a_d, 0, \ldots]$ and returns v.

Remarks

If $d > \#v - 1$, then the high coefficients of p are simply ignored.

Usage

companion p

Signature

companion: % → `DenseMatrix` R

Parameter	Type	Description
p	%	A polynomial

Returns

Returns the companion matrix

$$\begin{pmatrix} 0 & & & -a_0 \\ a_n & \ddots & & -a_1 \\ & \ddots & & -a_2 \\ & & & \vdots \\ & & a_n & -a_{n-1} \end{pmatrix}$$

where $p = \sum_{i=0}^{n} a_i P_i$.

Usage

content(p, σ)
primitive(p, σ)
primitivePart(p, σ)

Signatures

content: (%, `Automorphism` R) → R
primitive: (%, `Automorphism` R) → (R, %)
primitivePart: (%, `Automorphism` R) → %

Parameter	Type	Description
p	%	A polynomial
σ	`Automorphism` R	An automorphism of R

Description

content(p, σ) returns

$$\gcd(a_0, \sigma a_1, \ldots, \sigma^n a_n)$$

where $p = \sum_{i=0}^{n} a_i P_i$, while primitive(p, σ) and primitivePart(p, σ) return respectively $(c, c^{-1}p)$ and $c^{-1}p$ where c = content(p, σ). For efficiency, do not use those functions when σ is the identity function, but use `content`, `primitive` and `primitivePart` instead.

Usage

monomial!(p, c, n)

Signature

monomial!: (%, R, `Integer`) → %

Parameter	Type	Description
p	%	A polynomial (to be destroyed)
c	R	A scalar
n	`Integer`	An exponent

Returns

Returns the monomial cP_n.

Remarks

The storage used by p is allowed to be destroyed or reused so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

random(n[, m])
random(n, f[, m])

Signatures

random: (`Integer`, `Integer`) → %
random: (`Integer`, () → R, `Integer`) → %

Parameter	Type	Description
n	`Integer`	The desired degree
f	() → R	A random generator for R
m	`Integer`	The desired number of terms (optional)

Returns

Returns a monic random polynomial of degree n with at most m terms, ($n+1$ terms if m is not present). Uses $f()$ to generate the coefficients if the parameter f is present, the `random` function otherwise.

Usage

revert p
revert! p

Signature

revert: $\% \to \%$

Parameter	Type	Description
p	$\%$	A polynomial

Returns

Returns $\sum_{i=0}^{n} a_i P_{n-i}$ where $p = \sum_{i=0}^{n} a_i P_i$ and $a_n \neq 0$. Returns 0 if $p = 0$.

Remarks

The storage used by p is allowed to be destroyed or reused when revert! is used, so p is lost after those calls. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

`terms`

Usage

times(p, q, l, h)

Signature

times: (%, %, `Integer`, `Integer`) → %

Parameter	Type	Description
p, q	%	Polynomials
l, h	`Integer`	Lower and upper bound of coefficients to be computed

Returns

times(p, q, l, h) returns a polynomial s containing some coefficients of the product pq. The ith coefficient of s equals the $l + i$th coefficient of pq. Here $l \leq h$ should hold.

Usage

import from MonogenicAlgebra2(R,Rx,S,Sy)

Parameter	Type	Description
R, S	`ExpressionType` `ArithmeticType`	Coefficient domains
Rx	`MonogenicAlgebra` R	A monogenic algebra over R
Sy	`MonogenicAlgebra` S	A monogenic algebra over R

Description

MonogenicAlgebra2(R,Rx,S,Sy) provides tools for lifting maps $R \to S$ to maps $Rx \to Sy$.

Exports

`map`: (R → S) → Rx → Sy Lift a mapping

Usage

map f
map(f)(p)

Signature

map: (R → S) → Rx → Sy

Parameter	Type	Description
f	R → S	A map
p	%	A polynomial with coefficient in R

Description

map(f)(p) returns

$$f(p) = \sum_i f(a_i)y^i$$

where $p = \sum_i a_i x^i$, while map(f) returns the mapping $p \to f(p)$.

Usage

import from MonogenicAlgebraOverFraction(R, PR, Q, PQ)

Parameter	Type	Description
R	`IntegralDomain`	an integral domain
PR	`MonogenicAlgebra` R	a univariate free finite algebra type over R
Q	`FractionCategory` R	a fraction domain of R
PQ	`MonogenicAlgebra` R	a univariate free finite algebra type over Q

Description

MonogenicAlgebraOverFraction(R, PR, Q, PQ) provides useful conversions between polynomials with integral and rational coefficients.

Exports

`LinearCombinationFraction`(R, PR, Q, PQ)

Usage

MonogenicLinearArithmeticType R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

MonogenicLinearArithmeticType is a common category for commutative and noncommutative univariate polynomials and power series with coefficients in an arbitrary arithmetic system R and with respect to an arbitrary basis $(P_n)_{n\geq 0}$. Its elements are not assumed to have finite support, so this type cannot be asserted to be an `Algebra` R even when R is a `Ring`.

Exports

`IndexedFreeLinearArithmeticType`(R, Z)

`add!`:	(%, R, Z) → %	In-place addition of a term
`add!`:	(%, R, Z, %) → %	In-place product and sum
`apply`:	(%, TREE) → TREE	Conversion to an expression tree
`apply`:	(OUT, %, `Symbol`) → OUT	Write an element to a port
`coefficients`:	% → `Generator` R	Iterate over all the coefficients
`monom`:	→ %	Term with degree 1 and coefficient 1
`setCoefficient!`:	(%, Z, R) → %	In-place replacement of a coefficient
`shift`:	(%, Z) → %	Exponent translation
`shift!`:	(%, Z) → %	In-place exponent translation
`truncate`:	(%, Z) → %	Truncation
`truncate!`:	(%, Z) → %	In-place truncation

where

OUT == `TextWriter`
TREE == `ExpressionTree`
Z == `Integer`

Usage

apply(p, t)
p t
apply(port, p, x)
port(p, x)

Signatures

apply: (%, `ExpressionTree`) → `ExpressionTree`
apply: (`TextWriter`, %, `Symbol`) → `TextWriter`

Parameter	Type	Description
p	%	A polynomial or series
t	`ExpressionTree`	An expression tree
x	`Symbol`	A name for the variables
port	`TextWriter`	An output port

Description

apply(p, t) returns p as an expression tree, using t as root variable name, while apply(port, p, x) sends p to port using x as root variable name, and returns the output port afterwards.

Example

```
import from Integer, DenseUnivariatePolynomial Integer;

p := term(1, 3) + term(2, 1) - term(1, 0);  -- p = x^3 + 2 x - 1
stdout(map((n:Integer):Integer +-> n + n)(p), -"x");
```

writes

$$2\ x\hat{}\{3\} + 4\ x - 2$$

to the standard stream `stdout`.

Usage

for c in coefficients m repeat { ... }

Signature

coefficients: % → `Generator` R

Parameter	Type	Description
m	%	An element of the module

Returns

Returns a generator that produces all the coefficients of m, including the ones which are 0.

See also

`generator`, `terms`

Usage
monom

Signature
monom: %

Returns
Returns P_1, *i.e.* the term with degree 1 and coefficient 1.

See also
`monomial`

Usage

shift(p, m)
shift!(p, m)

Signature

shift: (%,`Integer`) → %

Parameter	Type	Description
p	%	A polynomial or series
m	`Integer`	The amount to shift

Returns

Returns

$$\sum_{i \geq \max(0,-m)} a_i P_{i+m}$$

where $p = \sum_{i \geq 0} a_i P_i$.

Remarks

When using shift!, the storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other series or polynomials.

Usage

truncate(p, m)
truncate!(p, m)

Signature

truncate: (%,`Integer`) → %

Parameter	Type	Description
p	%	A polynomial or series
m	`Integer`	The truncation order

Returns

Returns the truncation of p at order m, *i.e.*

$$\sum_{i=0}^{m-1} a_i P_i$$

where $p = \sum_{i \geq 0} a_i P_i$.

Remarks

When using truncate!, the storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other series or polynomials.

Usage

import from PrimeFieldUnivariateFactorizer(F, P)

Parameter	Type	Description
F	`PrimeFieldCategory0`	Coefficient ring of the polynomials
P	`UnivariatePolynomialCategory` F	A polynomial ring

Description

PrimeFieldUnivariateFactorizer provides implementations of various univariate factorization algorithms over a prime field.

Exports

`berlekamp`:	P → `List` P	Berlekamp's algorithm
`cantorZassenhaus`:	P → `List` P	Cantor-Zassenhaus algorithm
`factor`:	P → (F, PROD)	Factor (default algorithm)
`factor`:	(P → `List` P) → (P → (F, PROD))	Factor (given algorithm)
`roots`:	(P, `Boolean`) → `Generator` FR F	Roots of a polynomial

where

FR == `FractionalRoot`

PROD == `Product` P

Usage

berlekamp p

Signature

berlekamp: P → List P

Parameter	Type	Description
p	P	The square free and monic polynomial to factor.

Returns

Returns the list of irreducible factors of *p*.

Usage

cantorZassenhaus p

Signature

cantorZassenhaus: P → List P

Parameter	Type	Description
p	P	The square free and monic polynomial to factor.

Returns

Returns the list of irreducible factors of *p*.

Usage

roots(p, sqrfree?)

Signature

roots: (P, `Boolean` → `Generator` `FractionalRoot` F

Parameter	Type	Description
p	P	A polynomial.
sqfree?	`Boolean`	Indicates whether p is squarefree

Returns

Returns the roots of p in F with their multiplicities. Assumes that p is squarefree if $sqrfree?$ is *true*.

Usage

factor p
factor(e)(p)

Signatures

factor: P → (F, Product P)
factor: (P → List P) → (P → (F, Product P))

Parameter	Type	Description
p	P	The polynomial to factor.
e	P → List P	The factorization engine to use.

Returns

Returns $(c, p_1^{e_1} \cdots p_n^{e_n})$ such that each p_i is irreducible, the p_i's have no common factors, and

$$p = c \prod_{i=1}^{n} p_i^{e_i} .$$

Remarks

Uses the factorizer e for factoring the monic squarefree factors of p.

Usage

RationalRootRing: Category

Description

RationalRootRing is the category of gcd domains that export an algorithm for finding the rational roots of univariate polynomials over themselves.

Exports

`Ring`

`integerRoots`:	(P:POL %) → P → `Generator` FR `Integer`	Integer roots
`rationalRoots`:	(P:POL %) → P → `Generator` FR `Integer`	Rational roots

where

FR == `FractionalRoot`
POL == `UnivariatePolynomialCategory0`

Usage

integerRoots(P)(p)
rationalRoots(P)(p)

Signature

integerRoots,rationalRoots: (P: POL %) → P → `Generator RationalRoot`

where

POL == `UnivariatePolynomialCategory0`

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	A polynomial type
p	P	A polynomial

Returns

Return $[(r_1, e_1), \ldots, (r_n, e_n)]$ where the r_i's are the integer or rational roots of p and have multiplicity e_i.

Usage

import from Resultant(R,P)

Parameter	Type	Description
R	`IntegralDomain`	Coefficient ring for the polynomials
P	`UnivariatePolynomialCategory0` R	A polynomial ring

Exports

`lastSPRS`:	(P,P) → P	last non-zero remainder in the SPRS
`extendedLastSPRS`:	(P,P) → (P,P,P)	extended last non-zero remainder in the SPRS
`resultant`:	(P,P) → R	polynomial resultant
`SPRS`:	(P,P) → `List` P	Subresultant Polynomial Remainder Sequence
`subResultantGcd`:	(P,P) → P	GCD

Description

Resultant(R,P) implements polynomial resultant computations.

Usage

lastSPRS(a,b)

Signature

lastSPRS: (P,P) → P

Parameter	Type	Description
a,b	P	Two polynomials

Description

lastSPRS(a,b) returns the last non-zero remainder in the subresultant polynomial remainder sequence of a and b. a and b both should be non-zero and deg(a) should be at least deg(b).

Usage

extendedLastSPRS(a,b)

Signature

extendedLastSPRS: (P,P) → (P,P,P)

Parameter	Type	Description
a, b	P	Two polynomials

Description

extendedLastSPRS(a,b) returns (r, s, t), where r is the last non-zero remainder in the subresultant polynomial remainder sequence of a and b and s, t are polynomials such that $r = sa + tb$. a and b both should be non-zero and $\deg(a)$ should be at least $\deg(b)$.

Usage

resultant(a,b)

Signature

resultant: (P,P) → R

Parameter	Type	Description
a,b	P	Two polynomials

Description

resultant(a,b) returns the resultant of *a* and *b*. *a* and *b* both should be non-zero.

Usage

SPRS(a,b)

Signature

SPRS: (P,P) → `List` P

Parameter	Type	Description
a,b	P	Two polynomials

Description

SPRS(a,b) returns the list of remainders in the subresultant polynomial remainder sequence of a and b. a and b both should be non-zero and deg(a) should be at least deg(b). The list has increasing degree, i.e. the first element in the list is the last remainder in the remainder sequence.

Usage

subResultantGcd(a,b)

Signature

subResultantGcd: (P,P) → P

Parameter	Type	Description
a, b	P	Two polynomials

Description

subResultantGcd(a,b) returns the last non-zero remainder in the subresultant polynomial remainder sequence of either a and b or b and a. If R is a Gcd domain, this is then $\gcd(a, b)$. Also returns $\gcd(a, b)$ if either $a = 0$ or $b = 0$.

Usage

import from SparseUnivariatePolynomial0 R
import from SparseUnivariatePolynomial0(R, x)

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain
x	`Symbol`	The variable name (optional)

Description

SparseUnivariatePolynomial0(R, x) implements free modules over an arbitrary arithmetic system R, with respect to free monoid generated by x. Its elements are assumed to have finite support. The representation is sparse.

Exports

`IndexedFreeModule` (R,Integer)

Usage

import from SparseUnivariatePolynomial1 R
import from SparseUnivariatePolynomial1(R, x)

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain
x	`Symbol`	The variable name (optional)

Description

SparseUnivariatePolynomial1(R, x) implements sparse univariate polynomials with coefficients in R.

Exports

`UnivariatePolynomialAlgebra` R

Usage

import from SparseUnivariatePolynomial R
import from SparseUnivariatePolynomial(R, x)

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain
x	`Symbol`	The variable name (optional)

Description

SparseUnivariatePolynomial(R, x) implements sparse univariate polynomials with coefficients in R.

Exports

`UnivariatePolynomialCategory` R

Usage

import from UnivariateFactorialPolynomial(R, Rx)

Parameter	Type	Description
R	`Ring`	The coefficient domain
Rx	`UnivariatePolynomialCategory` R	A polynomial type over R

Description

UnivariateFactorialPolynomial(R, Rx) implements univariate factorial polynomials with coefficients in R. Those are polynomials with respect to the basis of the descending factorials $(x^{\underline{n}})_{n\geq 0}$, where $x^{\underline{n}} = x(x-1)\dots(x-n+1)$. Rx is used for representing the factorial polynomials, so you can choose between sparse and dense representations.

Exports

`MonogenicAlgebra` R

`coerce`:	Rx → %	Conversion to a factorial polynomial
`expand`:	% → Rx	Conversion from a factorial polynomial
`trailExpand`:	% → (`Integer`, Rx)	Conversion from a factorial polynomial

if R has `CommutativeRing` then
`CommutativeRing`

if R has `IntegralDomain` then
`IntegralDomain`

if R has `RationalRootRing` then

`integerRoots`:	% → `List FractionalRoot Integer`	Integer roots
`rationalRoots`:	% → `List FractionalRoot Integer`	Rational roots

Usage

p::%
coerce p
expand q
(n, h) := trailExpand q

Signatures

coerce:	Rx → %
expand:	% → Rx
trailExpand:	% → (`Integer`, Rx)

Parameter	**Type**	**Description**
p	Rx	A polynomial
q	%	A factorial polynomial

Description

p::% converts p from the power basis $(x^n)_{n\geq 0}$ to the factorial basis $(x^{\underline{n}})_{n\geq 0}$, while expand(q) performs the reverse conversion and trailExpand(q) returns (n, h) such that $q = x^{\underline{n}}h$.

Usage

integerRoots p
rationalRoots p

Signature

integerRoots,rationalRoots: % → `List FractionalRoot Integer`

Parameter	Type	Description
p	%	A factorial polynomial

Returns

Return $[(r_1, e_1), \ldots, (r_n, e_n)]$ where the r_i's are the integer or rational roots of p and have multiplicity e_i.

Usage

UnivariateGcdRing: Category

Description

UnivariateGcdRing is the category of rings which export a gcd algorithm for univariate polynomials over themselves.

Exports

`GcdDomain`

`gcdUP`:	(P:`UnivariatePolynomialCategory0`) → (P, P) → P	Gcd
`gcdUP!`:	(P:`UnivariatePolynomialCategory0`) → (P, P) → P	Gcd
`gcdquoUP`:	(P:`UnivariatePolynomialCategory0`) → (P, P) → (P,P,P)	Gcd

Usage

gcdUP(P)(p, q)
gcdUP!(P)(p, q)

Signature

gcdUP: (P: `UnivariatePolynomialCategory0` %) → (P, P) → P

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	A polynomial type
p,q	P	Polynomials

Returns

Both function return $\gcd(p, q)$. When gcdUP! is used, the storage used by x_1 and x_2 is allowed to be destroyed or reused, so p and q are lost after this call.

Usage

gcdquoUP(P)(p, q)

Signature

gcdquoUP: (P: `UnivariatePolynomialCategory0` %) → (P,P) → (P,P,P)

Parameter	Type	Description
P	`UnivariatePolynomialCategory0` %	A polynomial type
p,q	P	Polynomials

Returns

Returns (g, y, z) such that $g = \gcd(p, q)$, $p = gy$ and $q = gz$.

Usage

import from UnivariateIntegralFactorizer(Z, P)

Parameter	Type	Description
Z	`IntegerCategory`	An integer-like ring
P	`UnivariatePolynomialCategory0` Z	A polynomial type over Z

Description

UnivariateIntegralFactorizer(Z, P) implements a factorizer for polynomials with integer coefficients.

Exports

`factor`:	P → (Z, `Product` P)	Factor
`integerRoots`:	P → `Generator FractionalRoot Integer`	Integer roots
`rationalRoots`:	P → `Generator FractionalRoot Integer`	Rational roots

Usage

factor p

Signature

factor: P → (Z,`Product` P)

Parameter	Type	Description
p	P	A polynomial with integer coefficients

Returns

Returns $(c, p_1^{e_1} \cdots p_n^{e_n})$ such that each p_i is irreducible, the p_i's have no common factors, and

$$p = c \prod_{i=1}^{n} p_i^{e_i} .$$

See also

`squareFree`

Usage

integerRoots p
rationalRoots p

Signature

integerRoots,rationalRoots: P → `Generator FractionalRoot Integer`

Parameter	Type	Description
p	P	A polynomial with integer coefficients

Returns

integerRoots(p) (resp. rationalRoots(p)) return $[(r_1, e_1), \ldots, (r_n, e_n)]$ where the r_i's are the integer (resp. rational) roots of p and have multiplicity e_i.

Usage

import from UnivariateMonomial R
import from UnivariateMonomial(R, x)

Parameter	Type	Description
R	`ExpressionType`	Coefficients of the monomials
	`ArithmeticType`	
x	`Symbol`	The variable name (optional)

Description

This type implements univariate monomials with coefficients in R.

Exports

`ExpressionType`

`apply`:	(%, TREE) → TREE	Applies a monomial to a tree
`coefficient`:	% → R	Extraction of the coefficient
`degree`:	% → `Integer`	The degree of a monomial
`map`:	(R → R) → % → %	Lift a map
`map!`:	(R → R) → % → %	Lift a map
`monomial`:	(R, `Integer`) → %	Creation of a monomial
`setCoefficient!`:	(%, R) → R	In–place coefficient modification
`setDegree!`:	(%, Z) → Z	In-place degree modification

if R has `FiniteCharacteristic` then

`pthPower`:	% → %	Exponentiation to the characteristic
`pthPower!`:	% → %	In–place exponentiation to the characteristic

where

TREE == `ExpressionTree`

Usage

apply(p, t)

Signature

apply: (%, `ExpressionTree`) → `ExpressionTree`

Parameter	Type	Description
p	%	A monomial
t	`ExpressionTree`	An expression tree

Returns

Returns p as an expression tree using t as root variable name.

Usage

coefficient p

Signature

coefficient: % → R

Parameter	Type	Description
p	%	A monomial

Returns

Returns the coefficient of p, *i.e.c* where $p = c\,x^n$.

See also

`degree`, `setCoefficient!`

Usage

degree p

Signature

degree: % → `Integer`

Parameter	Type	Description
p	%	A monomial

Returns

Returns the degree of p, *i.e.*n where $p = c\,x^n$.

See also

`coefficient`

Usage

map f
map! f
map(f)(p)
map!(f)(p)

Signature

map: $(R \to R) \to \% \to \%$

Parameter	Type	Description
f	$R \to R$	A map
p	%	A monomial

Description

map(f)(p) returns $f(a)x^n$ where $p = ax^n$, while map(f) returns the mapping $p \to f(p)$. In both cases, map! does not make a copy of p but modifies it in place.

Usage

monomial(c, n)

Signature

monomial: (R, `Integer`) → %

Parameter	Type	Description
c	R	A scalar
n	`Integer`	An exponent

Returns

Returns the monomial $c\,x^n$.

Usage

pthPower p
pthPower! p

Signature

pthPower: $\% \to \%$

Parameter	Type	Description
p	%	A monomial

Returns

Returns $p^{\text{characteristic}}$.

Remarks

pthPower! does not make a copy of p, which is therefore modified after the call. It is unsafe to use the variable p after the call, unless it has been assigned to the result of the call, as in `p := pthPower! p`.

Usage

setCoefficient!(p, c)

Signature

setCoefficient!: (%, R) → R

Parameter	Type	Description
p	%	A monomial
c	R	A scalar

Description

Sets the coefficient of p to c,*i.e.*changes $p = d\ x^n$ into $c\ x^n$

Returns

Returns the new coefficient c.

See also

`coefficient`

Usage

setDegree!(p, c)

Signature

setDegree!: (%, `Integer`) → `Integer`

Parameter	Type	Description
p	%	A monomial
n	`Integer`	An exponent

Description

Sets the degree of p to n,*i.e.*changes $p = c\ x^m$ into $c\ x^n$

Returns

Returns the new degree n.

See also

`degree`

Usage

UnivariatePolynomialAlgebra R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

UnivariatePolynomialAlgebra is a common category for commutative and noncommutative univariate polynomials with coefficients in an arbitrary arithmetic system R and with respect to the power basis $(x^n)_{n\geq 0}$.

Exports

`MonogenicAlgebra` R

`add!`:	(%,R,Z,%, Z, Z) → %	In-place partial product and sum
`compose`:	(%, %) → %	Compose polynomials
`translate`:	(%, R) → %	Translate a polynomial

if R has `Parsable` then

`Parsable`

where

Z == `Integer`

Usage

add!(p, c, m, q, n, N)

Signature

add!: (%,R,`Integer`,%,`Integer`,`Integer`) → %

Parameter	Type	Description
p	%	A polynomial (to be destroyed)
c	R	A scalar
m	`Integer`	The degree of the monomial to add
q	%	A polynomial to be multiplied by cx^m and added to p
n	`Integer`	A lower threshold
N	`Integer`	An upper treshold

Description

add!(p, c, m, q, n, N) computes all the terms of degree at least n and at most N of

$$p + cx^m q = \sum_{i=0}^{d+m} (a_i + cb_{i-m})x^i \,,$$

where $p = \sum_{i=0}^{d} a_i x^i$ and $q = \sum_{i=0}^{d} b_i x^i$. Note that m is allowed to be negative. For efficiency reasons it is sometimes sufficient to compute some terms of that sum only. All other coefficients of p are not changed.

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other polynomials. Some functions, like `reductum` are not necessarily copying their arguments and can thus create memory aliases.

Usage

compose(p, q)
translate(p, r)

Parameter	Type	Description
p, q	%	Polynomials
r	R	Amount to translate

Returns

compose(p, q) returns

$$p(q) = \sum_{i=0}^{n} a_i q^i$$

where $p = \sum_{i=0}^{n} a_i x^i$, while translate(p, r) returns $p(x - r)$.

Usage

UnivariatePolynomialCategory R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

UnivariatePolynomialCategory is the category of univariate polynomials with coefficients in an arbitrary domain R and with respect to the power basis $(x^n)_{n\geq 0}$.

Exports

`UnivariatePolynomialAlgebra` R

`apply`:	(%, R) → R	Evaluate a polynomial
`apply`:	(%, %) → %	Evaluate a polynomial
`equal?`:	(%, %, %, `Integer`) → `Boolean`	Truncated equality
`Horner`:	(%, R) → (%, R)	Horner division by $x - a$

if R has `CharacteristicZero` then

`ordinaryPoint`:	% → `Integer`	Point where a polynomial is nonzero

if R has `CommutativeRing` then

`DifferentialRing`

`lift`:	(`Derivation` R, %) → `Derivation` %	Extend a derivation
`monicDivide`:	(%, %) → (%, %)	Polynomial division
`monicDivide!`:	(%, %) → (%, %)	Polynomial division
`monicDivideBy`:	% → % → (%, %)	Polynomial division
`monicDivideBy!`:	% → % → (%, %)	Polynomial division
`monicQuotient`:	(%, %) → %	Quotient
`monicQuotient!`:	(%, %) → %	Quotient
`monicQuotientBy`:	% → % → %	Quotient
`monicQuotientBy!`:	% → % → %	Quotient
`monicRemainder`:	(%, %) → %	Remainder
`monicRemainder!`:	(%, %) → %	Remainder
`monicRemainderBy`:	% → % → %	Remainder
`monicRemainderBy!`:	% → % → %	Remainder

if R has `CommutativeRing` and R has `RittRing` then

`integrate`:	% → %	Integration
	(%, `Integer`) → %	

if R has `FactorizationRing` then

`factor`:	% → (R, `Product` %)	Factorisation into irreducibles
`fractionalRoots`:	% → `Generator` `FractionalRoot` R	Roots in the fraction field
`roots`:	% → `Generator` `FractionalRoot` R	Roots in the coefficient ring

if R has `Field` then
`EuclideanDomain`
`sparseMultiple`: (%, `Integer`) → % Multiple in $k[x^n]$

if R has `FiniteField` then
LinearAlgebraRing

if R has `GcdDomain` then
`DecomposableRing`
`GcdDomain`

`squareFree`:	% → (R, `Product` %)	Squarefree factorisation
`squareFreePart`:	% → %	Squarefree part

if R has `GcdDomain` and R has `RationalRootRing` then

`dispersion`:	% → `Integer`	Dispersion
	(%, %) → `Integer`	
`integerDistances`:	% → `List Integer`	Integer spread
	(%, %) → `List Integer`	
`universalBound`:	(%,%) → `List Cross`(%, `Integer`)	Universal bound

if R has `IntegralDomain` then
`IntegralDomain`

`pseudoDivide`:	(%, %) → (%, %)	Polynomial pseudo–division
`pseudoRemainder`:	(%, %) → %	Pseudo–remainder
`pseudoRemainder!`:	(%, %) → %	Pseudo–remainder
`resultant`:	(%, %) → R	Resultant of 2 polynomials

if R has `OrderedArithmeticType` then
`height`: % → R Max norm over all the coefficients

if R has `RationalRootRing` then
`RationalRootRing`

`integerRoots`:	% → `Generator FractionalRoot Integer`	Integer roots
`rationalRoots`:	% → `Generator FractionalRoot Integer`	Rational roots

if R has `Ring` then
`values`: (%, R) → `Generator` R Generate values of a polynomial

if R has `Specializable` then
`Specializable`

Usage

apply(p, a)
apply(p, q)
p a
p q

Signatures

apply: (%, R) → R
apply: (%, %) → %

Parameter	Type	Description
p	%	A polynomial
q	%	A polynomial
a	R	A scalar

Returns

Returns

$$p(a) = \sum_{i=0}^{n} a_i a^i$$

or

$$p(q) = \sum_{i=0}^{n} a_i q^i$$

where $p = \sum_{i=0}^{n} a_i x^i$.

Usage

equal?(a, b, c, n)

Signature

equal?: (%, %, %, `Integer`) → `Boolean`

Parameter	Type	Description
a, b, c	%	Polynomials
n	`Integer`	The order of truncation

Returns

Returns *true* if $a = bc \pmod{x^n}$, *false* otherwise.

Usage

height p

Signature

height: % → R

Parameter	Type	Description
p	%	A polynomial

Returns

Returns

$$||p||_\infty = \max_{i=0}^{n}(|a_i|)$$

where $p = \sum_{i=0}^{n} a_i x^i$.

Usage

Horner(p, a)

Signature

Horner: (%, R) → (%, R)

Parameter	Type	Description
p	%	A polynomial
a	R	A point

Returns

Returns $(q, p(a))$ such that $p = q(x - a) + p(a)$.

Usage

integrate p
integrate(p, n)

Signatures

integrate: % → %
integrate: (%, `Integer`) → %

Parameter	Type	Description
p	%	A polynomial
n	`Integer`	The order of integration

Returns

integrate(p) returns $\int p(x)dx$, while integrate(s, n) returns $\int \ldots \int s(x)dx^n$.

Usage

lift(D, x')

Signature

lift: (`Derivation` R, %) → `Derivation` %

Parameter	Type	Description
D	`Derivation` R	A derivation on R
x'	%	The desired derivative of x

Returns

Returns the unique extension of the derivation D such that $Dx = x'$.

Usage

monicXXX(a, b)
monicXXX!(a, b)
monicXXXBy(b)(a)
monicXXXBy!(b)(a)

Signatures

monicDivide, monicDivide!:	(%, %) → (%, %)
monicDivideBy, monicDivideBy!:	% → % → (%, %)
monicQuotient, monicQuotient!:	(%, %) → %
monicRemainder, monicRemainder!:	(%, %) → %
monicQuotientBy, monicQuotientBy!:	% → % → %
monicRemainderBy, monicRemainderBy!:	% → % → %

Parameter	Type	Description
a	%	A polynomial
b	%	A polynomial whose leading coefficient is a unit in R

Returns

monicRemainder(a, b) returns r such that either $r = 0$ or $\deg(r) < \deg(b)$ or $a \equiv r \pmod{b}$, monicQuotient(a, b) returns q such that $a - bq = 0$ or $\deg(a - bq) < \deg(b)$, and monicDivide(a, b) returns (q, r) such that $a = bq + r$ and either $r = 0$ or $\deg(r) < \deg(b)$. The functions monicDivide!, monicQuotient! and monicRemainder! return the same results but allow the storage used by a to be destroyed or reused. Finally, monicXXXBy(b) returns the map $a \to$ monicXXX(a, b), while monicXXXBy!(b) returns the map $a \to$ monicXXX!(a, b).

Remarks

When using monicXXX!(a,b) or monicXXXBy!(b)(a), the storage used by a is allowed to be destroyed or reused, so a is lost after this call. This may cause a to be destroyed, so do not use this unless a has been locally allocated, and is thus guaranteed not to share space with other polynomials.

See also

`pseudoRemainder`

Usage

ordinaryPoint p

Signature

ordinaryPoint: % → `Integer`

Parameter	Type	Description
p	%	A nonzero polynomial

Returns

Returns an integer n such that $p(n) \neq 0$.

Usage

pseudoDivide(a, b)

Signature

pseudoDivide: (%, %) → (%, %)

Parameter	Type	Description
a	%	A polynomial
b	%	A nonzero polynomial

Returns

Returns (q, r) such that $c^n a = bq + r$ and either $r = 0$ or $\deg(r) < \deg(b)$, where c is the leading coefficient of b and $n = \deg(a) - \deg(b) + 1$.

See also

`pseudoRemainder`

Usage

pseudoRemainder(a, b)
pseudoRemainder!(a, b)

Signature

pseudoRemainder: (%, %) → %

Parameter	Type	Description
a	%	A polynomial
b	%	A nonzero polynomial

Returns

Returns r such that $c^n a = bq + r$ and either $r = 0$ or $\deg(r) < \deg(b)$, where c is the leading coefficient of b and $n = \deg(a) - \deg(b) + 1$.

Remarks

When using pseudoRemainder!(a, b), the storage used by a is allowed to be destroyed or reused, so a is lost after this call. This may cause a to be destroyed, so do not use this unless a has been locally allocated, and is thus guaranteed not to share space with other polynomials.

See also

`pseudoDivide`

Usage

resultant(p, q)

Signature

resultant: (%, %) → R

Parameter	Type	Description
p	%	A polynomial
q	%	A polynomial

Returns

Returns the resultant of p and q.

Usage

sparseMultiple(p, n)

Signature

sparseMultiple: (%, `Integer`) → %

Parameter	Type	Description
p	%	A polynomial
n	`Integer`	A positive integer

Returns

Returns a nonzero polynomial $q = \sum_{i=0}^{m} a_i x^i$ of minimal degree such that $q(x^n)$ is a multiple of $p(x)$.

Usage

squareFree p
squareFreePart p

Signatures

squareFree: % → (R, `Product` %)
squareFreePart: % → %

Parameter	Type	Description
p	%	A polynomial

Description

squareFree(p) returns $(c, p_1^{e_1} \cdots p_n^{e_n})$ such that each p_i is squarefree, the p_i's have no common factors, and

$$p = c \prod_{i=1}^{n} p_i^{e_i},$$

while squareFreePart(p) returns p^* such that p^* is squarefree, $p^* \mid p$ and every irreducible factor of p divides p^*.

Usage

values(p, a)

Signature

values: (%,R) → `Generator` R

Parameter	Type	Description
p	%	A polynomial
a	R	A scalar

Returns

Returns a generator generating the sequence $p(a), p(a+1), p(a+2), \ldots$

Remarks

values uses arrays of differences and can be more efficient that repeated Horner evaluation.

Usage

factor p

Signature

factor: % → (R, `Product` %)

Parameter	Type	Description
p	%	A polynomial

Returns

Returns $(c, p_1^{e_1} \cdots p_n^{e_n})$ such that each p_i is irreducible, the p_i's have no common factors, and

$$p = c \prod_{i=1}^{n} p_i^{e_i} .$$

Usage

fractionalRoots p
roots p

Signature

fractionalRoots,roots: % → `Generator FractionalRoot` %

Parameter	Type	Description
p	%	A polynomial

Returns

Returns a generator that produces all the roots *p* either in the ring or in its fraction field.

Usage

dispersion p
dispersion(p, q)
integerDistances p
integerDistances(p, q)

Signatures

dispersion:	% → `Integer`
dispersion:	(%,%) → `Integer`
integerDistances:	% → `List Integer`
integerDistances:	(%,%) → `List Integer`

Parameter	Type	Description
p	%	A nonzero polynomial
q	%	A nonzero polynomial (optional)

Returns

integerDistances(p, q) returns all the integers $e \in \mathbb{Z}$ such that for each such e there exists α in an algebraic closure of the fraction field of R such that $p(\alpha) = q(\alpha + e) = 0$, while dispersion(p, q) returns -1 if integerDistances(p, q) contains only elements strictly smaller than 0, its maximal nonnegative element otherwise.

Remarks

The parameter q is optional for both functions, its default value being p.

Usage

integerRoots p
rationalRoots p

Signature

integerRoots,rationalRoots: % → `Generator FractionalRoot Integer`

Parameter	Type	Description
p	%	A polynomial

Returns

Return $[(r_1, e_1), \ldots, (r_n, e_n)]$ where the r_i's are the integer or rational roots of p and have multiplicity e_i.

Usage

minIntegerRoot p
maxIntegerRoot p

Signature

minIntegerRoot,maxIntegerRoot: % → `Partial Integer`

Parameter	Type	Description
p	%	A polynomial

Returns

Return *failed* if *p* has no integer root, its smallest (resp. largest) one otherwise.

Usage

universalBound(a, b)

Signature

universalBound: (%,%) → `List Cross`(%, `Integer`)

Parameter	Type	Description
a, b	%	Nonzero polynomials

Returns

Return $[(p_1, e_1), \ldots, (p_n, e_n)]$ such that any polynomial bounded by a and b (in the sense of S.A. Abramov, *Rational solutions of linear difference and q–difference equations with polynomial coefficients*, Proceedings of ISSAC'95) must be a factor of $u = \prod_{i=1}^{n} \prod_{j=0}^{e_i} p_i(x - j)$.

Usage

import from UnivariatePolynomialKaratsuba R

Parameter	Type	Description
R	`CommutativeRing`	The coefficient ring of the polynomials

Description

UnivariatePolynomialKaratsuba R implements Karatsuba multiplication for dense univariate polynomials with coefficients in R.

Exports

`karatsuba!`: (A, A, Z, A, Z, Z, (A, A, Z, A, Z) -¿ ()) → () Karatsuba multiplication

where

A == `PrimitiveArray` R

Z == `MachineInteger`

Usage

import from UnivariatePolynomialSquareFree(R, P)

Parameter	Type	Description
R	`GcdDomain`	Coefficient ring of the polynomials
P	`UnivariatePolynomialCategory0` R	A polynomial ring

Description

UnivariatePolynomialSquareFree provides implementations of various squarefree factorization algorithms.

Exports

`musser`:	P → (R, `Product` P)	Musser's algorithm
`yun`:	P → (R, `Product` P)	Yun's algorithm

Usage

musser p

Signature

musser: P → (R, `Product` P)

Parameter	Type	Description
p	P	The polynomial to factor

Returns

Returns $(c, p_1^{e_1} \cdots p_n^{e_n})$ such that each p_i is squarefree, the p_i's have no common factors, and

$$p = c \prod_{i=1}^{n} p_i^{e_i} .$$

See also

`yun`

Usage

yun p

Signature

yun: P → (R, `Product` P)

Parameter	Type	Description
p	P	The polynomial to factor

Returns

Returns $(c, p_1^{e_1} \cdots p_n^{e_n})$ such that each p_i is squarefree, the p_i's have no common factors, and

$$p = c \prod_{i=1}^{n} p_i^{e_i} .$$

See also

`musser`

Usage

UnivariateTaylorSeriesCategory R: Category

Parameter	Type	Description
R	`ExpressionType` `ArithmeticType`	The coefficient domain

Description

UnivariateTaylorSeriesCategory R is the category of univariate Taylor series with coefficients in R.

Exports

`MonogenicLinearArithmeticType` R

`degree`:	% → `Partial Integer`	upper bound on the degree
`finite?`:	% → `Boolean`	check whether the support is finite
`series`:	`Sequence` R → %	creation of a series

where

UPC == UnivariatePolynomialCategory

if R has `CommutativeRing` then

`differentiate`:	% → %	Differentiation
	(%, `Integer`) → %	
`reciprocal`:	% → `Partial` %	Inverse

if R has `CommutativeRing` and R has `RittRing` then

`integrate`:	% → %	Integration
	(%, `Integer`) → %	

if R has `FloatType` then

`reciprocal`:	% → `Partial` %	Inverse

Usage

degree s
finite? s

Signatures

degree: % → `Partial Integer`
finite?: % → `Boolean`

Parameter	Type	Description
s	%	a series

Returns

finite?(s) returns *true* if s is known to have finite support and *false* otherwise, while degree(s) returns $[n]$ if s is known to have finite support and $\deg(s) \leq n = 0$, *failed* otherwise.

Usage

differentiate s
differentiate(s, n)
integrate s

Signatures

differentiate,integrate: $\% \to \%$
differentiate,integrate: (%, `Integer`) $\to$ %

Parameter	Type	Description
s	%	a series
n	`Integer`	The order of differentiation or integration

Returns

differentiate(s), differentiate(s, n), integrate(s) and integrate(s, n) return respectively ds/dx, $d^n s/dx^n$, $\int s(x)dx$ and $\int \ldots \int s(x)dx^n$.

Usage

series s

Signature

series: `Sequence` R → %

Parameter	Type	Description
s	`Sequence` R	a coefficient sequence

Returns

Returns *s* viewed as a series.

Usage

import from UnivariateTaylorSeriesCategory2Poly(R, RXX, RX)

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	
RXX	`UnivariateTaylorSeriesCategory` R	A series type over R
RX	`UnivariatePolynomialCategory` R	A polynomial type over R

Description

UnivariateTaylorSeriesCategory2Poly(R, RXX, RX) provides conversion tools between polynomials and series.

Exports

`expand`:	R → RX → RXX	series expansion around a point
`truncate`:	(RXX, `Integer`) → RX	truncation of a series

if R has `Field` then

`expandFraction`:	R → `Fraction` RX → (`Integer`, RXX)	series expansion around a point

if R has `FloatType` then

`expandFraction`:	R → `Fraction` RX → (`Integer`, RXX)	series expansion around a point

Usage

expand a
expand(a)(p)

Signature

expand: R → RX → RXX

Parameter	Type	Description
a	R	the expansion point
p	RX	a polynomial

Returns

expand(a)(p) returns the series expansion of *p* around *a*.

See also

`expandFraction`

Usage

expandFraction a
expandFraction(a)(f)

Signature

expandFraction: R → `Fraction` RX → (`Integer`, RXX)

Parameter	Type	Description
a	R	the expansion point
f	`Fraction` RX	a rational function

Returns

expandFraction(a)(f) returns (n, s) where $n \geq 0$ and the series expansion of f around a is $(x-a)^n s(x-a)$. In addition, $s(a) \neq 0$ whenever $n < 0$.

See also

`expand`

Usage

truncate(s, m)

Signature

truncate: (RXX,`Integer`) → RX

Parameter	Type	Description
s	%	a series
m	`Integer`	the truncation order

Returns

Returns the truncation of s at order m, *i.e.*

$$\sum_{n=0}^{m-1} a_n x^n$$

where $s = \sum_{n\geq 0} a_n x^n$.

Usage

import from UnivariateTaylorSeriesNewtonSolver(R, Rx, RxY)
import from UnivariateTaylorSeriesNewtonSolver(R, Rx, RX, RXY)

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	
Rx	`UnivariateTaylorSeriesCategory` R	Series over R
RxY	`UnivariatePolynomialCategory` Rx	Polynomials over Rx
RX	`UnivariatePolynomialCategory` R	Polynomials over R
RXY	`UnivariatePolynomialCategory` RX	Polynomials over RX

Description

UnivariateTaylorSeriesNewtonSolver provides a Newton solver for computing roots in $R[[x]]$ of polynomials in either $R[x, y]$ or $R[[x]][y]$.

Exports

if R has `CommutativeRing` then

`differentiate`:	RxY → RxY	Differentiation
`root`:	(RXY, R) → Rx	Root of a polynomial
	(RxY, R) → Rx	

if R has `FloatType` then

`root`:	(RXY, RXY, R) → Rx	Root of a polynomial
	(RxY, RxY, R) → Rx	

Usage

differentiate p

Signature

differentiate: RxY → RxY

Parameter	Type	Description
p	RxY	a polynomial

Returns

Returns $\frac{dp}{dy}$.

Usage

root(p, s_0)
root(p, p', s_0)

Signatures

root: (RXY, R) → Rx
root: (RxY, R) → Rx
root: (RXY, RXY, R) → Rx
root: (RxY, RxY, R) → Rx

Parameter	Type	Description
p	RXY	a nonzero polynomial
	RxY	
p'	RXY	the derivative of p with respect to y
	RxY	
s_0	R	a simple root of $p(0, y)$

Description

Returns a series $s(x)$ such that $p(x, s(x)) = 0$ and $s(0) = s_0$. The initial value s_0 must satisfy $p(0, s_0) = 0$, and in addition, $\frac{dp}{dy}(0, s_0)$ must be a unit in R.

Remarks

The parameter p' must be given only when R has `FloatType`, since differentiation is not available for polynomials over such rings.

Usage

import from DistributedMultivariatePolynomial0 (R,E)

Parameter	Type	Description
R	`ArithmeticType` `ExpressionType`	The coefficient domain
E	`GeneralExponentCategory` V	The exponent domain

Description

DistributedMultivariatePolynomial0 (R,E) provides an implementation of the free module over R with basis E. Roughly speaking, the elements of DistributedMultivariatePolynomial0 (R,E) are polynomials that cannot be multiplied. Each of these *polynomials* x is coded as a list of term (r, e) with $r \in R, r \neq 0$ and $e \in E$ such that x is the sum of these terms.

Exports

`CopyableType`
`IndexedFreeModule`(R,E)

Usage

import from DistributedMultivariatePolynomial1 (R,V,E)

Parameter	Type	Description
R	`Ring`	The coefficient ring
V	`VariableType`	The variable type
E	`ExponentCategory` V	The exponent domain

Description

DistributedMultivariatePolynomial1 (R,V,E) provides a basic domain for multivariate polynomials with coefficients in R and variables in V. The monomials are coded by means of exponents from E. Polynomials are represented in a sparse and distributed way. This means that each polynomial x is coded as a list of term (r, e) with $r \in R, r \neq 0$ and $e \in E$ such that x is the sum of these terms.

Exports

`FiniteAbelianMonoidRing0`(R,V,E)

Usage

import from DirectProduct (n,T)

Parameter	Type	Description
n	`MachineInteger`	The dimension of the direct product
T	`ExpressionType`	The type of the factors

Description

DirectProduct (n,T) provides n-ary direct products of elements from T. Such a product is represented by a primitive array of elements from T with size n. The indices of its components are in the range $0 \cdots n-1$.

Exports

`DirectProductCategory` (n,T)

Usage

DirectProductCategory (dim, T): Category

Parameter	Type	Description
dim	`MachineInteger`	The length of a direct product
T	`ExpressionType`	The type of each factor

Description

DirectProductCategory (dim, T) is the category of cartesian products of *dim* copies of *T*. Hence an elements of a domain of this category is a a (direct) product (or tuple) of elements from *T* with length *dim*. The components of such a tuple are indexed from `firstIndex` to `lastIndex`. Thus *dim* is $lastIndex - firstIndex + 1$. This category is essentially designed to support the implementation of multivariate monomials involving at most $dim - 1$ (if one component is used for storing the total degree) or *dim* (if not) variables.

Exports

`CopyableType`
`LinearStructureType` T
`ExpressionType`

`bracket`:	`Tuple` T → %	Conversion of a tuple whose length is *dim*
`lastIndex`:	`MachineInteger`	The biggest index of a direct product.
`generator`:	% → `Generator` T	The factors of a direct product.
`map`:	((T → T) → T) → (%, %) → %	Componentwise mapping.
`map`:	(T → T) → % → %	Mapping
`map!`:	(T → T) → % → %	Mapping that may modify its second arg

Usage

ExponentCategory V: Category

Parameter	Type	Description
V	`VariableType`	The domain of variables

Description

ExponentCategory V provides multivariate monomials (i.e. products of variables from V) looked as an addtive ordered monoid (with cancellation) by associating to every product of variables its sequence of degrees.

Exports

`GeneralExponentCategory`

`exponent`:	V → %	The exponent of a variable
`exponent`:	(V, Z) → %	The exponent of a power of a variable
`exponent`:	`Generator Cross` (V, Z) → %	The exponent of a power product
`exponent`:	(`List` V, `List` Z) → %	The exponent of a power product
`terms`:	% → `Generator Cross` (V, Z)	Inverse map of `exponent`
`mainVariable`:	% → `Partial` V	The biggest variable, if any
`variables`:	% → `List` V	The list of variables of a monomial
`degree`:	(%, V) → Z	The degree w.r.t. a variable
`totalDegree`:	% → Z	The sum of the degrees of an exponent
`totalDegree`:	(%, `List` V) → Z	The sum of the degrees w.r.t. a list
`gcd`:	(%, %) → %	Monomial gcd
`lcm`:	(%, %) → %	Monomial lcm
`syzygy`:	(%, %) → %	Cofactors w.r.t. lcm

where

Z == `Integer`

Usage

FiniteAbelianMonoidRing0 (R,V,E): Category

Parameter	Type	Description
R	`ArithmeticType`	The coefficient domain
	`ExpressionType`	
V	`VariableType`	The variable type
V	`ExponentCategory` V	The exponent domain

Description

FiniteAbelianMonoidRing0 (R,V,E) is a model for *distributed* polynomials. Such polynomials are looked as sums of terms $c_i m_i$ where the r_i are coefficients from R and the m_i are monomials from the free abelian monoid generated by V. Moreover these monomials are coded by means of exponents from E. These exponents commute with each other and with the coefficients. However the coefficients may or may not commute.

Exports

`IndexedFreeAlgebra`(R,E)
`PolynomialRing0`(R,V)

`add!`:	(R, %, R, E, %) → %	`add!`(c_1, x, c_2, e, y) is $c_1 x$+`term`$(c_2, e)y$ and may modify x
`map`:	(E → E) → % → %	`map`$(f)(x)$ maps f on the exponents of x
`map!`:	(E → E) → % → %	`map`$(f)(x)$ returns `map`$(f)(x)$ and may modify x

Usage

FiniteVariableType: Category

Description

FiniteVariableType is a category for multivariate polynomial variables that belong to a finite set. Hence, a domain of this category implements a finite set of ordered variables. The operations `variable` and `index` define a one-to-one map from % onto a range of (machine) integers.

Exports

`VariableType`

`variable`:	`MachineInteger` $\to$ %	Retuns the n-th variable of the type, if any
`index`:	% $\to$ `MachineInteger`	Retuns the associated machine integer
`#`:	`MachineInteger`	The number of elements of the type
`max`:	%	The greatest element of the type
`min`:	%	The smallest element of the type
`minToMax`:	`List` %	The elements of the type sorted in increasing order
`maxToMin`:	`List` %	The elements of the type sorted in decreasing order

Usage

variable i

Signature

variable: `MachineInteger` → %

Parameter	Type	Description
i	`MachineInteger`	an index

Returns

Returns the i-th variable of the type if i is the range of indices associated with the type. Otherwise, an error or an exception is raised.

Usage

GeneralExponentCategory: Category

Description

GeneralExponentCategory is the category of monomials looked as an additive ordered monoid with a cancellation function and endowed with an order such that the addition is consistent with this order. By consistent addition we mean that 'a ¿= b' implies 'a + c ¿= b + c'. By a cancellation function we mean an operation 'f' such that 'f(a,b)' is either 'failed' or 'c' such that 'a = b + c' holds. Here are two common examples of this operation. For integers 'f(a,b)' is 'a-b' if 'a¿=b' and 'failed' otherwise. For multivariate monomials 'f(a,b)' is 'c' if 'a=b*c' and 'failed' no such 'c' exists.

Exports

`ExpressionType`
`TotallyOrderedType`

0:	%	zero
+:	(%, %) → %	sum
`add!`:	(%, %) → %	in-place sum
`cancel`:	(%, %) → %	cancellation
`cancel?`:	(%, %) → `Boolean`	cancellation
`cancelIfCan`:	(%, %) → `Partial` %	cancellation
`times`:	(`Integer`, %) → %	product by an integer
`zero?`:	% → `Boolean`	test for 0

Remarks

GeneralExponentCategory is meant to implement efficient monomial arithmetic. In particular, for multivariate monomials with a finite set of variables it is meant to code exponents with primitive arrays of machine integers. Hence we cannot claim that every exponent domain belongs to `AbelianMonoid`. Since we cannot subtract any exponent to every other, we cannot claim that every exponent domain belongs to `AdditiveType` neither, which explains the need for this category.

Usage

cancel(x,y)
cancel?(x,y)
cancelIfCan(x,y)

Signatures

cancel:	(%, %) → %
cancel?:	(%, %) → `Boolean`
cancelIfCan:	(%, %) → `Partial` %

Parameter	Type	Description
x,y	%	Elements of the type

Description

cancelIfCan(x,y) returns z such that $z + y = x$ if there exist such a z in the type viewed as a monoid only, *failed* otherwise, while cancel?(x,y)' returns whether cancelIfCan(x,y) would fail, and cancel(x,y) returns z such that $z + y = x$, assuming that cancelIfCan(x,y) would not fail.

Usage

times(n, x)

Signature

times: (`Integer`, %) → %

Parameter	Type	Description
n	`Integer`	An integer
x	%	An element of the type

Returns

Returns the product nx.

Usage

IntegerExponentVectorCategory V: Category

Parameter	Type	Description
V	`FiniteVariableType`	The type of variables

Description

IntegerExponentVectorCategory V is a category for the exponents of the monomials (or power products) generated by the finite set of variables V and coded as direct products of positive integers.

Exports

`CopyableType`
`ExponentCategory` V
`HashType`

`free!`:	$\% \to ()$	Asserts that the input will no longer be used
`exponent`:	`Tuple Integer` $\to \%$	Creation from a tuple
`exponent`:	`PrimitiveArray Integer` $\to \%$	Creation from a primitive array Degrees start at slot 1 Slot 0 must be the total degree
`parray`:	$\% \to$ `PrimitiveArray Integer`	Inverse mapping of `exponent`

Usage

import from MachineIntegerDegreeLexicographicalExponent V

Parameter	Type	Description
V	`FiniteVariableType`	The type of the variables

Description

MachineIntegerDegreeLexicographicalExponent V implements the exponents of the monomials (or power products) generated by the finite set of variables V. Such an exponent is represented by a primtive array of machine integers with length $dim = n + 1$ where n is the number of elements in V. Given e in % and i in $1 \cdots n$ the degree of e w.r.t. the i-th variable of V is stored in slot i. Slot 0 is used to store the total degree of e, that is the sum of the content of all the other slots. An exponent a is greater than an exponent b if a is greater than b w.r.t. the degree-lexicographical ordering induced by V (by comparing a_i and b_i for i running from 1 to n, after comparing the total degrees of a and b).

Exports

`MachineIntegerExponentVectorCategory` V

Usage

import from MachineIntegerDegreeReverseLexicographicalExponent V

Parameter	Type	Description
V	`FiniteVariableType`	The type of the variables

Description

MachineIntegerDegreeReverseLexicographicalExponent V implements the exponents of the monomials (or power products) generated by the finite set of variables V. Such an exponent is represented by a primtive array of machine integers with length $dim = n + 1$ where n is the number of elements in V. Given e in % and i in $1 \cdots n$ the degree of e w.r.t. the i-th variable of V is stored in slot i. Slot 0 is used to store the total degree of e, that is the sum of the content of all the other slots. An exponent a is greater than an exponent b if a is greater than b w.r.t. the degree-reverse-lexicographical ordering induced by V (by comparing a_i and b_i for i running from n to 1, after comparing the total degrees of a and b).

Exports

`MachineIntegerExponentVectorCategory` V

Usage

MachineIntegerExponentVectorCategory V: Category

Parameter	Type	Description
V	`FiniteVariableType`	The type of variables

Description

MachineIntegerExponentVectorCategory V is a category for the exponents of the monomials (or power products) generated by the finite set of variables V and coded as direct products of positive machine integers.

Exports

`CopyableType`
`ExponentCategory` V
`HashType`

`free!`:	% → ()	Asserts that the input will no longer be used
`exponent`:	`Tuple` I → %	Creation from a tuple
`exponent`:	`PrimitiveArray` I → %	Creation from a primitive array Degrees start at slot 1 Slot 0 must be the total degree
`parray`:	% → `PrimitiveArray` I	Inverse mapping of `exponent`

where

I == `MachineInteger`

Usage

import from MachineIntegerLexicographicalExponent V

Parameter	Type	Description
V	`FiniteVariableType`	The type of the variables

Description

MachineIntegerLexicographicalExponent V implements the exponents of the monomials (or power products) generated by the finite set of variables V. Such an exponent is represented by a primtive array of machine integers with length $dim = n + 1$ where n is the number of elements in V. Given e in % and i in $1 \cdots n$ the degree of e w.r.t. the i-th variable of V is stored in slot i. Slot 0 is used to store the total degree of e, that is the sum of the content of all the other slots. An exponent a is greater than an exponent b if a is greater than b w.r.t. the lexicographical ordering induced by V (by comparing a_i and b_i for i running from 1 to n).

Exports

`MachineIntegerExponentVectorCategory` V

OrderedSymbol

Usage

import from OrderedSymbol

Description

OrderedSymbol implements symbols as a type of variables.

Exports

`VariableType`

`orderedSymbol`: `String` → % Conversion to an element of the type.

Usage

import from OrderedVariableList t

Parameter	Type	Description
t	`List Symbol`	The symbols defining the variables of the type

Description

OrderedVariableList t implements the finite set of ordered variables given by t. This set has n elements numbered from to 1 to n, where n is the size of the t. For $i = 1 \cdots n$ the i-th variable of the type is `variable` i and uses the output form of the i-th item in t. For $i, j = 1 \cdots n$ `variable` $i >$ `variable` j holds iff $i < j$ holds. Elements of OrderedVariableList t are internally represented as machine integers. The input list t may contain duplicates since t is only used for output forms matter. Operations from `ExpressionType`, `Parsable`, `HashType` and `SerializableType` are taken from `MachineInteger`.

Exports

`FiniteVariableType`

Usage

import from OrderedVariableTuple t

Parameter	Type	Description
t	`Tuple Symbol`	The symbols defining the variables of the type

Description

OrderedVariableTuple t implements the finite set of ordered variables given by t.

OrderedVariableTuple t is implemented as `OrderedVariableList` l where l is the list of items in t in the same order.

Exports

`FiniteVariableType`

Usage

PolynomialRing (R,V): Category

Parameter	Type	Description
R	`ArithmeticType`	The coefficient domain
	`ExpressionType`	
V	`TotallyOrderedType`	The variable domain
	`ExpressionType`	

Description

PolynomialRing (R,V) is a category which inherits from `PolynomialRing0`(R,V) and exports some additionnal operations related to differentiation, evaluation and polynomial type conversion.

Exports

`PolynomialRing0`(R,V)

`leadingCoefficient`:	(%, V) → %	Leading coefficient w.r.t. a variable
`reductum`:	(%, V) → %	Reductum w.r.t. a variable
`combine`:	(%,V) → GEN PZ	`combine`(x, v) generates x as a univ. w.r.t. v Pairs are sorted by decr. degree w.r.t. v
`combine`:	% → GEN PZ	`combine`(x) is `combine`$(x$,`mainVariable`$x)$
`eval`:	(%, V, R) → %	`eval`(x, v, r) evaluates x at $v = r$
`eval`:	(%, V, %) → %	`eval`(x, v, y) evaluates x at $v = y$

where

GEN == `Generator`
PZ == `Cross`(%, `Integer`)

if R has `CommutativeRing` then

`differentiate`:	(%, V) → %	Differentiation w.r.t. a variable
`differentiate`:	(%, V, `Integer`) → %	Iterated differentiation w.r.t. a variable

if R has `CommutativeRing` then
`CommutativeRing`

if R has `IntegralDomain` then
`IntegralDomain`

if R has `CharacteristicZero` then
`CharacteristicZero`

if R has `FiniteCharacteristic` then
`FiniteCharacteristic`

if R has `Parsable` and em V has `Parsable` then
`Parsable`

Usage

PolynomialRing0 (R,V): Category

Parameter	Type	Description
R	`ArithmeticType`	The coefficient domain
	`ExpressionType`	
V	`TotallyOrderedType`	The variable domain
	`ExpressionType`	

Description

PolynomialRing0 (R,V) is the category of the domains that implement a polynomial ring with coefficients in R and variables in V. If V is a finite set $\{v_1, \ldots, v_l\}$ this polynomial ring is just $R[v_1, \ldots, v_l]$. If V is finite or not, the set of monomials is the free abelian monoid E generated by V. Moreover the default total ordering endowing E is the the lexicographical one induced by V. Observe that the domain V is not assumed to satisfy `VariableType`. In fact, only weaker conditions are required. Hence it is possible to use any totally ordered set as a domain of variables. For instance a field of algebraic numbers. Observe that PolynomialRing0 (R,V) provides essentially operations related to the structure of a free algebra. For more sophisticated operations (differentiation, evaluation, ...) see the category constructor `PolynomialRing`.

Exports

`CopyableType`
`FreeAlgebra` R

`ground?`:	% → `Boolean`	Membership test to the coefficient ring
`coerce`:	V → %	Conversion of a variable to a polynomial
`variable?`:	% → `Boolean`	Membership test to the variable domain
`variable`:	% → V	Conversion of a polynomial to a variable
`variables`:	% → L V	The variables occuring in a polynomial
`mainVariable`:	% → V	Greatest variable occuring in a polynomial
`univariate?`:	% → `Boolean`	True iff a polynomial has a unique variable
`degree`:	(%, V) → Z	Degree of a polynomial w.r.t. a variable
`degrees`:	% → GEN VZ	`degree`(x, v) for $x \in$ `variables`(v)
`totalDegree`:	% → Z	Greatest total degree among all the monomials
`coefficient`:	(%, V, Z) → %	`coefficient`(x, v, n) is the coeff. of x w.r.t. v^n
`coefficient`:	(%, L V, L Z) → %	Coefficient w.r.t. a power product
`term`:	(R, V, Z) → %	`term`(r, v, n) returns $r\, v^n$ provided $n \geq 0$
`term`:	(R, GEN VZ)) → %	Term from a power product and a coeff.
`term`:	(R, L V, L Z) → %	Term from a power product and a coeff.
`variableProduct`:	% → GEN VZ	`variableProduct` x is g s.t. `term`$(1, g)$ is x
`times`:	(%, R, V , Z) → %	`times`(x, r, v, n) is $x\,(r\, v^n)$
`times!`:	(%, R, V , Z) → %	`times`(x, r, v, n) is $x\,(r\, v^n)$ and may modify x

where

GEN == `Generator`
L == `List`
Z == `Integer`
VZ == `Cross` (V, `Integer`)

Usage

RecursiveMultivariatePolynomialCategory0 (R,V): Category

Parameter	Type	Description
R	`ArithmeticType` `ExpressionType`	The coefficient domain
V	`TotallyOrderedType` `ExpressionType`	The variable domain

Description

RecursiveMultivariatePolynomialCategory0 (R,V) extends `PolynomialRing`(R,V) with some additional operations related to the recurive vision of a multivariate polynomial (as a univariate polynomial w.r.t. its main variable).

Exports

`PolynomialRing`(R,V)

`variables`:	% → `SortedSetV`	The sorted set of variables of a polynomial
`mvar`:	% → V	Greatest variable of a polynomial
`mdeg`:	% → `Integer`	Degree w.r.t. greatest variable
`rank`:	% → %	Leading monomial as univariate w.r.t. greatest variable
`init`:	% → %	Leading coefficient as univariate w.r.t. greatest variable
`tail`:	% → %	Reductum as univariate w.r.t. greatest variable
`head`:	% → %	Leading term as univariate w.r.t. greatest variable

VariableType

Usage

VariableType: Category

Description

VariableType is a category for multivariate polynomial variables looked as symbols with additionnal properties such as a total order.

Exports

`TotallyOrderedType`
`ExpressionType`
`Parsable`
`SerializableType`
`HashType`

`variable`:	`Symbol` → `Partial` %	Associated variable, if any
`symbol`:	% → `Symbol`	Associated symbol

Usage

import from ExpressionTree

Description

ExpressionTree is a type whose elements are expression trees.

Exports

`OutputType`
`PrimitiveType`

`aldor`:	(TEXT, %) → TEXT	Conversion to $A^{\#}$code
`apply`:	(OP, `List` %) → %	Apply an operator to arguments
`apply`:	(OP, `Tuple` %) → %	Apply an operator to arguments
`arguments`:	% → `List` %	Take the arguments of the root
`axiom`:	(TEXT, %) → TEXT	Conversion to Axiom code
`C`:	(TEXT, %) → TEXT	Conversion to C code
`extree`:	`ExpressionTreeLeaf` → %	Conversion to a tree
`fortran`:	TEXT, %) → TEXT	Conversion to FORTRAN code
`is?`:	(%, OP) → `Boolean`	Test for a specific operator
`leaf`:	% → `ExpressionTreeLeaf`	Conversion to a leaf
`leaf?`:	% → `Boolean`	Test whether tree is a leaf
`lisp`:	TEXT, %) → TEXT	Conversion to Lisp code
`maple`:	(TEXT, %) → TEXT	Conversion to Maple code
`operator`:	% → OP	Take the root operator
`tex`:	(TEXT, %) → TEXT	Conversion to LaTeX
`texParen?`:	(`MachineInteger`, %) → `Boolean`	Check whether to parenthetize

where

TEXT == `TextWriter`
OP == `ExpressionTreeOperator`

Usage

format(p, t)

Signature

aldor,axiom,C,fortran,lisp,maple,tex: (`TextWriter`, %) → `TextWriter`

Parameter	Type	Description
p	`TextWriter`	The port to write to
t	%	An expression tree

Description

Writes to p the expression corresponding to the tree t in the requested format.

Usage

apply(op, $t_1, \ldots, t_n$)
apply(op, $[t_1, \ldots, t_n]$)
op($t_1, \ldots, t_n$)
op $[t_1, \ldots, t_n]$

Signatures

apply: `List` % → %
apply: `Tuple` % → %

Parameter	Type	Description
op	`ExpressionTreeOperator`	An operator
t_i	%	Expression trees

Returns

Returns the tree whose root is *op*, with arguments $t_1, \ldots, t_n$.

Usage

arguments t

Signature

arguments: % → `List` %

Parameter	Type	Description
t	%	An expression tree

Returns

Returns the list of arguments of the root operator of t, which must not be a leaf.

See also

`operator`

Usage

extree a

Signature

extree: `ExpressionTreeLeaf` → %

Parameter	Type	Description
a	`ExpressionTreeLeaf`	A leaf

Returns

extree a returns *a* as an expression tree.

Usage
is?(t, op)

Signature
is?: (%, `ExpressionTreeOperator`) → `Boolean`

Parameter	Type	Description
t	%	An expression tree
op	`ExpressionTreeOperator`	An operator

Returns
is?(t, op) returns *true* if t is of the form op(args), *false* otherwise.

Usage

leaf t
leaf? t

Signatures

leaf: % → `ExpressionTreeLeaf`
leaf?: % → `Boolean`

Parameter	Type	Description
t	%	An expression tree

Returns

leaf a returns *a* as a leaf is *a* is a leaf. leaf? a returns *true* if a is a leaf, *false* otherwise.

Usage

negate t

Signature

negate: $\% \to \%$

Parameter	Type	Description
t	%	An expression tree

Returns

Returns the leaf $-t$ if t is a numerical leaf with $t < 0$, and returns s if t is of the form $(-s)$ for some tree s. t must be of one of the above 2 forms.

See also

`negative?`

Usage

negative? t

Signature

negative?: % → `Boolean`

Parameter	Type	Description
t	%	An expression tree

Returns

Returns *true* if either t is a numerical leaf and $t < 0$, or if t is of the form $(-s)$ for some tree s, *false* otherwise.

See also

`negate`

Usage

operator t

Signature

operator: % → `ExpressionTreeOperator`

Parameter	Type	Description
t	%	An expression tree

Returns

Returns the root operator of t, which must not be a leaf.

See also

`arguments`

Usage

texParen?(prec, t)

Signature

texParen?: (`MachineInteger`, %) → `Boolean`

Parameter	Type	Description
prec	`MachineInteger`	An operator precendence.
t	%	An expression tree

Returns

Returns *true* if *t* should be parenthetized when appearing as argument of an operator of precedence *prec*, *false* otherwise.

Usage

import from ExpressionTreeAnd

Description

ExpressionTreeAnd is the *logical and* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **C**, **fortran**.

Usage

import from ExpressionTreeAssign

Description

ExpressionTreeAssignis the assignment operator for expression trees.

Exports

`ExpressionTreeOperator`

ExpressionTreeBigO

Usage
import from ExpressionTreeBigO

Description
ExpressionTreeBigO is the $\mathcal{O}$ operator for expression trees.

Exports
`ExpressionTreeOperator`

Usage

import from ExpressionTreeCase

Description

ExpressionTreeCase is the *multi conditional* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **axiom**, **C**, **fortran**, **maple**.

Usage

import from ExpressionTreeExpt

Description

ExpressionTreeExpt is the exponentiation operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreeEqual

Description

ExpressionTreeEqual is the *equal* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **C**, **fortran**.

Usage

import from ExpressionTreeExpt

Description

ExpressionTreeExpt is the exponentiation operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreeFactorial

Description

ExpressionTreeFactorial is the generalized factorial operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreeGreaterEqual

Description

ExpressionTreeGreaterEqual is the *greater or equal* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **C**, **fortran**, **lisp**.

Usage

import from ExpressionTreeGreaterThan

Description

ExpressionTreeGreaterThan is the *greater than* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **C**, **fortran**, **lisp**.

Usage

import from ExpressionTreeIf

Description

ExpressionTreeIf is the *conditional* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following function is not yet implemented and produces dummy results : **fortran**.

Usage

import from ExpressionTreeLeaf

Description

ExpressionTreeLeaf is a type whose elements are the leafs (atoms) of expression trees. It provides conversions to and from the basic atomic types.

Exports

`OutputType`
`PrimitiveType`

`aldor`:	(TEXT, %) → TEXT	Conversion to ALDOR code
`axiom`:	(TEXT, %) → TEXT	Conversion to Axiom code
`boolean`:	% → `Boolean`	Conversion to a boolean
`boolean?`:	% → `Boolean`	Test for a boolean
`C`:	(TEXT, %) → TEXT	Conversion to C code
`doubleFloat`:	% → `DoubleFloat`	Conversion to a double precision float
`doubleFloat?`:	% → `Boolean`	Test for a double precision float
`float`:	% → `Float`	Conversion to a software big float
`float?`:	% → `Boolean`	Test for a software big float
`fortran`:	TEXT, %) → TEXT	Conversion to FORTRAN code
`integer`:	% → `Integer`	Conversion to a software big integer
`integer?`:	% → `Boolean`	Test for a software big integer
`leaf`:	`Boolean` → %	Conversion to a leaf
`leaf`:	`DoubleFloat` → %	Conversion to a leaf
`leaf`:	`MachineInteger` → %	Conversion to a leaf
`leaf`:	`Integer` → %	Conversion to a leaf
`leaf`:	`SingleFloat` → %	Conversion to a leaf
`leaf`:	`String` → %	Conversion to a leaf
`leaf`:	`Symbol` → %	Conversion to a leaf
`lisp`:	(TEXT, %) → TEXT	Conversion to Lisp code
`singleFloat`:	% → `SingleFloat`	Conversion to a single precision float
`singleFloat?`:	% → `Boolean`	Test for a single precision float
`machineInteger`:	% → `MachineInteger`	Conversion to a machine integer
`machineInteger?`:	% → `Boolean`	Test for a machine integer
`maple`:	(TEXT, %) → TEXT	Conversion to Maple code
`string`:	% → `String`	Conversion to a string
`string?`:	% → `Boolean`	Test for a string
`symbol`:	% → `Symbol`	Conversion to a symbol
`symbol?`:	% → `Boolean`	Test for a symbol
`tex`:	(TEXT, %) → TEXT	Conversion to LaTeX
`texParen?`:	% → `Boolean`	Check whether to parenthetize

where

TEXT == `TextWriter`

Usage

format(p, a)

Signature

aldor,axiom,C,fortran,lisp,maple,tex: (`TextWriter`, %) → `TextWriter`

Parameter	Type	Description
p	`TextWriter`	The port to write to
a	%	A leaf

Description

Writes to p the expression corresponding to the leaf a in the requested format.

Usage

boolean a
boolean? a

Signatures

boolean: % → `Boolean`
boolean?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

boolean a returns the value of *a* as a `Boolean` if that is the type of *a*.
boolean? a returns *true* if a is a `Boolean`, *false* otherwise.

Usage

doubleFloat a
doubleFloat? a

Signatures

doubleFloat: % → `DoubleFloat`
doubleFloat?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

doubleFloat a returns the value of *a* as a `DoubleFloat` if that is the type of *a*.
doubleFloat? a returns *true* if a is a `DoubleFloat`, *false* otherwise.

Usage

float a
float? a

Signatures

float: % → `Float`
float?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

float a returns the value of *a* as a `Float` if that is the type of *a*.
float? a returns *true* if a is a `Float`, *false* otherwise.

Usage

integer a
integer? a

Signatures

integer: % → `Integer`
integer?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

integer a returns the value of *a* as an `Integer` if that is the type of *a*.
integer? a returns *true* if a is an `Integer`, *false* otherwise.

Usage

leaf a

Signatures

leaf: `Boolean` → %
leaf: `DoubleFloat` → %
leaf: `Integer` → %
leaf: `Float` → %
leaf: `MachineInteger` → %
leaf: `SingleFloat` → %
leaf: `String` → %
leaf: `Symbol` → %

Parameter	Type	Description
a	`Boolean`	A constant
	`DoubleFloat`	
	`Float`	
	`Integer`	
	`MachineInteger`	
	`SingleFloat`	
	`String`	
	`Symbol`	

Returns

leaf a returns *a* as a leaf.

Remarks

A string leaf prints with quotes, and should be used for string constants, while a symbol leaf prints without quotes, and should be used for names.

Usage

negate a

Signature

negate: $\% \to \%$

Parameter	Type	Description
a	$\%$	A leaf

Returns

Returns the leaf $-a$ if a is a numerical leaf, a otherwise.

See also

`negative?`

Usage

negative? a

Signature

negative?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

Returns *true* if a is a numerical leaf and $a < 0$, *false* otherwise.

See also

`negate`

Usage

machineInteger a
machineInteger? a

Signatures

machineInteger: % → `MachineInteger`
machineInteger?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

machineInteger a returns the value of *a* as a `MachineInteger` if that is the type of *a*.
machineInteger? a returns *true* if a is a `MachineInteger`, *false* otherwise.

Usage

singleFloat a
singleFloat? a

Signatures

singleFloat: % → `SingleFloat`
singleFloat?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

singleFloat a returns the value of *a* as a `SingleFloat` if that is the type of *a*.
singleFloat? a returns *true* if a is a `SingleFloat`, *false* otherwise.

Usage

string a
string? a

Signatures

string: % → `String`
string?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

string a returns the value of *a* as a `String` if that is the type of *a*.
string? a returns *true* if a is a `String`, *false* otherwise.

Usage

symbol a
symbol? a

Signatures

symbol: % → `Symbol`
symbol?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

symbol a returns the value of *a* as a `Symbol` if that is the type of *a*.
symbol? a returns *true* if a is a `Symbol`, *false* otherwise.

Usage

texParen? a

Signature

texParen?: % → `Boolean`

Parameter	Type	Description
a	%	A leaf

Returns

Returns *true* if the leaf *a* should be parenthetized, *false* otherwise.

Usage

import from ExpressionTreeLessEqual

Description

ExpressionTreeLessEqual is the *less or equal* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **C**, **fortran**, **lisp**.

Usage

import from ExpressionTreeLessThan

Description

ExpressionTreeLessThan is the *less than* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **C**, **fortran**, **lisp**.

Usage

import from ExpressionTreeLispList

Description

ExpressionTreeLispList is the lisp list operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following function is not yet implemented and produces dummy results : **fortran**.

Usage

import from ExpressionTreeList

Description

ExpressionTreeList is the list operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following function is not yet implemented and produces dummy results : **fortran**.

Usage

import from ExpressionTreeMatrix

Description

ExpressionTreeMatrix is the matrix operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following function is not yet implemented and produces dummy results : **fortran**.

Usage

import from ExpressionTreeMinus

Description

ExpressionTreeMinus is the unary/binary minus operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreeNotEqual

Description

ExpressionTreeNotEqual is the *not equal* operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following functions are not yet implemented and produce dummy results : **C**, **fortran**.

Usage

ExpressionTreeOperator: Category

Description

ExpressionTreeOperator is the category of operators for expression trees.

Exports

`aldor`:	(TEXT, `List` TREE) → TEXT	Conversion to $A^{\#}$code
`arity`:	`MachineInteger`	Number of arguments
`axiom`:	(TEXT, `List` TREE) → TEXT	Conversion to Axiom code
`C`:	(TEXT, `List` TREE) → TEXT	Conversion to C code
`fortran`:	(TEXT, `List` TREE) → TEXT	Conversion to FORTRAN code
`lisp`:	(TEXT, `List` TREE) → TEXT	Conversion to Lisp code
`maple`:	(TEXT, `List` TREE) → TEXT	Conversion to Maple code
`name`:	`Symbol`	Operator name
`tex`:	(TEXT, `List` TREE) → TEXT	Conversion to LaTeX
`texParen?`:	`MachineInteger` → `Boolean`	Check whether to parenthetize
`uniqueId`:	`MachineInteger`	A unique key per operator

where

TEXT == `TextWriter`
TREE == `ExpressionTree`

Usage

format(p, $[t_1, \ldots, t_n]$)

Signature

aldor,axiom,C,fortran,lisp,maple,tex: (`TextWriter`, `List ExpressionTree`) → `TextWriter`

Parameter	Type	Description
p	`TextWriter`	The port to write to
t_i	`ExpressionTree`	The arguments of the operator

Description

Writes to p the expression corresponding to this operator applied to the arguments $(t_1, \ldots, t_n)$ in the requested format.

Usage

arity

Signature

arity: → `MachineInteger`

Returns

Returns -1 if this operator can be applied to any number of arguments, $n \geq 0$ if this operator can be applied to exactly n arguments.

Usage
name

Signature
name: → `Symbol`

Returns
Returns the name of this operator.

Usage

texParen? prec

Signature

texParen?: `MachineInteger` → `Boolean`

Parameter	Type	Description
prec	`MachineInteger`	An operator precendence.

Returns

Returns *true* if an expression tree with this operator as root should be parenthetized when appearing as argument of an operator of precedence *prec*, *false* otherwise.

Usage
uniqueId

Signature
uniqueId: → `MachineInteger`

Returns
Returns a integer key which is associated to this operator only. This is used for testing whether two operators are equal.

Usage

import from ExpressionTreeOperatorTools

Description

ExpressionTreeOperatorTools provides utilities that make it simpler to write new operator types.

Exports

`infix`: (TEXT, `List` TREE, TREE → `Boolean`,
(TEXT, TREE) → TEXT,
`String`, `String`, `String`) → TEXT — Write as infix

`prefix`: (TEXT, `List` TREE,
(TEXT, TREE) → TEXT,
`String`, `String`, `String`) → TEXT — Write as prefix

where

TEXT == `TextWriter`
TREE == `ExpressionTree`

Usage

infix(p, $[t_1, \ldots, t_n]$, paren?, farg, op, left, right)

Signatures

infix: (TEXT, `List` TREE, TREE → `Boolean`, (TEXT, TREE) → TEXT, `String`, `String`, `String`) → TEXT

Parameter	Type	Description
p	TEXT	The port to write to
[$t_1, \ldots, t_n$]	`List` TREE	The arguments of the operator
paren?	TREE → `Boolean`	The parenthetization function
farg	(TEXT, TREE) → TEXT	The function for the arguments
op	`String`	The infix symbol
left	`String`	The left parenthesis (optional)
right	`String`	The right parenthesis (optional)

where

TEXT == `TextWriter`
TREE == `ExpressionTree`

Description

Writes $farg(t_1)$ *op* ... *op* $farg(t_n)$ to *p*, calling *farg* on each argument, and calling *paren*? to decide whether to parenthetize each argument. Uses $left$ and $right$, which default to "(" and ")" when parenthetizing.

See also

`prefix`

Usage

lisp(p, s, $[t_1, \ldots, t_n]$)

Signature

lisp: (TEXT, `String`, `List` TREE) → TEXT

Parameter	Type	Description
p	TEXT	The port to write to
s	`String`	A Lisp operator name
$[t_1, \ldots, t_n]$	`List` TREE	The arguments of the operator

Description

Writes $(st_1 \ldots t_n)$ to p, where each t_i is written in Lisp format.

Usage

prefix(p, t, paren?, farg, op, left, right)
prefix(p, $[t_1, \ldots, t_n]$, farg, op, left, right)

Signatures

prefix: (TEXT, TREE, TREE → `Boolean`, (TEXT, TREE) → TEXT, `String`, `String`, `String`) → TEXT
prefix: (TEXT, `List` TREE, (TEXT, TREE) → TEXT, `String`, `String`, `String`) → TEXT

Parameter	Type	Description
p	TEXT	The port to write to
[$t_1, \ldots, t_n$]	`List` TREE	The arguments of the operator
paren?	TREE → `Boolean`	The parenthetization function
farg	(TEXT, TREE) → TEXT	The function for the arguments
op	`String`	The prefix symbol
left	`String`	The left parenthesis (optional)
right	`String`	The right parenthesis (optional)

where
TEXT == `TextWriter`
TREE == `ExpressionTree`

Description

Writes $op(farg(t_1), \ldots, farg(t_n))$ to p, calling $farg$ on each argument. Uses $left$ and $right$, which default to "(" and ")" for parenthetizing. The unary version calls *paren*? to decide whether to parenthetize the argument.

See also

`infix`

Usage

import from ExpressionTreePlus

Description

ExpressionTreePlus is the addition operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreePrefix s

Description

ExpressionTreePrefix s is the prefix operator with name *s* for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

All those operators share the same uniqueId, so they are equal as expression trees even though their names may be different.

Usage

import from ExpressionTreeQuotient

Description

ExpressionTreeQuotient is the division operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreeSubscript

Description

ExpressionTreeSubscript is the subscript operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreeTimes

Description

ExpressionTreeTimes is the multiplication operator for expression trees.

Exports

`ExpressionTreeOperator`

Usage

import from ExpressionTreeVector

Description

ExpressionTreeVector is the vector operator for expression trees.

Exports

`ExpressionTreeOperator`

Remarks

The following function is not yet implemented and produces dummy results : **fortran**.

Usage

Evaluator R: Category

Parameter	Type	Description
R	`PartialRing`	Resulting type of the evaluation

Description

Evaluator R is a category of interpreters, *i.e.* types which convert expression trees into elements of R whenever possible.

Exports

`eval!`:	(TEXT, TREE, SYM R) $\to \overline{R}$	Interpret an arbitrary tree
`evalLeaf!`:	(LEAF, SYM R) $\to \overline{R}$	Interpret a leaf
`evalOp!`:	(TEXT, TREE, SYM R) $\to \overline{R}$	Interpret $op(args)$
`evalPrefix!`:	(`String`, Z, `List` R, SYM R) $\to \overline{R}$	Interpret $op(args)$, op is prefix

where

Z	==	`MachineInteger`
LEAF	==	`ExpressionTreeLeaf`
$\overline{R}$	==	`Partial` R
SYM	==	`SymbolTable`
TEXT	==	`TextWriter`
TREE	==	`ExpressionTree`

Usage

import from InfixExpressionParser

Description

InfixExpressionParser implements infix expression parsers.

Exports

`ParserReader`

precedences!: (%, `MachineInteger`, `MachineInteger`) → % Set operator precedences

Usage

precedences!(p, op, n)

Signature

precedences!: (%, `MachineInteger`, `MachineInteger`) → %

Parameter	Type	Description
p	%	A parser
op	`MachineInteger`	An operator code
n	`MachineInteger`	Its new precedence

Description

Sets the precedence of op to n and returns the new parser. op must be one of PARSER__PLUS, PARSER__MINUS, PARSER__TIMES, PARSER__DIVIDE or PARSER__POWER.

Usage

import from LispExpressionParser

Description

LispExpressionParser implements lisp expression parsers.

Exports

`ParserReader`

Usage

import from MakePartialRing(R, f)

Parameter	Type	Description
R	`Ring`	A ring
f	R → `Partial Integer`	A retraction to the integers

Description

MakePartialRing(R, f) is a partial ring isomorphic to R. The function f is used to retract elements of this partial ring to integers whenever possible. This type can be used in order to build an evaluator that interprets expression trees into R.

Exports

`PartialRing`

`coerce`:	R → %	Convert an element of R to one of the partial ring
`coerce`:	% → R	Convert an element of the partial ring to one of R

Maple

Usage

import from Maple

Description

Maple provides utilities that allow its clients to batch MAPLE sessions and recover the output.

Exports

`input:`	% → `TextWriter`	Input stream of the maple session
`maple:`	() → %	Create a maple session
`run:`	% → `Partial ExpressionTree`	Run a maple session

Usage

input m

Signature

input: % → `TextWriter`

Parameter	Type	Description
m	%	A maple session

Returns

Returns the input stream for the maple session.

Remarks

Use that stream to send sequences of valid maple commands, making sure that all the commands are terminated with ':' in order to avoid any printing from MAPLE. Do not use any of MAPLE's printing functions. Note that the system maple is not started until 'run' is called.

Usage

maple()

Signature

maple: () → %

Description

Creates a maple session, by associating unique communications channels to and from that session.

Remarks

The maple input and output files used for communication are located in the /tmp directory and are deleted after the session is run, unless the call `maple(true)` is used, in which case they remain and can be inspected. Note that the system maple is not started until 'run' is called.

Usage
run m

Signature
run: % → `Partial ExpressionTree`

Parameter	Type	Description
m	%	A maple session

Description
Launches the system maple and executes all the commands that were sent to the input stream of the session. Returns the expression tree corresponding to the value returned by the last maple command executed.

Example
This examples shows how to call MAPLE to compute the integral of the 5^{th} Legendre polynomial:

```
import from Integer, Maple, ExpressionTree, Partial ExpressionTree;

n := 5;
-- create a session (maple is not launched but a unique link is created)
mapl := maple();

-- send the maple code to compute the integral of the n-th legendre poly
-- note that all the maple commands are terminated with ":"
-- so that they do not generate any output
-- here again, nothing happens, the commands are only stored
input(mapl) << "with(orthopoly): p := P(" << n << ", x): int(p, x):";

-- now launch maple and recover the result of the last command ("int")
tree := run mapl;

failed? tree => error "Unable to parse Maple's output";
retract tree;
```

Running the above code produces the following expression tree:
(+ (- (* (/ 21 16) (^ x 6)) (* (/ 35 16) (^ x 4))) (* (/ 15 16) (^ x 2)))

Usage

Parsable: Category

Description

Parsable is the category of types that convert expression trees into themselves whenever possible.

Exports

`InputType`

`eval:`	`ExpressionTree` → `Partial` %	Interpret a tree
`eval:`	`ExpressionTreeLeaf` → `Partial` %	Interpret a leaf
`eval:`	(`MachineInteger`, `List ExpressionTree`) → `Partial` %	Interpret a node

Usage

eval e
eval t
eval(op,$[e_1, \ldots, e_n]$)

Signatures

eval: `ExpressionTree` → `Partial` %
eval: `ExpressionTreeLeaf` → `Partial` %
eval: (`MachineInteger`, `List ExpressionTree`) → `Partial` %

Parameter	Type	Description
e,e_i	`ExpressionTree`	Expression trees
t	`ExpressionTreeLeaf`	A leaf
op	`MachineInteger`	Code for an operator

Returns

eval(e) and eval(t) return the result of evaluating the given tree or leaf in the type, while eval(op,$[e_1, \ldots, e_n]$) returns the result of evaluating $op(e_1, \ldots, e_n)$ in the type, where op is a code from `include/algebrauid.as`.

Parser

Usage

Parser: Category

Description

Parser is the category for parser objects.

Exports

`eof?`:	% → `Boolean`	Check for end of input
`lastError`:	% → `MachineInteger`	Code for last parsing error
`parse!`:	% → `Partial ExpressionTree`	Parse one expression

Usage

eof? p

Signature

eof?: % → `Boolean`

Parameter	Type	Description
p	%	A parser

Returns

Returns *true* if the input is finished, *false* otherwise.

Usage

lastError p

Signature

lastError: % → `MachineInteger`

Parameter	Type	Description
p	%	A parser

Returns

Returns the code for the last parsing error.

Usage

parse! p

Signature

parse!: % → `Partial ExpressionTree`

Parameter	Type	Description
p	%	A parser

Returns

Returns either an expression tree for the next parsed expression, or *failed* in case of error or end of input.

See also

lastError(%)

ParserReader

Usage

ParserReader: Category

Description

ParserReader is the category for parser objects that parse text readers.

Exports

`parser`: `TextReader` → % Create a parser

Usage

parser r

Signature

parser: `TextReader` → %

Parameter	Type	Description
r	`TextReader`	The input stream to parse

Returns

Returns a parser that takes its input on r.

Usage

import from ParsingTools R

Parameter	Type	Description
R	`Parsable ArithmeticType`	A parsable arithmetic system

Description

ParsingTools R provides tools for converting expression trees into elements of R.

Exports

`evalArith`:	(Z, `List` TREE) → `Partial` R	Interpret an arithmetic expression
`evalInt`:	TREE → `Partial Integer`	Interpret an integer

where

TREE == `ExpressionTree`
Z == `MachineInteger`

Usage

evalArith(op,$[e_1, \ldots, e_n]$)

Signature

evalArith: (`MachineInteger`, `List ExpressionTree`) → `Partial` R

Parameter	Type	Description
op	`MachineInteger`	Code for an operator
e_i	`ExpressionTree`	Expression trees

Returns

Returns the result of evaluating $op(e_1, \ldots, e_n)$ where *op* is a code from `include/algebrauid.as`. Provides support for the evaluation of the operators $+$, $-$, $*$ and $\hat{}$, as well as $/$ when R has `CommutativeRing` or `FloatType`.

Usage

evalInt e

Signature

evalInt: `ExpressionTree` → `Partial Integer`

Parameter	Type	Description
e	`ExpressionTree`	An expression tree

Returns

Returns the value of *e* as an integer if it is an integer–valued leaf, *failed* otherwise.

Usage

PartialRing: Category

Description

PartialRing is the category of rings where all the arithmetic operations are partial, *i.e.* they are allowed to fail. Typical examples are matrices of different sizes, or unions of several true rings.

Exports

`ExpressionType`

`0`:	%	Additive identity
`1`:	%	Multiplicative identity
`-`:	% → `Partial` %	Negation
`-`:	(%, %) → `Partial` %	Substraction
`+`:	(%, %) → `Partial` %	Addition
`*`:	(%, %) → `Partial` %	Multiplication
`/`:	(%, %) → `Partial` %	Exact division
`^`:	(%, %) → `Partial` %	Exponentiation
`<`:	(%, %) → `Partial` %	Comparison
`>`:	(%, %) → `Partial` %	Comparison
≤:	(%, %) → `Partial` %	Comparison
≥:	(%, %) → `Partial` %	Comparison
[]:	`Tuple` % → `Partial` %	Construct a structure
`coerce`:	`Boolean` → %	Convert a boolean to a ring element
`coerce`:	`Integer` → %	Convert an integer to a ring element
`integer`:	% → `Partial Integer`	Convert to an integer
`product`:	List % → `Partial` %	Multiplication
`sum`:	List % → `Partial` %	Addition

Usage

import from Scanner

Description

Scanner provides a simple scanner for mathematical expressions.

Exports

`scan`: `TextReader` → `Token` Scan a token

Usage

scan p

Signature

scan: `TextReader` → `Token`

Parameter	Type	Description
p	`TextReader`	Text to scan

Returns

Returns the next token read from the reader *p*.

Usage

import from Shell(P, R, E)

Parameter	Type	Description
P	`Parser`	A parser type for the input
R	`PartialRing`	Resulting type of the evaluation
E	`Evaluator` R	An interpreter with target type R

Description

Shell(P, R, E) provides a basic read-eval-loop that converts its input to expressions of type R.

Exports

`center`:	(TW, `String`) → TW	center a line of text
`shell!`:	(P, TW, TW, `SymbolTable` R, `Boolean`, `Boolean`, `String`) → Z	Read-eval loop

where

TW == `TextWriter`
Z == `MachineInteger`

Usage

center(p, s)

Signature

center: (`TextWriter`, `String`) → `TextWriter`

Parameter	Type	Description
p	`TextWriter`	The port to write to
s	`String`	A string to center

Description

Writes *s* centered in a 66-character line to the port *p*, followed by a newline, and returns the port after the write.

Usage

shell!(r, p_1, p_2, t, verbose?, time?, s)

Signature

shell!: (P, TW, TW, `SymbolTable` R, `Boolean`, `Boolean`, `String`) → `MachineInteger`

where

TW == `TextWriter`

Parameter	Type	Description
r	P	A parser
p_1, p_2	`TextWriter`	The ports to write to
t	`SymbolTable` R	The initial symbol table
verbose?	`Boolean`	Select verbose or quiet mode
time?	`Boolean`	Select whether to return times in msecs
s	`String`	A string to write after processing commands

Description

Starts a read-eval loop which takes its input from r and writes its output to p_1 or p_2. Mathematical output generated by user commands is written to p_1, while prompts and execution times are written to p_2. if verbose? is *false* (quiet mode), nothing is printed to p_2 and no prompts or execution times are communicated. Regardless of verbose?, if time? is *true*, then the execution and garbage collection times are written to p_1 in milliseconds after each command. If s is not empty, then it is sent to p_1 after each command is executed. The table t contains the initial environment, and can be modified by the loop. Returns an integer $n = b_1 b_0$ where b_0 is 1 if a syntax error occured, 0 otherwise, and b_1 is 1 if any other error occured, 0 otherwise.

Usage

import from SymbolTable T

Parameter	Type	Description
T	`PrimitiveType`	Type of the symbols in the table

Description

SymbolTable T provides symbol tables where all the symbols have type T.

Exports

`apply`:	(%, `Symbol`) → `Partial` T	Search for a symbol
`set!`:	(%, `Symbol`, T) → T	Add a symbol
`table`:	() → %	Create an empty table

Usage

apply(t, x)
t x

Signature

apply: (%, `Symbol`) → `Partial` T

Parameter	Type	Description
t	%	A symbol table
x	`Symbol`	A variable name

Returns

Returns the value that x has in t if it is found, *failed* otherwise.

Usage

set!(t, x, v)
t.x := v

Signature

set!: (%, `Symbol`, T) → T

Parameter	Type	Description
t	%	A symbol table
x	`Symbol`	A variable name
v	T	A value

Description

Assigns the value v to x in t. If x already had a value in t, the older value is lost.

Returns

Returns v.

Usage

table()

Signature

table: () → %

Returns

Returns an empty symbol table.

See also

`set!`

Usage

import from Token

Description

Token is a type whose elements are parser tokens.

Exports

`OutputType`
`PrimitiveType`

`float`:	(`List Character`, `List Character`) → %	Create a float token
`integer`:	`List Character` → %	Create an integer token
`leaf`:	% → `ExpressionTreeLeaf`	Conversion to a leaf
`leaf?`:	% → `Boolean`	Test for a leaf
`name`:	`List Character` → %	Create a constant name token
`operator`:	% → `ExpressionTreeOperator`	Conversion to an operator
`operator?`:	% → `Boolean`	Test for an operator
`prefix`:	`List Character` → %	Create a prefix function token
`special`:	% → `MachineInteger`	Conversion to a special token
`special?`:	% → `Boolean`	Test for a special token
`string`:	`List Character` → %	Create a string
`token`:	`Character` → `Partial` %	Create a single character token

Usage

float(l_1, l_2)

Signature

float: (`List Character`, `List Character`) $\to$ %

Parameter	Type	Description
$[d_0, \ldots, d_n]$	`List Character`	A list of digits
$[e_0, \ldots, e_m]$	`List Character`	A list of digits

Returns

float$([d_0, \ldots, d_n], [e_0, \ldots, e_m])$ returns the float $d_n d_{n-1} \ldots d_0 . e_m e_{m-1} \ldots e_0$ as a token.

See also

`integer`

Usage

integer l

Signature

integer: `List Character` $\to$ %

Parameter	Type	Description
$[d_0, \ldots, d_n]$	`List Character`	A list of digits

Returns

integer($[d_0, \ldots, d_n]$) returns the integer $d_0 + 10d_1 + \ldots + 10^n d_n$ as a token.

See also

`float`

Usage

leaf t
leaf? t

Signatures

leaf: % → `ExpressionTreeLeaf`
leaf?: % → `Boolean`

Parameter	Type	Description
t	%	A token

Returns

leaf a returns *a* as a leaf is *a* is a leaf. leaf? a returns *true* if a is a leaf, *false* otherwise.

See also

`operator`, `special`

Usage

name l

Signature

name: `List Character` → %

Parameter	Type	Description
$[c_0, \ldots, c_n]$	`List Character`	A list of characters

Returns

name($[c_0, \ldots, c_n]$) returns the name $c_n c_{n-1} \ldots c_0$ as a token representing a constant symbol.

See also

`prefix`, `string`

Usage

operator t
operator? t

Signatures

operator: % → `ExpressionTreeOperator`
operator?: % → `Boolean`

Parameter	Type	Description
t	%	A token

Returns

operator a returns *a* as an operator is *a* is an operator. operator? a returns *true* if a is an operator, *false* otherwise.

See also

`leaf`, `special`

Usage

prefix l

Signature

prefix: `List Character` → %

Parameter	Type	Description
$[c_0, \ldots, c_n]$	`List Character`	A list of characters

Returns

prefix($[c_0, \ldots, c_n]$) returns the name "$c_n c_{n-1} \ldots c_0''$ as a token representing a prefix function.

See also

`name`

Usage

special t
special? t

Signatures

special: % → `MachineInteger`
special?: % → `Boolean`

Parameter	Type	Description
t	%	A token

Returns

special a returns *a* as a special token is *a* is a special token. special? a returns *true* if a is a special token, *false* otherwise.

See also

`leaf`, `special`

Usage

name l

Signature

name: `List Character` → %

Parameter	Type	Description
$[c_0, \ldots, c_n]$	`List Character`	A list of characters

Returns

name($[c_0, \ldots, c_n]$) returns the string "$c_n c_{n-1} \ldots c_0''$ as a token representing a constant.

See also

`name`, `prefix`

Usage

token a

Signatures

token: `Character` → `Partial` %
token: `ExpressionTreeOperator` → `Partial` %
token: `MachineInteger` → `Partial` %

Parameter	Type	Description
a	`Character`	A single character token
	`ExpressionTreeOperator`	An operator
	`MachineInteger`	A code of a special token

Returns

Returns the token corresponding to the single character *a*, *failed* if there is none.

Usage

import from AlgebraLibraryInformation

Description

AlgebraLibraryInformation provides version information about the `Algebra` library.

Exports

`VersionInformationType`

Usage

import from Bits

Description

Bits is a type whose elements are arrays of bits, which can be used as sets of flags. The implementation is more compact than arrays of boolean values, but the functionality is the same.

Exports

`apply`:	(%, Z) → `Boolean`	test a bit
`dimension`:	(%, Z) → Z	number of true bits
`false`:	Z → %	create a set of false bits
`set!`:	(%, Z, `Boolean`) → `Boolean`	set a bit
`true`:	Z → %	create a set of true bits

where

Z == `MachineInteger`

Usage

b n
apply(b, n)

Signature

apply: (%, `MachineInteger`) → `Boolean`

Parameter	Type	Description
b	%	A set of bits
n	`MachineInteger`	An index

Returns

Returns the value of the n^{th} bit of *b*, the first index being 1.

Signature

dimension: (%, `MachineInteger`) → `MachineInteger`

Usage

dimension(b, n)

Parameter	Type	Description
b	%	A set of bits
n	`MachineInteger`	An index

Returns

Returns the number of bits of index between 1 and n that are true.

Usage

false n
true n

Signature

false,true: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	The number of bits to create

Returns

false(*n*) (resp. true(*n*)) returns an array of *n* bits, all set to *false* (resp. *true*).

Usage

b.n := v
set!(b, n, v)

Signature

set!: (%, `MachineInteger`, `Boolean`) → `Boolean`

Parameter	Type	Description
b	%	A set of bits
n	`MachineInteger`	An index
v	`Boolean`	A binary value

Returns

Sets the n^{th} bit of b to v and returns v, the first index being 1.

Usage

import from IndexedVariable S
import from IndexedVariable(S, x)

Parameter	Type	Description
S	`ExpressionType` `TotallyOrderedType`	The type of the indices
x	`Symbol`	The root variable (optional)

Description

IndexedVariable provides sorted symbols of the form x_s where $s \in S$.

Exports

`ExpressionType`
`TotallyOrderedType`

`index`:	% → S	Index of a variable
`variable`:	S → %	Create an indexed variable

Usage

index v

Signature

index: % → S

Parameter	Type	Description
v	%	An indexed variable

Returns

Returns the index of *v*.

Usage

variable s

Signature

variable: S → %

Parameter	Type	Description
s	S	An index

Returns

Returns the variable x_s.

Usage

import from PartialFunction(R, S)

Parameter	Type	Description
R	`Type`	Contains the domain of the function
S	`Type`	Contains the image of the function

Description

PartialFunction provides partial functions from R to S, *i.e.*functions from R to S which are allowed to fail on some elements of R.

Exports

`apply`:	(%, R) → S	Apply a partial function
`inDomain?`:	(%, R) → `Boolean`	Check if an element is in the domain
`mapping`:	% → (R → S)	Action of a function
`partialApply`:	(%, R) → `Partial` S	Apply a partial function
`partialMapping`:	% → (R → `Partial` S)	Action of a function
`predicate`:	% → (R → `Boolean`)	Domain of a partial function
`partialFunction`:	(R → `Partial` S) → %	Create a partial function
`partialFunction`:	((R → `Boolean`, R → S)) → %	Create a partial function

Usage

apply(σ, x)
σx

Signature

apply: (%, R) → S

Parameter	Type	Description
σ	%	A partial function
x	R	An element of R

Returns

Returns σx.

Remarks

This map can cause an error if it is used on an element which is not in the domain of σ. Use only when x is known to be in the domain, otherwise use *partialApply*.

See also

`inDomain?`, `predicate`, `partialApply`

Usage

inDomain? x

Signature

inDomain?: (%, R) → `Boolean`

Parameter	Type	Description
σ	%	A partial function
x	R	An element of R

Returns

Returns *true* if x is in the domain of σ, *false* otherwise.

See also

`predicate`

Usage

mapping σ

Signature

mapping: $\% \to (\mathrm{R} \to \mathrm{S})$

Parameter	Type	Description
σ	%	A partial function

Returns

Returns the map corresponding to the action of σ on R.

Remarks

The map returned can cause an error if it is used on an element which is not in the domain of σ. Use only when x is known to be in the domain, otherwise use *partialMapping*.

See also

`inDomain?`, `partialMapping`, `predicate`

Usage

partialMapping σ

Signature

partialMapping: % → (R → `Partial` S)

Parameter	Type	Description
σ	%	A partial function

Returns

Returns the map corresponding to the action of σ on R.

See also

`inDomain?`, `mapping`, `predicate`

Usage

predicate σ

Signature

predicate: $\% \to (\mathrm{R} \to$ `Boolean`

Parameter	Type	Description
σ	%	A partial function

Returns

Returns the predicate defining the domain of σ.

See also

`inDomain?`

Usage

partialFunction f
partialFunction(p, g)

Signatures

partialFunction: (R → `Partial` S) → %
partialFunction: (R → `Boolean`, R → S) → %

Parameter	Type	Description
f	R → `Partial` S	A partial map
p	R → `Boolean`	A predicate
g	R → S	A map

Returns

partialFunction(f) returns the partial function σ on R given by

$$\sigma x = f(x)$$

for any $x \in R$, while partialFunction(p, g) returns the partial function σ on R given by

$$\sigma x = \begin{cases} g(x) & if p(x) = true \\ failed & if p(x) = false \end{cases}$$

Usage

partialApply x

Signature

partialApply: (%, R) → `Partial` S

Parameter	Type	Description
σ	%	A partial function
x	R	An element of R

Returns

Returns σx, or *failed* if x is not in the domain of σ.

See also

`apply`, `inDomain?`, `predicate`

Usage

import from Permutation n

Parameter	Type	Description
n	`MachineInteger`	The number of elements

Description

Permutation(n) implements the group of permutations on n elements.

Exports

`CopyableType`
`Group`

`apply`:	(%, Z) → Z	Image of an element
`mapping`:	% → (Z → Z)	Action of a permutation
`sign`:	% → Z	Sign
`transpose`:	(Z, Z) → %	Transposition
`transpose!`:	(%, Z, Z) → %	Compose with a transposition

where

Z == `MachineInteger`

Usage

apply(σ, x)
σx

Signature

apply: (%, `MachineInteger`) → `MachineInteger`

Parameter	Type	Description
σ	%	A permutation
x	`MachineInteger`	An index

Returns

Returns σx.

Usage

mapping σ

Signature

mapping: $\% \to (\texttt{MachineInteger} \to \texttt{MachineInteger})$

Parameter	Type	Description
σ	$\%$	A permutation

Returns

Returns the map corresponding to σ.

See also

`apply`

Usage

sign σ

Signature

sign: % → `MachineInteger`

Parameter	Type	Description
σ	%	A permutation

Returns

Returns the sign of σ, *i.e.*$(-1)^{\epsilon}$ where ϵ is the number of tranpositions in the factorization of σ.

Usage

transpose(x,y)
transpose!(σ,x,y)

Signatures

transpose: (`MachineInteger`, `MachineInteger`) $\to$ %
transpose!: (%, `MachineInteger`, `MachineInteger`) $\to$ %

Parameter	Type	Description
σ	%	A permutation
x,y	`MachineInteger`	Indices

Returns

transpose(x,y) returns the transposition (xy), while transpose!(σ,x,y) replaces σ by the composition $(xy) \circ \sigma$ and returns it.

Remarks

transpose! does not make a copy of σ, but performs all the computations in–place, storing the final result in σ. Also, it does not create the transposition (xy).

Usage

import from Sequence R

Parameter	Type	Description
R	`ExpressionType`	The coefficient domain
	`ArithmeticType`	

Description

Sequence R implements infinite sequences with coefficients in R.

Exports

`LinearStructureType` R

`MonogenicLinearArithmeticType` R

`#`:	% → Z	number of computed elements
`bound`:	% → `MachineInteger`	upper bound on the support size
`finite?`:	% → `Boolean`	check whether the support is finite
`sequence`:	`Stream` R → %	create a sequence

if R has `Ring` then

`random`:	() → %	random sequence

Usage

s

Signatures

#: % → `MachineInteger`

Parameter	Type	Description
s	%	a sequence

Returns

Returns the number of elements of *s* that have been computed.

Usage

bound s
finite? s

Signatures

bound: % → `MachineInteger`
finite?: % → `Boolean`

Parameter	Type	Description
s	%	a sequence

Returns

finite?(s) returns *true* if s is known to have finite support and *false* otherwise, while bound(s) returns $n \geq 0$ if s is known to have finite support and $s.m = 0$ for $m \geq n$, -1 otherwise.

Usage

random()

Signature

random: () → %

Returns

Returns a sequence with random entries.

Usage

sequence s

Signature

sequence: `Stream` R → %

Parameter	Type	Description
s	`Stream` R	a stream

Returns

Returns *s* viewed as a sequence.

Symbol

Usage

import from Symbol

Description

Symbolprovides symbols, *i.e.*read–only strings with constant–time comparison.

Exports

`HashType`
`InputType`
`SerializableType`
`ExpressionType`

−:	`String` → %	Create a symbol
`name`:	% → `String`	Name of a symbol
`new`:	() → %	Create a new symbol

Subscripted name of a symbol

Usage

$-s$

Signature

$-$: `String` $\to$ %

Parameter	Type	Description
s	`String`	A string

Returns

Returns s as a symbol.

Usage

name s

Signature

name: % → `String`

Parameter	Type	Description
s	%	A symbol

Returns

Returns a new copy of the name of s.

Remarks

Modifying name(s) does not modify s, since a new copy is created at each call.

Usage
new()

Signature
new: () → %

Returns
Returns a new symbol.

4 libaldor Reference Manual

Usage

import from AldorInteger

Description

AldorInteger provides an interface to the software ("infinite" precision) integers provided by the ALDOR virtual machine.

Exports

`IntegerType`

Usage

AdditiveType: Category

Description

AdditiveType is the category of types with addition/substraction operations.

Exports

`PrimitiveType`

`0` :	%	zero
`+` :	(%, %) → %	addition
`-` :	% → %	opposite
`-` :	(%, %) → %	substraction
`add!`:	(%, %) → %	In–place addition
`minus!`:	% → %	In–place opposite
`minus!`:	(%, %) → %	In–place substraction
`zero?`:	% → `Boolean`	test for 0

Usage

0

Signature

0: %

Returns

Return the 0 constant of the type.

Usage

$x + y$
$x - y$
$-x$

Signatures

$-$: $\% \to \%$
$+, -$: $(\%, \%) \to \%$

Parameter	Type	Description
x,y	$\%$	elements of the type

Returns

$x + y, x - y$ return respectively the sum and difference x with y, while $-x$ returns the opposite of x.

See also

`add!`, `minus!`

Usage

add!(x, y)
minus!(x, y)
minus! x

Signatures

minus!:	$\% \to \%$
add!, minus!:	$(\%, \%) \to \%$

Parameter	Type	Description
x, y	%	Elements of the type

Returns

add!(x, y) and minus!(x, y) returns respectively $x + y$ and $x - y$, while minus! x returns the opposite of x. In all cases, the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

Those functions may cause x to be destroyed, so do not use them unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

+,-

Usage

zero? x

Signature

zero?: % → `Boolean`

Parameter	Type	Description
x	%	an element of the type

Returns

Returns the result of $x = 0$ using the semantics of $=$ of the type.

Usage

ArithmeticType: Category

Description

ArithmeticType is the category of types with standard arithmetic operations.

Exports

`AdditiveType`

`1`:	%	one
`*`:	(%, %) → %	product
`^`:	(%, `MachineInteger`) → %	exponentiation
`commutative?`:	`Boolean`	check whether $*$ is commutative
`one?`:	% → `Boolean`	test for 1
`times!`:	(%, %) → %	In–place product

Usage

1

Signature

1: %

Returns

Return the 1 constant of the type.

Usage

$x * y$
$x \hat{} n$

Signatures

$*$: (%, %) → %
$\hat{}$: (%, `MachineInteger`) → %

Parameter	Type	Description
x,y	%	elements of the type
n	`MachineInteger`	an exponent

Returns

$x * y$ returns the product of x with y, while $x \hat{} n$ returns x to the power n.

See also

`times!`

Usage

commutative?

Signature

commutative?: `Boolean`

Returns

Returns *true* if `*` is commutative, *false* otherwise.

Usage

one? x

Signature

one?: % → `Boolean`

Parameter	Type	Description
x	%	an element of the type

Returns

Returns the result of $x = 1$ using the semantics of $=$ of the type.

Usage

times!(x, y)

Signature

times!: (%, %) → %

Parameter	Type	Description
x, y	%	Elements of the type

Returns

Return xy, where the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

This function may cause x to be destroyed, so do not use it unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

*

Usage

import from BinaryPowering(T, Z)

Parameter	Type	Description
T	`ArithmeticType`	An arithmetic system
Z	`IntegerType`	An integer–like type

Description

BinaryPowering provides binary exponentiation of elements of T with exponents in Z.

Exports

`binaryExponentiation`:	(T, Z) → T	Binary powering
`binaryExponentiation!`:	(T, Z) → T	In–place binary powering

Usage

binaryExponentiation(a, n)
binaryExponentiation!(a, n)

Signature

binaryExponentiation: (T, Z) → T

Parameter	Type	Description
a	T	The element to exponentiate
n	Z	The exponent

Returns

Returns a^n. The exponent n must be nonnegative. When using binaryExponentiation!(a, n), the storage used by a and n is allowed to be destroyed or reused, so a and n are lost after this call.

Remarks

A call to binaryExponentiation!(a, n) may cause a and n to be destroyed, so do not use it unless a and n have been locally allocated, and are guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

import from Boolean

Description

Boolean implements the boolean values *true* and *false*.

Exports

`BooleanArithmeticType`
`HashType`
`false`: % *false*
`true`: % *true*

Usage

true
false

Signature

true,false: %

Returns

true and false return the boolean values *true* and *false* respectively.

Usage

BooleanArithmeticType: Category

Description

BooleanArithmeticType is the category of types allowing boolean arithmetic.

Exports

`PrimitiveType`

`~`:	% → %	negation
/\:	(%, %) → %	and
\/:	(%, %) → %	or
`xor`:	(%, %) → %	exclusive or

Usage

~ a
a /\ b
a \/ b
xor(a, b)

Signatures

~ : % → %
/\, \/,xor: (%,%) → %

Parameter	Type	Description
a, b	%	elements of the type

Returns

~*a* returns *not*(*a*), while $a\backslash/b$ returns (*a or b*), $a/\backslash b$ returns (*a and b*), and xor(a, b) returns

$$(a/\backslash\tilde{}\, b)\backslash/(\tilde{}\, a/\backslash b)\,.$$

The semantics of *not*, *or* and *and* can be logical or bitwise, depending on the actual type.

Remarks

For the type `Boolean`, the difference between $a\backslash/b$ and `a or b` is that $a\backslash/b$ guarantees that both expressions *a* and *b* are evaluated while `a or b` may evaluate only *a* and return *true* if *a* evaluates to *true*. There is a similar difference between $a/\backslash b$ and `a and b`.

Example

If a and b are the `MachineInteger` 5 and 7, then $\tilde{}\,a = -6$, $a/\backslash b = 5$, $a\backslash/b = 7$ and xor$(a, b) = 2$.

Usage

import from Complex R

Parameter	Type	Description
R	`ArithmeticType`	Type to be extended

Description

Complex R implements the algebraic extension of R generated by a root of $X^2 + 1 = 0$. The coefficient type R must satisfy 2 additional mathematical properties for the arithmetic of Complex R to be correct, namely:

- $X^2 + 1$ is irreducible over R.
- The element α such that $\alpha^2 + 1 = 0$ must commute with all the elements of R (which is obviously satisfied when the multiplication of R is commutative).

Exports

`ArithmeticType`
`CopyableType`
`LinearCombinationType` R

`coerce`:	R → %	Natural embedding
`complex`:	(R, R) → %	create a complex
`conjugate`:	% → %	conjugation
`conjugate!`:	% → %	in-place conjugation
`copy!`:	(%, R, R) → %	in-place copy
`imag`:	% → R	imaginary part
`real`:	% → R	real part

if R has `FloatType` then
`/`: (%, %) → % Division

if R has `InputType` then
`InputType`

if R has `OutputType` then
`OutputType`

if R has `SerializableType` then
`SerializableType`

Usage

coerce x
x::%

Signature

coerce: R → %

Parameter	Type	Description
x	R	an element of the base type

Returns

Returns the complex $x + 0\sqrt{-1}$.

Usage

complex(x,y)

Signature

complex: (R, R) → %

Parameter	Type	Description
x, y	R	real and imaginary parts

Returns

Returns the complex $x + y\sqrt{-1}$.

Usage

conjugate z
conjugate! z

Signature

conjugate: $\% \to \%$

Parameter	Type	Description
z	$\%$	a complex

Description

Returns $x - y\sqrt{-1}$ where $z = x + y\sqrt{-1}$. When using conjugate!, the storage used by z is allowed to be destroyed or reused, so z is lost after this call.

Usage

copy!(z,x,y)

Signature

copy!: (%, R, R) → %

Parameter	Type	Description
z	%	a complex
x, y	R	real and imaginary parts

Description

Replaces z by $x + y\sqrt{-1}$ and return z, where the storage used by z is allowed to be destroyed or reused, so z is lost after this call.

Remarks

This call may cause z to be destroyed, so do not use it unless z has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

`copy!`

Usage

imag,real z

Signature

imag,real: % → R

Parameter	Type	Description
z	%	a complex

Returns

imag(z) and real(z) return respectively y and x where $z = x + y\sqrt{-1}$.

Usage

norm z

Signature

norm: $\% \to \%$

Parameter	Type	Description
z	$\%$	a complex

Returns

Returns $z\bar{z}$ where $\bar{z}$ is the conjugate of z.

Remarks

When R is commutative, the imaginary part of $z\bar{z}$ is 0, and the usual complex norm can be computed via `real norm z`. The imaginary part of $z\bar{z}$ is not necessarily 0 when R is not commutative.

Usage

import from DoubleFloat

Description

DoubleFloat implements the single-precision signed machine floats.

Exports

`CopyableType`
`FloatType`
`PackableType`

$\hat{}$:	(%, %) → %	exponentiation
`max`:	%	largest single–precision machine float
`min`:	%	smallest single–precision machine float

Usage

$x \hat{} y$

Signatures

$\hat{}$: (%,%) → %

Parameter	Type	Description
x,y	%	floats

Returns

Returns x to the power y.

Usage

max
min

Signature

max,min: %

Returns

max and min return respectively the largest and the smallest double–precision machine floats.

Usage

FloatType: Category

Description

FloatType is the category of types representing floats.

Exports

`InputType`
`OrderedArithmeticType`
`OutputType`
`SerializableType`

/:	(%, %) → %	division
`coerce`:	`MachineInteger` → %	conversion to a float
`fraction`:	% → %	fractional part
`truncate`:	% → `AldorInteger`	truncation

Usage

x/y

Signature

/: (%,%) → %

Parameter	Type	Description
x,y	%	floats

Returns

Returns the quotient of x by y.

Usage

n::%

Signature

coerce: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	a machine integer

Returns

Returns n converted to a float.

Usage

fraction x
truncate x

Signatures

fraction: % → %
truncate: % → `AldorInteger`

Parameter	Type	Description
x	%	a float

Returns

truncate(x) returns n such that $nx \geq 0$ and $|n| \leq |x| < |n| + 1$, while fraction(x) returns $x -$ truncate(x).

Usage

import from GMPFloat

Description

GMPFloat implements arbitrary precision floats with the float arithmetic provided by GMP.

Exports

`CopyableType`
`FloatType`

`coerce`:	`DoubleFloat` → %	conversion
`limbs`:	% → `Generator MachineInteger`	iteration
`new`:	() → %	initialization (for calling GMP)
`precision`:	% → `MachineInteger`	precision
`setPrecision!`:	(%, `MachineInteger`) → %	adjust precision

Usage

x::%

Signature

coerce: `DoubleFloat` → %

Parameter	Type	Description
x	`DoubleFloat`	a machine float

Returns

Converts x to a GMP float.

Usage

precision x
setPrecision!(x, n)

Signatures

precision: % → `MachineInteger`
setPrecision!: (%, `MachineInteger`) → %

Parameter	Type	Description
x	%	a float
n	`MachineInteger`	a precision

Description

precision(x) returns the precision acutally used for x, while setPrecision!(x, n) sets the precision for x to be at least n bits.

Remarks

Those functions are wrapper to the `mpf_get_prec` and `mpf_set_prec` GMP functions, so setPrecision! can involve a call to `realloc`.

Usage

for d in limbs x repeat { ... }

Signature

limbs: % → `Generator MachineInteger`

Parameter	Type	Description
x	%	a float

Description

This function allows the individual limbs of a GMP float to be iterated independently of the machine size. This generator yields the limbs from the least to the most significant.

Remarks

The limbs of x describe $|x|$, so combine it with `sign` if you need a complete description of x.

Signature

new: () → %

Returns

Returns an initialized GMP float but with no value stored into it. Results from this function can be used only as parameters to explicit calls to `mpf_` functions.

Usage

import from GMPInteger

Description

GMPInteger implements arbitrary precision integers with the integer arithmetic provided by GMP.

Exports

`CopyableType`

`IntegerType`

`coerce:`	`AldorInteger` → %	conversion
`coerce:`	% → `AldorInteger`	conversion
`limbs:`	% → `Generator MachineInteger`	iteration
`new:`	() → %	initialization (for calling GMP)

Usage

m::%
n::`AldorInteger`

Signatures

coerce: `AldorInteger` → %
coerce: % → `AldorInteger`

Parameter	Type	Description
m	%	an ALDOR integer
n	`AldorInteger`	a gmp integer

Description

Those functions convert between ALDOR and GMP integers.

Remarks

The conversion to an `AldorInteger` can be quadratic in the number of bits of the integer, which is expensive.

Usage

for d in limbs n repeat { ... }

Signature

limbs: % → `Generator MachineInteger`

Parameter	Type	Description
n	%	an integer

Description

This function allows the individual limbs of a GMP integer to be iterated independently of the machine size. This generator yields the limbs from the least to the most significant.

Remarks

The limbs of n describe $|n|$, so combine it with `sign` if you need a complete description of n.

Signature

new: () → %

Returns

Returns an initialized GMP integer but with no value stored into it. Results from this function can be used only as parameters to explicit calls to `mpz_` functions.

Usage

import from IntegerSegment Z

Parameter	Type	Description
Z	`IntegerType`	an integer type

Description

IntegerSegment(Z) implements open and closed segments of Z, *i.e.*a selection of equally spaced integers in a range of the form $[a, b]$ or $[a, +\infty)$.

Exports

`PrimitiveType`
`InputType`
`OutputType`
`SerializableType`

`..`:	Z → %	creation of a segment
	(Z, Z) → %	
`by`:	(%, Z) → %	change the spacing
`generator`:	% → `Generator` Z	iterate over a segment
`high`:	% → Z	upper bound
`low`:	% → Z	lower bound
`open?`:	% → `Boolean`	check whether a segment is open
`step`:	% → Z	spacing

Usage

a..
a..b

Signatures

..: Z → %
..: (Z, Z) → %

Parameter	Type	Description
a,b	Z	integers

Returns

a.. returns the open range $[a, +\infty)$ while a..b returns the closed range $[a, b]$. Every integer in the range belongs to the resulting segment.

See also

`by`

Usage

s by n
step s

Signatures

by: (%,Z) → %
step: % → Z

Parameter	Type	Description
s	%	a segment
n	Z	a step

Returns

s by n changes s to become the segment consisting of every n^{th} integer in its range, *i.e.*the integers $a, a+n, a+2n, \ldots$ that are within the range $[a, b]$ or $[a, +\infty)$ of s, while step(s) returns n such that s is the segment consisting of every n^{th} integer in its range.

Remarks

The function s by n does not create a new segment but side-effects s, whose former step is lost.

Usage

for x in s repeat { ... }
for x in generator s repeat { ... }

Signature

generator: % → `Generator` Z

Parameter	Type	Description
s	%	a segment

Description

This functions allows a segment to be iterated. This generator yields the integers of s in succession.

Example

The following code computes the sum of all the positive even machine integers that are smaller than n:

```
evenSum(n:MachineInteger):MachineInteger == {
    s := 0;
    for x in 2..prev(n) by 2 repeat s := s + x;
    s;
}
```

Usage

high s
low s

Signature

high,low: % → Z

Parameter	Type	Description
s	%	a segment

Returns

high(s) and low(s) return respectively the upper and lower bound of the range of s. The result of high(s) is undefined if s is an open segment.

Usage

open? s

Signature

open?: % → `Boolean`

Parameter	Type	Description
s	%	a segment

Returns

Returns *true* if the range of s is infinite, *false* otherwise.

Usage

IntegerType: Category

Description

IntegerType is the category of types representing integers.

Exports

`BooleanArithmeticType`
`HashType`
`InputType`
`OrderedArithmeticType`
`OutputType`
`SerializableType`

`bit?`:	(%, `MachineInteger`) → `Boolean`	check a bit
`clear`:	(%, `MachineInteger`) → %	clear a bit
`coerce`:	`MachineInteger` → %	conversion from machine integer
`divide`:	(%, %) → (%, %)	Euclidean division
`even?`:	% → `Boolean`	test whether a number is even
`factorial`:	% → %	factorial
`gcd`:	(%, %) → %	greatest common divisor
`lcm`:	(%, %) → %	least common multiple
`length`:	% → `MachineInteger`	number of bits
`machine`:	% → `MachineInteger`	conversion to a machine integer
`mod`:	(%, %) → %	remainder
	(%, `MachineInteger`) → `MachineInteger`	
`next`:	% → %	next greater integer
`nthRoot`:	(%, %) → (`Boolean`, %)	n^{th}–root
`odd?`:	% → `Boolean`	test whether a number is odd
`prev`:	% → %	next smaller integer
`quo`:	(%, %) → %	quotient
`random`:	() → %	random integer
	`MachineInteger` → %	
`rem`:	(%, %) → %	remainder
`set`:	(%, `MachineInteger`) → %	set a bit
`shift`:	(%, `MachineInteger`) → %	shift
`shift!`:	(%, `MachineInteger`) → %	in–place shift

Usage

bit?(a, n)
clear(a, n)
set(a, n)

Signatures

bit?:	(%, `MachineInteger`) → `Boolean`
clear, set:	(%, `MachineInteger`) → %

Parameter	Type	Description
a	%	an integer
n	`MachineInteger`	a nonnegative machine integer

Returns

bit?(a, n) returns *true* if the n^{th} bit of a is 1, *false* if it is 0, while clear(a, n) and set(a, n) return copies of a where the n^{th} bit is set respectively to 0 and 1. For all 3 functions, the rightmost bit of a is the 0^{th} bit and so on.

Usage

n::%
machine a

Signatures

coerce: `MachineInteger` → %
machine: % → `MachineInteger`

Parameter	Type	Description
a	%	an integer
n	`MachineInteger`	a machine integer

Returns

n::% returns n converted to the current type, while machine(a) returns the low machine word of a converted to a `MachineInteger`. That operation can cause a loss of precision if a is greater than a machine word.

Usage

divide(a, b)
a mod n
a mod b
a quo b
a rem b

Signatures

divide:	(%,%) → (%, %)
mod,quo,rem:	(%,%) → %
mod:	(%,`MachineInteger`) → `MachineInteger`

Parameter	Type	Description
a,b	%	integers, $b \neq 0$
n	`MachineInteger`	a nonzero machine integer

Returns

a mod b (resp. a mod n) returns m such that $0 \leq m < |b|$ (resp. $0 \leq m < |n|$) and $a \equiv m \pmod{b}$ (resp n), while a rem b returns r such that $-|b| < r < |b|$ and $a \equiv r \pmod{b}$, a quo b returns $(a - (a \text{ rem } b))/b$, and divide(a, b) returns the pair (a quo b, a rem b).

Remarks

mod returns a unique remainder modulo b, but is more expensive to compute than rem, and is not guaranteed to be compatible with the result of quo. The version whose second argument is a `MachineInteger` allows for more efficient implementations.

Usage

even? a
odd? a

Signature

even?,odd?: % → `Boolean`

Parameter	Type	Description
a	%	an integer

Returns

even?(a) and odd?(a) return *true* when a is even, respectively odd, *false* otherwise.

Usage

factorial a

Signature

factorial: $\% \to \%$

Parameter	Type	Description
a	$\%$	a nonnegative integer

Returns

Returns $a! = \prod_{i=1}^{a} i$.

Usage

gcd(a, b)
lcm(a, b)

Signature

gcd,lcm: (%,%) → %

Parameter	Type	Description
a,b	%	integers

Returns

gcd(a, b) and lcm(a, b) return respectively a greatest common divisor and a least common multiple of a and b.

Usage

length a

Signature

length: % → `MachineInteger`

Parameter	Type	Description
a	%	an integer

Returns

Returns the number of binary bits of a, *i.e.*n such that `bit?`$(a, n-1)$ is *true* and `bit?`(a, m) is *false* for $m \geq n$.

Usage

next a
prev a

Signature

next,prev: $\% \to \%$

Parameter	Type	Description
a	$\%$	an integer

Returns

next(a) and prev(a) return $a+1$ and $a-1$ respectively.

Usage

nthRoot(a, b)

Signature

nthRoot: (%,%) → (`Boolean`,%)

Parameter	Type	Description
a	%	an integer
b	%	a positive integer

Returns

Returns (found?, n) such that $a = n^b$ if found? is *true*. Otherwise, found? is *false* and

$$n^b < a < (n+1)^b .$$

Usage

random()
random n

Signatures

random: () → %
random: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	a positive size

Returns

random() returns a random integer, while random(n) returns a random integer with n limbs.

Usage

shift(a, n)
shift!(a, n)

Signature

shift: (%, `MachineInteger`) → %

Parameter	Type	Description
a	%	an integer
n	`MachineInteger`	a machine integer

Returns

Returns a shifted left n times if $n \geq 0$, shifted right $-n$ times if $n \leq 0$.

Remarks

shift! does not make a copy of x, which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := shift!(x, n)`.

Usage

LinearCombinationType R:Category

Parameter	Type	Description
R	`AdditiveType`	The coefficient domain

Description

LinearCombinationType R is the category of types containing linear combinations of their elements with coefficients in R.

Exports

`AdditiveType`

$*$:	(R, %) → %	Left-multiplication by a scalar
`add!`:	(%, R, %) → %	In–place product and sum
`times!`:	(R, %) → %	In–place product by a scalar

Usage

r $*$ p

Signature

$*$: (R, %) $\to$ %

Parameter	Type	Description
r	R	A scalar
p	%	An element of the type

Returns

Returns the product *rp*.

See also

`times!`

Usage

add!(p, r, q)

Signature

add!: (%, R, %) → %

Parameter	Type	Description
p	%	An element of the type (to be destroyed)
r	R	A scalar
q	%	An element of the type

Returns

add!(p, r, q) returns $p + rq$.

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This function may cause p to be destroyed, so do not use it unless p has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

`times!`

Usage

times!(r, p)

Signature

times!: (R, %) → %

Parameter	Type	Description
r	R	A scalar to be multiplied by p
p	%	An element of the type (to be destroyed)

Returns

Returns the product *rp*.

Remarks

The storage used by p is allowed to be destroyed or reused, so p is lost after this call. This may cause p to be destroyed, so do not use this unless p has been locally allocated, and is thus guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

See also

∗,`add!`

Usage

import from MachineInteger

Description

MachineInteger implements the full-word signed machine integers.

Exports

`CopyableType`
`IntegerType`

`bytes`:	%	machine word-size
`max`:	%	largest machine integer
`min`:	%	smallest machine integer
`mod_+`:	(%, %, %) → %	modular addition
`mod_-`:	(%, %, %) → %	modular substraction
`mod_*`:	(%, %, %) → %	modular multiplication
`mod_/`:	(%, %, %) → %	modular division
`mod_^`:	(%, %, %) → %	modular exponentiation
`modInverse`:	(%, %) → %	modular inverse

Usage

bytes
max
min

Signature

bytes,max,min: %

Returns

bytes, max and min return respectively the size in bytes of a machine integer, the largest and the smallest machine integers.

Usage

mod_X(a, b, n)
modInverse(a, n)

Parameter	Type	Description
a, b, n	%	machine integers

Signatures

mod_X: (%, %, %) → %
modInverse: (%, %) → %

Returns

mod_X(a, b, n) returns $(aXb) \pmod{n}$ where X is one of $+, -, *, /, \hat{}$, while modInverse(a, b) returns the inverse of a modulo n.

Remarks

Those operations require that $0 \leq a, b < n$.

Usage

OrderedArithmeticType: Category

Description

OrderedArithmeticType is the category of ordered types with the standard arithmetic operations.

Exports

`ArithmeticType`
`TtotallyOrderedType`

`abs`:	$\% \to \%$	norm
`sign`:	$\% \to$ `MachineInteger`	sign

Usage

abs x

Signature

abs: $\% \to \%$

Parameter	Type	Description
x	$\%$	an element of the type

Returns

Returns the norm $|x|$ of x.

Usage

sign x

Signature

sign: % → `MachineInteger`

Parameter	Type	Description
x	%	an element of the type

Returns

Returns 1 if $x > 0$, 0 if $x = 0$ and -1 if $x < 0$.

Usage

PartiallyOrderedType: Category

Description

PartiallyOrderedType is the category of partially ordered types. Hence, a domain of this category is endowed with a transitive binary relation $>$ such that either $x > y$ or $y > x$ does not hold for every x and y. If both do not hold then x and y are called not comparable for $>$. In particular if $x = y$ holds then x and y are not omparable for $>$.

Exports

`PrimitiveType`

$<$:	(%, %) $\rightarrow$ `Boolean`	stricly less than
$>$:	(%, %) $\rightarrow$ `Boolean`	stricly greater than
$\leq$:	(%, %) $\rightarrow$ `Boolean`	less than or equal to
$\geq$:	(%, %) $\rightarrow$ `Boolean`	greater than or equal to

Usage

$a < b$
$a > b$
$a \leq b$
$a \geq b$

Signature

<,>: (%,%) → `Boolean`

Parameter	Type	Description
a, b	%	elements of the type

Returns

$a < b$, $a > b$, $a \leq b$, $a \geq b$ return *true* when a is respectively stricly smaller than, stricly greater than, less than or equal to, greater than or equal to b. Observe that if neither $a > b$ nor $a = b$ hold, this does not imply that $a < b$ holds, since a and b may not be comparable, except if the type has `TotallyOrderedType`.

Usage

PackableType: Category

Description

PackableType is the category of types providing their own primitive packed arrays over themselves. You can use `PackedPrimitiveArray`(T) as a packed alternative to `PrimitiveArray`(T) whenever T provides such functions.

Exports

`getPackedArray`:	(`Pointer`, `MachineInteger`) → %	access an array element
`newPackedArray`:	`MachineInteger` → `Pointer`	create an array
`setPackedArray!`:	(`Pointer`, `MachineInteger`, %) → ()	changes an array element

Usage

getPackedArray(a, n)
setPackedArray!(a, n, x)

Signatures

getPackedArray: (`Pointer`, `MachineInteger`) → %
setPackedArray!: (`Pointer`, `MachineInteger`, %) → ()

Parameter	Type	Description
a	`Pointer`	an array
n	`MachineInteger`	an index
x	%	a value

Description

getPackedArray(a, n) returns the element at position n in a, while setPackedArray!(a, n, x) sets the element at position n in a. Note that both functions are 0-indexed.

Usage

newPackedArray n

Signature

newPackedArray: `MachineInteger` → `Pointer`

Parameter	Type	Description
n	`MachineInteger`	a size

Returns

Returns an array for *n* elements

Usage

PrimitiveType: Category

Description

PrimitiveType is the category of the most basic types.

Exports

=:	(%, %) → `Boolean`	equality test
~=:	(%, %) → `Boolean`	inequality test

Usage

a = b
a ~= b

Signatures

=: (%,%) → `Boolean`
~=: (%,%) → `Boolean`

Parameter	Type	Description
a, b	%	elements of the type

Returns

If $a = b$ returns *true*, then a and b are guaranteed to represent the same element of the type. The behavior if $a = b$ returns *false* depends on the type, since a full equality test might not be available. At least, it is guaranteed that a and b do not share the same memory location in that case. The semantics of $a \tilde{} = b$ is the boolean negation of $a = b$.

Usage

import from RandomNumberGenerator

Description

RandomNumberGenerator provides independent pseudo-random number generators.

Exports

`apply`:	% → Z	generate a random number
`generator`:	% → `Generator` Z	generate random numbers
`max`:	% → Z	largest number that can be generated
`min`:	% → Z	smallest number that can be generated
`numberOfGenerators`:	Z	number of predefined generators
`randomGenerator`:	() → %	get a generator
`randomGenerator`:	Z → %	get a generator
`randomGenerator`:	(Z, Z) → %	get a bounded generator
`randomGenerator`:	(Z, Z, Z) → %	get a bounded generator
`randomInteger`:	() → Z	generate a random number
`seed`:	(%, Z) → Z	set the seed

where

Z == `MachineInteger`

Usage

apply r
r()

Signature

apply: % → `MachineInteger`

Parameter	Type	Description
r	%	a pseudo-random number generator

Returns

Returns a pseudo-random integer.

Usage

for n in r repeat { ... }
for n in generator r repeat { ... }

Signature

generator: % → `Generator MachineInteger`

Parameter	Type	Description
r	%	a pseudo-random number generator

Description

This functions allows a for-loop that generates infinitely many pseudo-random numbers.

Example

The following code computes the number of tries it takes for a random generator to generate a multiple of 10:

```
multiple10():MachineInteger == {
    r := randomGenerator();
    for n in r for tries in 1.. repeat {
                zero?(n rem 10) => return tries;
    }
    never;
}
```

Usage

max r
min r

Signature

max,min: % → `MachineInteger`

Parameter	Type	Description
r	%	a pseudo-random number generator

Returns

max(r) and min(r) return respectively the largest and smallest random integers that r can generate.

Usage
numberOfGenerators

Signature
numberOfGenerators: `MachineInteger`

Returns
Returns the number of independent generators provided.

Usage

randomGenerator()
randomGenerator n
randomGenerator(a, b)
randomGenerator(a, b, n)

Signatures

randomGenerator: () → %
randomGenerator: `MachineInteger` → %
randomGenerator: (`MachineInteger`, `MachineInteger`) → %
randomGenerator: (`MachineInteger`, `MachineInteger`, `MachineInteger`) → %

Parameter	Type	Description
a, b	`MachineInteger`	bounds for the generator
n	`MachineInteger`	identifier for the generator (optional)

Returns

randomGenerator() returns a pseudo-random generator, while randomGenerator(a, b) returns a pseudo-random number generator that generates numbers between a and b inclusive. If the optional argument n is given, then the n^{th} independent generator is returned, which allows for several independent sources of random numbers.

See also

`numberOfGenerators`

Usage

randomInteger()

Signature

randomInteger: () → `MachineInteger`

Returns

Returns a pseudo-random integer.

Usage

seed(r, s)

Signature

seed: (%, `MachineInteger`) → %

Parameter	Type	Description
r	%	a pseudo-random number generator
s	`MachineInteger`	a nonzero seed

Description

Sets the seed of r to s and returns s. This is useful when a reproducible pseudo-random sequence is desired. If an unseeded generator is called, then it seeds itself from the system clock the first time it is used, so the sequence is not reproducible. Note that setting the seed to 0 causes the generator to generate an infinite sequence of 0.

Usage

import from SingleFloat

Description

SingleFloat implements the single-precision signed machine floats.

Exports

`CopyableType`
`FloatType`
`PackableType`

$\hat{}$:	(%, %) → %	exponentiation
`max`:	%	largest single–precision machine float
`min`:	%	smallest single–precision machine float

Usage

$x \hat{} y$

Signatures

$\hat{}$: (%,%) → %

Parameter	Type	Description
x,y	%	floats

Returns

Returns x to the power y.

Usage

max
min

Signature

max,min: %

Returns

max and min return respectively the largest and the smallest single–precision machine floats.

Usage

TotallyOrderedType: Category

Description

TotallyOrderedType is the category of totally ordered types, *i.e.* partially ordered types where every pair $x \neq y$ is comparable.

Exports

`PartiallyOrderedType`

`max`: (%,%) → % greater element

`min`: (%,%) → % smaller element

Usage

max(x,y)
min(x,y)

Signature

max,min: (%,%) → %

Parameter	Type	Description
a, b	%	elements of the type

Returns

max(a,b) and min(a,b) respectively the largest and the smallest among a and b.

BinaryReader

Usage

import from BinaryReader

Description

BinaryReader provides various binary input streams.

Exports

`bin`:	%	standard input stream
`binaryReader`:	() → `Byte` → %	create a stream
`read!`:	% → `Byte`	read from a stream

Usage
bin

Signature
bin: %

Returns
bin is the standard input stream.

Usage

binaryReader f

Signature

binaryReader: (() → `Byte`) → %

Parameter	Type	Description
f	() → `Byte`	the single–byte read function

Returns

Returns the input stream for which *f*() reads a byte.

Usage

read! s

Signature

read!: % → `Byte`

Parameter	Type	Description
s	%	a stream

Returns

Reads a byte from *s* and returns it. Some streams may return the byte `eof` if the end of file is reached.

Usage

import from BinaryWriter

Description

BinaryWriter provides various binary output streams.

Exports

`berr`:	%	the binary standard error stream
`bout`:	%	the binary standard output stream
`binaryWriter`:	(`Byte` → ()) → %	create a stream
`binaryWriter`:	(`Byte` → (), () → ()) → %	create a stream
`flush!`:	% → %	flush a stream
`write!`:	(`Byte`, %) → ()	write to a stream

Usage

berr
bout

Signature

berr,bout: %

Returns

berr and bout are the binary standard error and standard output streams respectively.

Usage

binaryWriter f
binaryWriter(f, g)

Signatures

binaryWriter: (`Byte` → ()) → %
binaryWriter: (`Byte` → (), () → ()) → %

Parameter	Type	Description
f	`Byte` → ()	the single–byte write function
g	() → ()	the flush function (optional)

Returns

Returns the output stream for which $f(c)$ writes the byte c and such that $g()$ flushes the stream. If g is not given, then flushing the resulting stream has no effect.

Usage

flush! s

Signature

flush!: % → %

Parameter	Type	Description
s	%	a stream

Description

flush!(s) causes all previous values inserted into s to be really written and returns the stream. Has no effect on unbuffered streams, such as `berr`.

Usage

write!(c, s)

Signature

write!: (`Byte`, %) → ()

Parameter	Type	Description
c	`Byte`	byte to write
s	%	a stream

Returns

Writes the byte *c* to the stream *s* and returns *c*.

Usage

import from Byte

Description

Byte implements machine bytes.

Exports

`HashType`
`InputType`
`OutputType`
`PackableType`
`SerializableType`

`coerce`:	% → `MachineInteger`	conversion to an integer
`coerce`:	% → `Character`	conversion to a character
`coerce`:	`Character` → %	conversion from a character
`eof`:	%	end–of–file marker
`lowByte`:	`MachineInteger` → %	low–byte of an integer

Usage

b::`MachineInteger`
b::`Character`
c::%

Signatures

coerce: % → `MachineInteger`
coerce: % → `Character`
coerce: `Character` → %

Parameter	Type	Description
b	%	a byte
c	`Character`	a character

Returns

b::`MachineInteger` and b::`Character` return b converted to an integer and a character respectively, while c::% returns c converted to a byte.

Usage
eof

Signature
eof: %

Returns
eof is the end–of–file marker.

Usage

lowByte n

Signature

lowByte: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	an integer

Returns

Returns the low–byte of n.

Usage

import from Character

Description

Character implements machine characters.

Exports

`HashType`
`InputType`
`OutputType`
`PackableType`
`SerializableType`
`TtotallyOrderedType`

`char`:	`MachineInteger` → %	create a character
`digit?`:	% → `Boolean`	test for a decimal digit
`eof`:	%	end–of–file character
`letter?`:	% → `Boolean`	test for a letter
`lower`:	% → %	convert to lower case
`newline`:	%	newline character
`null`:	%	null character
`ord`:	% → `MachineInteger`	character code
`space?`:	% → `Boolean`	test for a blank space
`tab`:	%	tab character
`upper`:	% → %	convert to upper case

Usage

char n
ord c

Signatures

char: `MachineInteger` → %
ord: % → `MachineInteger`

Parameter	Type	Description
n	`MachineInteger`	a character code
c	%	a character

Returns

char(n) returns the character whose code is n, while ord(c) returns the code corresponding to the character c.

Usage

digit? c
letter? c
space? c

Signature

digit?,letter?,space?: % → `Boolean`

Parameter	Type	Description
c	%	a character

Returns

digit?(c) returns *true* if c is in the range '0'–'9', *false* otherwise, while letter?(c) returns *true* if c is in the range 'a'–'z' or the range 'A'–'Z', *false* otherwise and space?(c) returns *true* if c is a blank space, *i.e.*a space or a tab, *false* otherwise.

Usage

eof
newline
null
tab

Signature

eof,newline,null,tab: %

Returns

eof is the end–of–file character, newline is the newline character, null is the 0-character (used to terminate strings) and tab is the tab character.

Usage

lower c
upper c

Signature

lower,upper: $\% \to \%$

Parameter	Type	Description
c	%	a character

Returns

lower(c) and upper(c) return c converted to lower, respectively upper, case.

Usage

import from File

Description

File is a type whose elements are operating system files.

Exports

`close!`:	% → ()	close a file
`coerce`:	% → `BinaryReader`	conversion to a binary input stream
`coerce`:	% → `BinaryWriter`	conversion to a binary output stream
`coerce`:	% → `TextReader`	conversion to a text input stream
`coerce`:	% → `TextWriter`	conversion to a text output stream
`fileAppend`:	`MachineInteger`	mode for `open`
`fileBinary`:	`MachineInteger`	mode for `open`
`fileRead`:	`MachineInteger`	mode for `open`
`fileText`:	`MachineInteger`	mode for `open`
`fileWrite`:	`MachineInteger`	mode for `open`
`open`:	(`String`, `MachineInteger`) → %	open a file
`remove`:	`String` → ()	removes a file
`uniqueName`:	`String` → `String`	get a unique filename

Usage

close! f

Signature

close!: % → ()

Parameter	Type	Description
f	%	a file

Description

Closes the file f.

Usage

f::BinaryReader
f::BinaryWriter
f::TextReader
f::TextWriter

Signatures

coerce: % → `BinaryReader`
coerce: % → `BinaryWriter`
coerce: % → `TextReader`
coerce: % → `TextWriter`

Parameter	Type	Description
f	%	a file

Description

Converts the file f to to a binary or text reader or writer. This is necessary before reading from or writing to the file. The file must have been opened in an appropriate mode for reading or writing respectively.

Remarks

Coercing a file to a reader or writer allocates memory, so it is advisable to assign the resulting stream to a variable. Unlike the ones for `String`, those coercions do not reset the file to its beginning.

Usage

fileAppend
fileBinary
fileRead
fileText
fileWrite

Signature

fileAppend,fileBinary,fileRead,fileText,fileWrite: `MachineInteger`

Description

Those constants are for use in the mode parameter of the `open` function.

Usage

open(s,m)

Signature

open: (`String`, `MachineInteger`) → %

Parameter	Type	Description
s	`String`	a filename
m	`MachineInteger`	a mode (optional)

Description

Opens the file with the name s in the mode m, and returns the opened file. The mode is any combination of the constants `fileAppend`, `fileRead`, and `fileWrite`, together with one of `fileBinary` or `fileText`, grouped together with `+` or \/. The default is `fileRead` + `fileText`.

Remarks

open returns `nil` and throws the exception `FileException` if the file cannot be opened for any reason.

Usage

remove s

Signature

remove: `String` → ()

Parameter	Type	Description
s	`String`	A file name

Description

Removes the file with name s in the file system.

Usage

uniqueName s

Signature

uniqueName: `String` → `String`

Parameter	Type	Description
s	`String`	A root string

Returns

Returns a unique name with prefix s in the file system.

FileException

Usage

throw FileException
try ... catch E in { E has `FileExceptionType` =¿ ... }

Description

FileException is an exception type thrown by file operations.

FileExceptionType

Usage

FileExceptionType: Category

Description

FileExceptionType is the category of exceptions thrown by file operations.

InputType

Usage

InputType: Category

Description

InputType is the category of types whose objects can be read in in text format.

Exports

<<: `TextReader` → % read using text encoding

Usage

<< s

Signature

<<: `TextReader` → %

Parameter	Type	Description
s	`TextReader`	an input stream

Returns

<< s reads an element of the current type in text format from the stream s and returns the element read.

OutputType

Usage

OutputType: Category

Description

OutputType is the category of types whose objects can be written onto text writers.

Exports

<<: (`TextWriter`, %) → `TextWriter` write using text encoding

Usage

s << x

Signature

<<: (TextWriter, %) → TextWriter

Parameter	Type	Description
s	TextWriter	an output stream
x	%	an object of the type

Returns

writes x in text format to the stream s and returns s after the write.

Example

```
import from TextWriter, MachineInteger, Character;
stdout << 65 << space;
```

writes "65 " to the standard output stream.

Usage

SerializableType: Category

Description

SerializableType is the category of types whose objects can be read in and written out in binary mode.

Exports

<<:	`BinaryReader` → %	read using binary encoding
<<:	(`BinaryWriter`, %) → `BinaryWriter`	write using binary encoding

Usage

out << x
<< in

Signatures

<<: (`BinaryWriter`, %) → `BinaryWriter`
<<: `BinaryReader` → %

Parameter	Type	Description
in	`BinaryReader`	an input stream
out	`BinaryWriter`	an output stream
x	%	an object of the type

Returns

out << x writes x in binary format to the stream out and returns the stream after the write, while << in reads an element of the type in binary format from the stream in and returns the element read.

Usage

throw SyntaxException
try ... catch E in { E has `SyntaxExceptionType` =¿ ... }

Description

SyntaxException is an exception type thrown by read operations from a `TextReader`.

Usage

SyntaxExceptionType: Category

Description

SyntaxExceptionType is the category of exceptions thrown by read operations from a `TextReader`.

Usage

import from TextReader

Description

TextReader provides various input streams.

Exports

`push!`:	(`Character`, %) → ()	push back a character
`read!`:	% → `Character`	read from a stream
`stdin`:	%	standard input stream
`textReader`:	(() → `Character`, `Character` → ()) → %	create a stream

Usage

push!(c, s)

Signature

push!: (`Character`,%) → ()

Parameter	Type	Description
c	`Character`	a character
s	%	a stream

Returns

Pushes the character *c* back onto *s*. One character of pushback is guaranteed, this function might however fail if it is called too many times on the same stream without intervening read or positionning calls.

Usage

read! s

Signature

read!: % → `Character`

Parameter	Type	Description
s	%	a stream

Returns

Reads a character from *s* and returns it. Returns `eof` if the end of file is reached.

Usage
stdin

Signature
stdin: %

Returns
stdin is the standard input stream.

Usage

textReader(f, g)

Signature

textReader: (() → `Character`, `Character` → ()) → %

Parameter	Type	Description
f	() → `Character`	the single–character read function
g	`Character` → ()	the single–character pushback function

Returns

Returns the input stream for which $f()$ reads a character and $g(c)$ pushes back the character c.

Usage

import from TextWriter

Description

TextWriter provides various output streams.

Exports

`flush!`:	% → %	flush a stream
`stderr`:	%	the standard error stream
`stdout`:	%	the standard output stream
`textWriter`:	(`Character` → ()) → %	create a stream
`textWriter`:	(`Character` → (), () → ()) → %	create a stream
`write!`:	(`Character`, %) → ()	write to a stream

Usage

flush! s

Signature

flush!: % → %

Parameter	Type	Description
s	%	a stream

Description

flush!(s) causes all previous values inserted into s to be really written and returns the stream. Has no effect on unbuffered streams, such as `stderr`.

Usage

stderr
stdout

Signature

stderr,stdout: %

Returns

stderr and stdout are the standard error and standard output streams respectively.

Usage

textWriter f
textWriter(f, g)

Signatures

textWriter: (`Character` → ()) → %
textWriter: (`Character` → (), () → ()) → %

Parameter	Type	Description
f	`Character` → ()	the single–character write function
g	() → ()	the flush function (optional)

Returns

Returns the output stream for which $f(c)$ writes the character c and such that $g()$ flushes the stream. If g is not given, then flushing the resulting stream has no effect.

Usage

write!(c, s)

Signature

write!: (`Character`, %) → ()

Parameter	Type	Description
c	`Character`	character to write
s	%	a stream

Returns

Writes the character *c* to the stream *s* and returns *c*.

Usage

import from WriterManipulator

Description

WriterManipulator provides manipulators for text or binary writers.

Exports

<<:	(`BinaryWriter`, %) → `BinaryWriter`	manipulate a binary writer
`endnl`:	%	send a newline and flush the stream
`flush`:	%	flush the stream

Usage

out << x

Signature

<<: (`BinaryWriter`, %) → `BinaryWriter`

Parameter	Type	Description
out	`BinaryWriter`	an output stream
x	%	a manipulator

Returns

out << x takes the action given by x on the stream out and returns the stream after the action.

Usage
endnl

Signature
endnl: %

Description
Sending endnl to a text or binary writer causes a `newline` to be sent to the stream and then the stream to be flushed, so `s << endnl` is equivalent to `flush!(s << newline)`.

Usage
flush

Signature
flush: %

Description
Sending flush to a text or binary writer causes the stream to be flushed, so `s << flush` is equivalent to `flush!(s)`. Has no effect on unbuffered streams, such as `stderr`.

Array

Usage

import from Array T

Parameter	Type	Description
T	Type	the type of the array entries

Description

Array provides arrays of entries of type T, 0-indexed and without bound checking.

Exports

`ArrayType`(T, `PrimitiveArray` T)

Usage

throw ArrayException
try . . . catch E in { E has `ArrayExceptionType` =¿ . . . }

Description

ArrayException is an exception type thrown by array access.

ArrayExceptionType

Usage

ArrayExceptionType: Category

Description

ArrayExceptionType is the category of exceptions thrown by array access.

Usage

ArrayType(T, P): Category

Parameter	Type	Description
T	Type	the type of the array entries
P	`PrimitiveArrayType` T	the type of the underlying data

Description

ArrayType is the category of arrays whose entries are of type T and whose underlying data is of type P.

Exports

`BoundedFiniteLinearStructureType` T

`array`:	(P, `MachineInteger`) → %	construction of an array
`data`:	% → P	access to raw data
`new`:	`MachineInteger` → %	creation of an array
`resize!`:	(%, `MachineInteger`) → %	resize an array
`sort!`:	(%, (T, T) → `Boolean`) → %	sort an array

if T has `TotallyOrderedType` then

`TotallyOrderedType`

`binarySearch`:	(T, %) → (`Boolean`, `MachineInteger`)	binary search
`sort!`:	% → %	sort an array

Usage

array(p, n)

Signature

array: (P, `MachineInteger`) → %

Parameter	Type	Description
p	P	a primitive array structure
n	`MachineInteger`	a number of elements

Returns

Returns an array containing the first n entries of p. No copying is made.

Usage

binarySearch(t, a)

Signature

binarySearch: (T, %) → (`Boolean`, `MachineInteger`)

Parameter	Type	Description
t	T	the value to search for
a	%	an array

Returns

Returns (found?, i) such that $0 \leq i < \#a$ and $t = a.i$ if found? is *true*. Otherwise, found? is *false* and:

- if $i < 0$ then $t < a.0$;
- if $i \geq \#a - 1$ then $t > a(\#a - 1)$;
- if $0 \leq i < \#a - 1$, then $a.i < t < a(i + 1)$.

The array a must be sorted in increasing order. If a is sorted with respect to a different order, it is still possible to use binary search, but from `BinarySearch`.

See also

`linearSearch`

Usage

data a

Signature

data: % → P

Parameter	Type	Description
a	%	an array

Returns

Returns the raw data of the array a. No copying is made. This function can be used for efficiency before accessing the elements of a inside a loop, or in order to pass the elements of a to a C function.

Usage

new n

Signature

new: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	a nonnegative size

Returns

Returns an array of n uninitialized entries.

Usage

resize!(a, n)

Signature

resize!: (%,`MachineInteger`) → %

Parameter	Type	Description
a	%	an array
n	`MachineInteger`	a nonnegative size

Returns

Returns an array of n entries, whose first $\#a$ entries are the first $\#a$ entries of a and whose remaining entries are uninitialized.

Usage

sort! a
sort!(a, f)

Signature

sort!: (%,(T,T) → `Boolean`) → %

Parameter	Type	Description
a	%	a primitive array
f	(T, T) → `Boolean`	a comparison function

Description

Sorts the array a using the ordering $x < y \iff f(x, y)$. The comparison function f is optional if T has `TotallyOrderedType`, in which case the order function of T is taken.

Usage

BoundedFiniteDataStructureType T: Category

Parameter	Type	Description
T	`Type`	the type of the entries

Description

BoundedFiniteDataStructureType is the category of finite general structures whose entries are of type T and whose size is always known.

Exports

`CopyableType`
`DataStructureType`

`#`:	% → `MachineInteger`	number of entries
`generator`:	% → `Generator` T	iteration over a structure

if T has `PrimitiveType` then

`findAll`:	(T, %) → `Generator Cross(MachineInteger`, T)	linear search
`member?`:	(T, %) → `Boolean`	look for a value

if T has `HashType` then
`HashType`

if T has `OutputType` then
`OutputType`

Usage

a

Signatures

#: % → `MachineInteger`

Parameter	Type	Description
a	%	a finite data structure

Returns

Returns the number of entries in the structure *a*.

Usage

for pair in findAll(t, a) repeat { (pos, val) := pair; ... }

Signature

findAll: (T, %) → `Generator Cross(MachineInteger`, T)

Parameter	Type	Description
t	T	the value to search for
a	%	a finite data structure

Description

Iterates trough all pairs (i, x) such that $t = x$ (using the equality of the type T). The index i is the position of x in the iteration of t by the function `generator`: $i = 1$ means x is the first element generated, $i = 2$ means x is the second element generated, etc.

See also

`member?`

Usage

for x in a repeat { ... }
for x in generator a repeat { ... }

Signature

generator: % → `Generator` T

Parameter	Type	Description
a	%	a finite data structure

Description

This function allows a structure to be iterated independently of its representation. This generator yields the elements of *a* in some order, which is determined by the actual type.

Example

The following code computes the sum of all the elements of an array of machine integers:

```
sum(a:Array MachineInteger, n:MachineInteger):MachineInteger == {
    s:MachineInteger := 0;
    for x in a repeat s := s + x;
    s;
}
```

Usage

member?(t, a)

Signature

member?: (T, %) → `Boolean`

Parameter	Type	Description
t	T	the value to search for
a	%	a finite data structure

Returns

Returns *true* if t is a member of a, *false* otherwise.

Usage

BoundedFiniteLinearStructureType T: Category

Parameter	Type	Description
T	`Type`	the type of the entries

Description

BoundedFiniteLinearStructureType is the category of finite linear structures whose entries are of type T and whose size is always known.

Exports

`BoundedFiniteDataStructureType` T
`FiniteLinearStructureType` T
`map`: (T → T) → % → % lift a mapping
`map!`: (T → T) → % → % lift a mapping

if T has `PrimitiveType` then
`PrimitiveType`
`linearSearch`: (T, %) → (`Boolean`, `MachineInteger`, T) linear search
(T, %, `MachineInteger`) → (`Boolean`, `MachineInteger`, T)

if T has `InputType` then
`InputType`

if T has `SerializableType` then
`SerializableType`

Usage

linearSearch(t, a)
linearSearch(t, a, n)

Signature

linearSearch: (T, %, `MachineInteger`) → `Boolean`, `MachineInteger`, T

Parameter	Type	Description
t	T	the value to search for
a	%	a finite linear structure
n	`MachineInteger`	an initial index (optional)

Returns

Returns (found?, i, a.i) such that $t = a.i$ if found? is *true*, t is not in a otherwise. If the optional argument n is present, then the search starts at the entry $a.n$ and ignores the previous ones.

See also

`findAll`

Usage

map f
map! f
map(f)(a)
map!(f)(a)

Signature

map: (T → T) → % → %

Parameter	Type	Description
f	T → T	a map
a	%	a finite linear structure

Returns

map(f)(a) returns the new structure `[f(x) for x in a]`, while map(f) returns the mapping *a* → `[f(x) for x in a]`. In both cases, map! does not make a copy of the structure a but modifies it in place.

Usage

import from CheckingArray T

Parameter	Type	Description
T	Type	the type of the array entries

Description

CheckingArray provides arrays of entries of type T, 0-indexed and with bound checking.

Exports

`ArrayType`(T, `PrimitiveArray` T)

Remarks

The functions `apply` and `set!` throw the exception `ArrayException` when attempting to access an array out of its bounds.

Usage

import from CheckingList T

Parameter	Type	Description
T	Type	the type of the list entries

Description

CheckingList provides lists of entries of type T, 1-indexed and with bound checking.

Exports

`ListType` T

Usage

import from CheckingMemoryBlock

Description

CheckingMemoryBlock provides packed arrays of bytes, 0-indexed and with bound checking.

Exports

`ArrayType`(`Byte`, `PrimitiveMemoryBlock`)

Remarks

The functions `apply` and `set!` throw the exception `ArrayException` when attempting to access a memory block out of its bounds.

DataStructureType

Usage

DataStructureType: Category

Description

DataStructureType is the category of general data structures, not necessarily finite or linear.

Exports

`empty?`:	% → `Boolean`	test whether a structure is empty
`free!`:	% → ()	memory disposal

Usage

empty? a

Signature

empty?: % → `Boolean`

Parameter	Type	Description
a	%	a data structure

Returns

Returns *true* if *a* contains no element, *false* otherwise.

Usage

free! a

Signature

free!: % → ()

Parameter	Type	Description
a	%	a data structure

Description

Releases the memory occupied by *a*.

Usage

DynamicDataStructureType T: Category

Parameter	Type	Description
T	`Type`	the type of the entries

Description

DynamicDataStructureType is the category of finite general structures in which entries of type T can be inserted or removed dynamically.

Exports

`BoundedFiniteDataStructureType` T

`insert`: (T, %) → % add an element
`insert!`: (T, %) → % add an element

if T has `PrimitiveType` then

`remove`: (T, %) → % remove an element
`remove!`: (T, %) → % remove an element
`removeAll`: (T, %) → % remove all occurrences of an element
`removeAll!`: (T, %) → % remove all occurrences of an element

Usage

insert(t, a)
insert!(t, a)

Signature

insert: (T, %) → %

Parameter	Type	Description
t	T	an element to add
a	%	a dynamic data structure

Description

Adds *t* to *a* and returns the new structure. insertcreates a new structure while insert! modifies *a* itself.

Usage

remove(t, a)
remove!(t, a)
removeAll(t, a)
removeAll!(t, a)

Parameter	Type	Description
t	T	an element to remove
a	%	a dynamic data structure

Description

remove(t, a) and remove!(t, a) remove the first occurence of *t* in *a* and return the new structure, while removeAll(t, a) and removeAll!(t, a) remove all the occurences of *t* in *a*. remove and removeAll create a new structure, while remove! and removeAll! modify *a* itself.

Usage

FiniteLinearStructureType T: Category

Parameter	Type	Description
T	Type	the type of the entries

Description

FiniteLinearStructureType is the category of finite linear structures whose entries are of type T.

Exports

`LinearStructureType` T

`[]`:	`Tuple` T → %	construction of a structure
`empty`:	%	empty structure
`new`:	(`MachineInteger`, T) → %	creation of a structure

Usage

$[t_1, \ldots, t_n]$

Signature

[]: `Tuple` T $\rightarrow$ %

Parameter	Type	Description
$t_1, \ldots, t_n$	T	elements of T

Returns

Returns the structure $[t_1, \ldots, t_n]$.

Usage
empty

Signature
empty: empty

Returns
Returns an empty structure.

See also
`empty?`

Usage

new(n, x)

Signature

new: (`MachineInteger`, T) → %

Parameter	Type	Description
n	`MachineInteger`	a nonnegative size
x	T	an entry

Returns

Returns a structure of n entries, all of them set to x.

Usage

import from HashTable(K, V)
import from HashTable(K, V, h)

Parameter	Type	Description
K	`PrimitiveType` `HashType`	the type of the keys
V	Type	the type of the entries
h	K → `MachineInteger`	the hash function to use

Description

HashTable provides hash tables with keys of type K, entries of type V and that uses the hash–function h. If K has `HashType`, then the parameter h is optional as the function `hash` is used by default in that case.

Exports

`TableType`(K, V)

Usage

import from KeyEntry(K, V)

Parameter	Type	Description
K	`Type`	the type of the keys
V	`Type`	the type of the entries

Description

KeyEntry(K, V)provides pairs consisting of a key from K, and a entry from V. When K is a `PrimitiveType`, then two pairs are equal if they share the same key. When K is a `TotallyOrderedType`, then the pair x is greater than the pair y if $key(x) > key(y)$. Hence key-entry pairs are useful for building tables of entries where each slot is given by a unique key.

Exports

`CopyableType`

`[]`:	(K, V) → %	construction of a key-entry pair
`entry`:	% → V	get the entry
`explode`:	% → (K, V)	get the key and the entry
`key`:	% → K	get the key
`free!`:	% → ()	memory disposal
`setEntry!`:	(%, V) → V	change the entry
`setKey!`:	(%, K) → K	change the key

if K has `PrimitiveType` then
`PrimitiveType`

if K has `TotallyOrderedType` then
`TotallyOrderedType`

Usage

[*k*,*v*]

Signature

[]: (K, V) → %

Parameter	Type	Description
k	K	a key
v	V	an entry

Returns

Returns the key-entry pair [*k*, *v*].

Usage

entry p
(k, v) := explode p
key p

Signatures

entry:	% → V
explode:	% → (K, V)
key:	% → K

Parameter	Type	Description
p	%	a key-entry pair

Returns

entry(p) and key(p) return respectively the entry and key of *p*, while explode(p) returns the pair (key p, entry p).

Usage

setEntry!(p, v)
setKey!(p, k)

Signatures

setEntry!: (%, V) → V
setKey!: (%, K) → K

Parameter	Type	Description
p	%	a key-entry pair
k	K	a key
v	V	an entry

Description

setEntry!(p, v) (resp. setKey!(p, k)) changes the entry (resp. key) of p to v (resp. k) and returns v (resp. k).

HashType

Usage

HashType: Category

Description

HashType is the category of types whose objects can be hashed into machine integers.

Exports

`PrimitiveType`

`hash`: $\% \to$ `MachineInteger` hash function

Usage

hash x

Signature

hash: % → `MachineInteger`

Parameter	Type	Description
x	%	an element of the type

Returns

Returns a hash–code for x.

Usage

LinearStructureType T: Category

Parameter	Type	Description
T	Type	the type of the entries

Description

LinearStructureType is the category of linear structures whose entries are of type T.

Exports

`DataStructureType`

`[]`:	`Generator` T → %	construction of a structure
`apply`:	(%, `MachineInteger`) → T	extraction of an entry
`firstIndex`:	`MachineInteger`	index of first element
`set!`:	(%, `MachineInteger`, T) → T	modification of an entry

if T has `PrimitiveType` then

`equal?`:	(%, %, `MachineInteger`) → `Boolean`	compare the first n elements

Usage

[t for t in g]

Signature

[]: `Generator` T → %

Parameter	Type	Description
g	`Generator` T	an iterator producing elements of T

Returns

Returns the structure composed of all the elements generated by g in the order in which they are generated.

Usage

apply(a, n)
a.n

Signature

apply: (%, `MachineInteger`) → T

Parameter	Type	Description
a	%	a linear structure
n	`MachineInteger`	an index

Returns

Returns the element of a with index n. The position of that element depends on the indexing scheme of each actual type, for example if it is 0-indexed, then $a.n$ returns the $(n+1)^{\text{st}}$ element of a, while if it is 1-indexed, then $a.n$ returns the n^{th} element of a.

Remarks

Bound checking depends on the actual type, some perform it and some do not.

Usage
firstIndex

Signature
firstIndex: `MachineInteger`

Returns
Returns the index of the first element of a structure.

Usage

set!(a, n, x)
a.n := x;

Signature

set!: (%, `MachineInteger`, T) → T

Parameter	Type	Description
a	%	a linear structure
n	`MachineInteger`	an index
x	T	an entry

Description

Sets the element of a with index n to x and returns x. The position of that element depends on the indexing scheme of each actual type, for example if it is 0-indexed, then $a.n := x$ sets the $(n+1)^{\text{st}}$ element of a, while if it is 1-indexed, then $a.n := x$ sets the n^{th} element of a.

Remarks

Bound checking depends on the actual type, some perform it and some do not.

Usage

equal?(a, b, n)

Signature

equal?: (%, %, `MachineInteger`) → `Boolean`

Parameter	Type	Description
a, b	%	linear structures
n	`MachineInteger`	a nonnegative integer

Returns

Returns *true* if the first n elements of a and b are all equal, *false* otherwise.

Remarks

a and b must both have at least n elements.

Usage

import from List T

Parameter	Type	Description
T	Type	the type of the list entries

Description

List provides lists of entries of type T, 1-indexed and without bound checking.

Exports

`ListType` T

ListException

Usage

throw ListException
try . . . catch E in { E has `ListExceptionType` =¿ . . . }

Description

ListException is an exception type thrown by list access.

ListExceptionType

Usage

ListExceptionType: Category

Description

ListExceptionType is the category of exceptions thrown by list access.

Usage

ListType T: Category

Parameter	Type	Description
T	Type	the type of the list entries

Description

ListType is the category of lists of entries of type T.

Exports

`BoundedFiniteLinearStructureType` T
`DynamicDataStructureType` T

+:	(%, `MachineInteger`) → %	translate the base
`append!`:	(%, T) → %	adds an entry at the end
`append!`:	(%, %) → %	adds a list at the end
`cons`:	(T, %) → %	adds an entry at the front
`delete!`:	(%, `MachineInteger`) → %	remove an entry
`first`:	% → T	first entry
`rest`:	% → %	all entries after the first
`reverse`:	% → %	reverse a list
`reverse!`:	% → %	reverse a list in–place
`setFirst!`:	(%, T) → T	changes the first element of a list
`setRest!`:	(%, %) → %	changes the rest of a list
`sort!`:	(%, (T, T) → `Boolean`) → %	sort a list

if T has `PrimitiveType` then
`find`: (T, %) → (%, `MachineInteger`) linear search

if T has `TotallyOrderedType` then
`TotallyOrderedType`
`sort!`: % → % sort a list

Usage

l + n

Signature

+: (%, `MachineInteger`) → %

Parameter	Type	Description
l	%	a list
n	`MachineInteger`	an index

Returns

l+n returns the sublist of l starting at the $(n+1)^{\text{st}}$ element of l, *i.e.*$l+0$ returns l, $l+1$ returns the subslist starting at the second element of l, etc.... No copy of l is made.

Example

If l is the list of `MachineInteger` $[1,2,3,4,5]$, then $l+2$ is the list $[3,4,5]$.

Usage

append!(l, x)
append!(l, s)

Signatures

append!: (%, T) → %
append!: (%, %) → %

Parameter	Type	Description
l,s	%	lists
x	T	an entry

Returns

append!(l,x) and append!(l,s) return the lists $[l, x]$ and $[l, s]$ respectively.

Remarks

append! does not make a copy of l, which is therefore modified after the call. It is unsafe to use the variable l after the call, unless it has been assigned to the result of the call, as in `l := append!(l, x)`.

See also

`cons`

Usage

cons(x, l)

Signature

cons: (T, %) → %

Parameter	Type	Description
x	T	an entry
l	%	a list

Returns

Returns the list $[x, l]$. Does not make a copy of l.

See also

`append!`

Usage

delete!(l, n)

Signature

delete!: (%, `MachineInteger`) → %

Parameter	Type	Description
l	%	a list
n	`MachineInteger`	an index

Returns

Returns the list l with $l.n$ removed. Does not make a copy of l.

Usage

find(t, l)

Signature

find: (T, %) → (%, `MachineInteger`)

Parameter	Type	Description
t	T	the value to search for
l	%	a list

Returns

Returns (b, i) such that:

- if b is not `empty`, then $l.i = t$ is the first occurence of t in l, and b is the sublist of l starting at $l.i$,
- if b is `empty`, then i is undefined and t does not occur in l.

The list a does not need to be sorted, and the complexity is expected to be linear in its size.

Remarks

No copy of l is made: if b is not `empty`, then its data is shared with l. This function allows all the occurences of t to be found successively in a list, as in the example below.

Example

If `l1` is the list of `MachineInteger` [1,6,2,5,3,7,2,4], then `(l2, n) := find(2, l1)` sets `l2` to [2,5,3,7,2,4] and `n` to 3, the further call `(l3, n) := find(2, rest l2)` sets `l3` to [2,4] and `n` to 4, and the further call `(l4, n) := find(2, rest l3)` sets `l4` to `empty`.

Usage

first l

Signature

first: % → T

Parameter	Type	Description
l	%	a nonempty list

Returns

Returns the first entry of *l*.

Usage

rest l

Signature

rest: % → %

Parameter	Type	Description
l	%	a nonempty list

Returns

Returns *l* with the first element removed. Does not make a copy of *l*.

See also

`setRest!`

Usage

reverse l
reverse! l

Signature

reverse: % → %

Parameter	Type	Description
l	%	a list

Returns

reverse(l) returns a copy of *l* with the elements in reverse order, while reverse!(l) reverses *l* without copying it.

Remarks

reverse! does not make a copy of *l*, which is therefore modified after the call. It is unsafe to use the variable *l* after the call, unless it has been assigned to the result of the call, as in `l := reverse! l`.

Usage
setFirst!(l,t)

Signature
setFirst!: (%,T) → T

Parameter	Type	Description
l	%	a nonempty list
t	%	an element

Description
Replaces the first element of *l* by *t* and returns *t*.

Remarks
setFirst!(l,t) does not make a copy of *l*, which is therefore modified after the call.

See also
`setRest!`

Usage
setRest!(l,t)

Signature
setRest!: (%,%) → %

Parameter	Type	Description
l	%	a nonempty list
t	%	a list

Description
Replaces l by the list `[first(l),t]` and returns t.

Remarks
setRest!(l,t) does not make a copy of *l*, which is therefore modified after the call.

See also
`setFirst!`

Usage

sort! l
sort!(l, f)

Signature

sort!: (%,(T,T) → `Boolean`) → %

Parameter	Type	Description
l	%	a list
f	(T, T) → `Boolean`	a comparison function

Description

Sorts the list a using the ordering $x < y \iff f(x, y)$. The comparison function f is optional if T has `TotallyOrderedType`, in which case the order function of T is taken.

Remarks

sort! does not make a copy of l, which is therefore modified after the call. It is unsafe to use the variable l after the call, unless it has been assigned to the result of the call, as in `l := sort! l`.

Usage

import from MemoryBlock

Description

MemoryBlock provides packed arrays of bytes, 0-indexed and without bound checking (the debug version of `libaldor` provides bound–checking for memory blocks).

Exports

`ArrayType`(`Byte`, `PrimitiveMemoryBlock`)

Usage

import from PackedPrimitiveArray T

Parameter	Type	Description
T	`PackableType`	the type of the array entries

Description

PackedPrimitiveArray provides packed arrays of entries of type T, 0-indexed and without bound checking (the debug version of `libaldor` provides bound–checking for packed primitive arrays).

Exports

`PrimitiveArrayType` T

Usage

import from PrimitiveArray T

Parameter	Type	Description
T	Type	the type of the array entries

Description

PrimitiveArray provides arrays of entries of type T, 0-indexed and without bound checking (the debug version of `libaldor` provides bound–checking for primitive arrays).

Exports

`PrimitiveArrayType` T

Usage

import from PrimitiveArrayType T

Parameter	Type	Description
T	Type	the type of the array entries

Description

PrimitiveArrayType is the category of primitive arrays whose entries are of type T.

Exports

`FiniteLinearStructureType` T

`array`:	(`Pointer`, Z) → %	conversion from a C–array
`new`:	Z → %	creation of an array
`pointer`:	% → `Pointer`	conversion to a C–array
`resize!`:	(%, Z, Z) → %	resize an array
`sort!`:	(%, Z, Z, (T, T) → `Boolean`) → %	sort an array

if T has `SerializableType` then

`read`:	(`BinaryReader`, Z) → %	read using binary encoding
`write`:	(`BinaryWriter`, write using binary encoding	

if T has `TotallyOrderedType` then

`sort!`:	(%, Z, Z) → %	sort an array

where

Z == `MachineInteger`

Usage

array(p, n)
pointer a

Signatures

array: (`Pointer`, `MachineInteger`) → %
pointer: % → `Pointer`

Parameter	Type	Description
p	`Pointer`	a C–array
n	`MachineInteger`	a nonnegative size
a	%	A primitive array

Description

Use those functions to safely convert between pointers returned or needed by C–functions and primitive arrays. Those function have no effect in the release version of `libaldor`, but they are necessary when using the debug version, so it is recommended to use them in all cases.

Usage

new n

Signature

new: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	a nonnegative size

Returns

Returns a primitive array of n uninitialized entries.

Usage

read(in, n)
write(out, a, n)

Signatures

read: (`BinaryReader`, `MachineInteger`) → %
write: (`BinaryWriter`, %, `MachineInteger`) → `BinaryWriter`

Parameter	Type	Description
in	`BinaryReader`	an input stream
out	`BinaryWriter`	an output stream
n	`MachineInteger`	a nonnegative size
a	%	A primitive array

Returns

read(in, n) reads a primitive array of n elements in binary format from the stream in and returns the array read, while write(out, a, n) writes the first n elements of a to the stream out and returns the stream after the write.

Usage

resize!(a, n, m)

Signature

resize!: (%,`MachineInteger`,`MachineInteger`) → %

Parameter	Type	Description
a	%	a primitive array
n, m	`MachineInteger`	nonnegative sizes

Returns

Returns a primitive array of m entries, whose first n entries are the first n entries of a and whose remaining entries are uninitialized.

Remarks

resize! may free the space previously used by a, so it is unsafe to use the variable a after the call, unless it has been assigned to the result of the call, as in `a := resize!(a, n, m)`.

Usage

sort!(a, n, m)
sort!(a, n, m, f)

Signature

sort!: (%,`MachineInteger`,`MachineInteger`, (T,T) → `Boolean`) → %

Parameter	Type	Description
a	%	a primitive array
n, m	`MachineInteger`	indices
f	(T, T) → `Boolean`	a comparison function

Description

Sorts the subarray $[a.n, \ldots, a.m]$ using the ordering $x < y \iff f(x, y)$. The comparison function f is optional if T has `TotallyOrderedType`, in which case the order function of T is taken.

Usage

import from PrimitiveMemoryBlock

Description

PrimitiveMemoryBlock provides packed arrays of bytes, 0-indexed and without bound checking (the debug version of `libaldor` provides bound–checking for primitive memory blocks).

Exports

`PrimitiveArrayType Byte`

`coerce`:	% → `BinaryReader`	conversion to a binary input stream
`coerce`:	% → `BinaryWriter`	conversion to a binary output stream

Usage

a::BinaryReader
a::BinaryWriter

Signatures

coerce: % → `BinaryReader`
coerce: % → `BinaryWriter`

Parameter	Type	Description
a	%	a primitive memory block

Description

a::T where T is an I/O stream type converts the block s to a binary reader or writer, allowing one to read data or write data to it.

Remarks

When writing to a memory block, you must ensure that the block is large enough for all the data that will be written to it, since the block will not be extended and this function does not protect you against overwriting. When reading from or writing to a memory block, each coercion to a reader or writer resets the stream to the beginning of the block, and the block is not side–affected by the subsequent read or write operations, while the stream is side–affected. Thus, when reading several values from the same block, you must assign the reader to a variable and read the values from that variable.

See also

`coerce`

.

Usage

import from Set T

Parameter	Type	Description
T	`PrimitiveType`	the type of the set entries

Description

Set provides sets of entries of type T, 1-indexed and without bound checking.

Exports

`BoundedFiniteLinearStructureType` T

`DynamicDataStructureType` T

−:	(%, %) → %	set difference
`intersection`:	(%, %) → %	intersect two sets
`intersection!`:	(%, %) → %	intersect two sets
`minus!`:	(%, %) → %	set difference
`union`:	(%, T) → %	add an element
	(%, %) → %	add a set of elements
`union!`:	(%, T) → %	add an element
	(%, %) → %	add a set of elements

Usage

$x - y$

Signature

$-$: $(\%, \%) \to \%$

Parameter	Type	Description
x,y	$\%$	sets

Returns

Return $x - y = \{a \in x \text{ such that } a \notin y\}$.

See also

`minus!`

Usage

intersection(x,y)
intersection!(x,y)

Signature

intersection: (%, %) → %

Parameter	Type	Description
x,y	%	sets

Returns

Return $x \cap y = \{a \text{ such that } a \in x \text{ and } a \in y\}$.

Remarks

intersection! does not make a copy of x, which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := intersection!(x, y)`.

Usage

minus!(x,y)

Signature

minus!: (%, %) → %

Parameter	Type	Description
x,y	%	sets

Description

Return $x - y = \{a \in x \text{ such that } a \notin y\}$.

Remarks

minus! does not make a copy of x, which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := minus!(x, y)`.

See also

–

Usage

union(x,t)
union(x,y)
union!(x,t)
union!(x,y)

Signature

union: (%, %) → %

Parameter	Type	Description
x,y	%	sets
t	T	an element

Returns

union(x,y) and union!(x,y) both return $x \cup y = \{a \text{ such that } a \in x \text{ or } a \in y\}$, while union(x,t) and union!(x,t) both return $x \cup \{t\}$.

Remarks

union! does not make a copy of x, which is therefore modified after the call. It is unsafe to use the variable x after the call, unless it has been assigned to the result of the call, as in `x := union!(x, y)`.

Usage

import from SortedAssociationSet(K, V)

Parameter	Type	Description
K	`TotallyOrderedType`	the type of the keys
V	`Type`	the type of the entries

Description

SortedAssociationSet(K, V) implements sorted sets of key-entry pairs. The keys come from K with a total ordering and the entries come from T which is any type. A sorted association set can be viewed as a hash-table with the identity as hash-function.

Exports

`TableType`(K, V)

Usage

import from SortedList T

Parameter	Type	Description
T	`PartiallyOrderedType`	the type of the entries

Description

SortedList(T) provides sorted lists whose entries belong to a partially ordered type. Duplicate entries are allowed.

Exports

`BoundedFiniteLinearStructureType` T
`DynamicDataStructureType` T

Usage

import from SortedSet T

Parameter	Type	Description
T	`PartiallyOrderedType`	the type of the entries

Description

SortedSet(T) provides sorted sets whose entries belong to a partially ordered type. Duplicate entries are not allowed.

Exports

`BoundedFiniteLinearStructureType` T
`DynamicDataStructureType` T

Usage

import from Stream T

Parameter	Type	Description
T	Type	the type of the stream entries

Description

Stream provides streams of entries of type T, 0-indexed and with bound checking.

Exports

`LinearStructureType` T

`#`:	% → Z	number of computed elements
`constant`:	% → (Z, `Partial` T)	check for an eventually constant stream
`generator` :	% → `Generator` T	iteration over a stream
`map!`:	(T → T) → % → %	lift a mapping
`orbit`:	(T, T → T) → %	creation of a stream
`stream`:	T → %	creation of a stream
	(() → T) → %	
	((Z, T) → T) → %	
	(Z, Z → T) → %	
	(Z, Z → T, Z, T) → %	
	(`Generator` T, T) → %	
	(`Generator` T, Z, T) → %	

where

Z == `MachineInteger`

if T has `OutputType` then

`OutputType`

Usage

s

Signatures

#: % → MachineInteger

Parameter	Type	Description
s	%	a stream

Returns

Returns the number of elements of *s* that have been computed.

Usage

constant s

Signature

constant: % → (`MachineInteger`, `Partial` T)

Parameter	Type	Description
s	%	a stream

Returns

Returns $(-1, \mathit{failed})$ if s is not known to be eventually constant, otherwise returns $(n \geq 0, [t])$ such that $s.k = s.n = t$ for $k \geq n$.

Usage

for x in s repeat { ... }
for x in generator s repeat { ... }

Signature

generator: % → `Generator` T

Parameter	Type	Description
s	%	a stream

Description

This function allows the elements of a stream to be iterated.

Remarks

Since those generators are infinite, you should have a termination condition either inside the loop or in parallel with the generator in order to guarantee termination.

Example

The following code creates the stream of all the squares for $n \geq 0$ and prints those of them that are smaller than 1000:

```
import from MachineInteger, Stream MachineInteger;
s := stream(0, (n:MachineInteger):MachineInteger +-> n^2);
for x in s while x < 1000 repeat { stdout << x << newline; }
```

Usage

map! f
map!(f)(s)

Signature

map!: $(T \to T) \to \% \to \%$

Parameter	Type	Description
f	$T \to T$	a map
s	%	a stream

Returns

map!(f)(s) returns the stream `[f(x) for x in s]`, while map!(f) returns the mapping $s \to$ `[f(x) for x in s]`. In both cases, the stream s is modified in place and no copy is made.

Usage

orbit(t, f)

Signature

orbit: (T, T → T) → %

Parameter	Type	Description
t	T	an element of T
f	T → T	a function

Returns

Returns the stream $[t, f(t), f^2(t), \ldots]$.

See also

`stream`

Usage

stream t
stream f
stream h
stream(n, g)
stream(n, g, m, t)
stream(gen, t)
stream(gen, m, t)

Signatures

stream: T → %
stream: (() → T) → %
stream: ((`MachineInteger`, %) → T) → %
stream: (`MachineInteger`, `MachineInteger` → T) → %
stream: (`MachineInteger`, `MachineInteger` → T, `MachineInteger`, T) → %
stream: (`Generator` T, T) → %
stream: (`Generator` T, `MachineInteger`, T) → %

Parameter	Type	Description
n, m	`MachineInteger`	machine integers
t	T	an element
f	() → T	a function
g	`MachineInteger` → T	a function
h	(`MachineInteger`, %) → T	a function
gen	`Generator` T	a generator

Description

stream(t) returns the constant stream $[t, t, t, \ldots]$, while stream(f) returns the stream obtained by successive calls to $f()$, stream(n, g) returns the stream $[g(n), g(n+1), g(n+2), \ldots]$, stream(n, g, m, t) returns the eventually constant stream $[g(n), g(n+1), g(n+2), \ldots, g(m-1), t, t, t, \ldots]$ and stream(h) returns the stream $[s_0, s_1, s_2, \ldots]$ where $s_k = h(k, s_0, \ldots s_{i-k})$ for any $k \geq 0$. When using generators, stream(gen, t) returns the stream $[x$ for x in $gen]$ followed by $[t, t, t, \ldots]$ if gen is finite, while stream(gen, m, t) returns the first m elements of the stream $[x$ for x in $gen]$ followed by $[t, t, t, \ldots]$.

See also

`orbit`

Remarks

The function g must be defined for all $n \leq k$ (resp. $n \leq k < m$) and h must be defined for all $n \geq 0$, even if you intend to use `set!` to set specific values of the stream later. Note that stream(f) does not necessarily returns a constant stream since each call to $f()$ can side–effect its environment. Constant or eventually constant streams do not repeatedly store their constant value and are therefore preferable. For example, `stream(0)` stores only one value no matter how many values are requested, while `stream(() : MachineInteger +- > 0)` stores n times 0 when its n^{th} element is requested.

Example

The following code creates the stream of the Fibonnaci numbers and prints the first 10 of them:

```
import from MachineInteger, AldorInteger, Stream AldorInteger;
fib(n:MachineInteger, f:Stream AldorInteger):AldorInteger == {
        n = 0 or n = 1 => 1;
        f(n-1) + f(n-2);
}
sfib := stream fib;
a := sfib.9;    -- computes sfib.0 to sfib.9
stdout << a << newline;
```

Usage

import from String

Description

String provides basic strings, null–terminated, 0-indexed and without bound checking.

Exports

`ArrayType(Character, PackedPrimitiveArray Character)`

+:	(%, %) → %	concatenation
`char`:	% → `Character`	first character
`coerce`:	`Character` → %	conversion to a string
`coerce`:	% → `TextReader`	conversion to a text input stream
`coerce`:	% → `TextWriter`	conversion to a text output stream
`error`:	% → `Exit`	error exit
`new`:	`MachineInteger` → %	buffer allocation
`pointer`:	% → `Pointer`	conversion to a null–terminated C–string
`string`:	`Pointer` → %	conversion from a null–terminated C–string

Usage

s + t

Signature

+: (%, %) → %

Parameter	Type	Description
s,t	%	strings

Returns

s+*t* returns the string *st*. Copies all the characters of *s* and *t*, so *s* is unchanged after the call, and *s* and *t* do not share characters with $s + t$.

Remarks

$s +$ `empty` and `empty` $+ s$ both return a copy of *s*.

Example

If *s* is the string "abcde", then $s + (s + 2)$ is the string "abcdecde".

Usage

char s

Signature

char: % → `Character`

Parameter	Type	Description
s	%	a nonempty string

Description

Returns the first character of s.

Usage

c::%
s::TextReader
s::TextWriter

Signatures

coerce: `Character` → %
coerce: % → `TextReader`
coerce: % → `TextWriter`

Parameter	Type	Description
c	`Character`	a character
s	%	a string

Description

c::String converts the character c to the string "c", while s::T where T is an I/O stream type converts the string s to a text reader or writer, allowing one to read data or write data to it.

Remarks

When writing to a string, you must ensure that the string is large enough for all the data that will be written to it, since the string will not be extended and this function does not protect you against overwriting. When reading from or writing to a string, each coercion to a reader or writer resets the stream to the beginning of the string, and the string is not side–affected by the subsequent read or write operations, while the stream is side–affected. Thus, when reading several values from the same string, you must assign the reader to a variable and read the values from that variable, as in the example below.

Example

```
import from MachineInteger, String;
s := "  12  56";
a:MachineInteger := << s::TextReader;
b:MachineInteger := << s::TextReader;
```

assigns the value 12 to both a and b, while

```
import from MachineInteger, String;
s := "  12  56";
p := s::TextReader;
a:MachineInteger := << p;
b:MachineInteger := << p;
```

assigns 12 to a and 56 to b.

See also

PrimitiveMemoryBlock

coerce

Usage

error s

Signature

error: % → `Exit`

Parameter	Type	Description
s	%	a string

Description

Writes the message s to `stderr` and exits.

Usage

new n

Signature

new: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	a nonnegative size

Returns

Returns a string of $n + 1$ null characters. This is useful when creating a memory buffer of a specified number of bytes, or when creating a string to be used as an `TextWriter`.

Usage

pointer s
string p

Signatures

pointer: % → `Pointer`
string: `Pointer` → %

Parameter	Type	Description
s	%	A `libaldor` string
p	`Pointer`	A null–terminated C–string

Description

Use those functions to safely convert between null–terminated returned or needed by C–functions and `libaldor` strings. Those functions have no effect in the release version of `libaldor`, but they are necessary when using the debug version, so it is recommended to use them in all cases. The C–type `char*` should be replaced by `Pointer` in the prototypes when using C–functions in `libaldor` clients.

Usage

TableType(K, V): Category

Parameter	Type	Description
K	`PrimitiveType`	the type of the keys
V	Type	the type of the entries

Description

TableType(K, V) is the category of tables, *i.e.* discrete many-to-one mappings from keys to entries. More precisely, every element of a domain of this category is a table whose slots contain elements from V and such that every slot is given by a unique key from K.

Exports

`BoundedFiniteDataStructureType` `Cross`(K, V)

`apply`:	(%, K) → V	extraction of an entry
`entries`:	% → `Generator` V	iterate through the entries
`find`:	(K, %) → `Partial` V	search for an entry
`keys`:	% → `Generator` K	iterate through the keys
`numberOfEntries`:	% → `MachineInteger`	number of entries
`set!`:	(%, K, V) → V	modification of an entry
`table`:	() → %	creation of a table
	`MachineInteger` → %	

if K has `InputType` and V has `InputType` then `InputType`

if K has `OutputType` and V has `OutputType` then `OutputType`

if K has `SerializableType` and V has `SerializableType` then `SerializableType`

Usage

apply(t, k)
t.k

Signature

apply: (%, K) → V

Parameter	Type	Description
t	%	a table
k	K	a key

Returns

Returns the element of *t* with key *k*, which must be present in the table.

Remarks

Produces an error if *k* is not in *t*, use `find` if it is not known whether *k* is present in table.

Usage

for v in entries t repeat {...}
for k in keys t repeat {...}

Signatures

entries: % → `Generator` V
keys: % → `Generator` K

Parameter	Type	Description
t	%	a table

Description

These generators yield respectively all the entries, or keys in the table *t*.

Usage

find(k, t)

Signature

find: (K, %) → `Partial` V

Parameter	Type	Description
k	K	a key
t	%	a table

Returns

Returns *failed* if there is no element with key *k* in *t*, the element of *t* with key *k* otherwise.

See also

`apply`

Usage

numberOfEntries t

Parameter	Type	Description
t	%	a table

Returns

Returns the actual number of entries in the table *t*. That number can be different from the size of the table.

Usage

set!(t, k, v)
t.k := v;

Signature

set!: (%, K, V) → V

Parameter	Type	Description
t	%	a table
k	K	a key
v	V	an entry

Returns

Sets the element of t with key k to v and returns v.

Usage

table()
table n

Signature

table: `MachineInteger` → %

Parameter	Type	Description
n	`MachineInteger`	a starting size (optional)

Returns

Returns an empty table with initial space for *n* entries. That space grows when needed as elements are inserted in the table.

Usage

import from AldorLibraryInformation

Description

AldorLibraryInformation provides `libaldor` version information.

Exports

`VersionInformationType`

VersionInformationType

Usage

VersionInformationType: Category

Description

VersionInformationType is the category of types providing version information.

Exports

`name`:	`String`	library name
`credits`:	`List String`	various credits
`version`:	`String`	version information

Signature

credits: `List String`

Returns

Returns a list of lines, crediting the various authors of a library.

Usage
name

Signature
name: `String`

Returns
Returns the name of the library.

Usage
version

Signature
version: `String`

Returns
Returns a string describing the current version of a library.

Usage

import from BinarySearch(R, S)

Description

BinarySearch(R, S) provides a general version of binary search.

Parameter	Type	Description
R	`IntegerType`	The space being searched
S	`TotallyOrderedType`	The target values being searched for

Exports

`binarySearch`: (S, R → S, R, R) → (`Boolean`, R) binary search

Usage

binarySearch(s, f, a, b)

Signature

binarySearch: (S, R → S, R, R) → (`Boolean`, R)

Parameter	Type	Description
s	S	The value to search for
f	R → S	A monotonic increasing function
a	R	The left end of the interval to search
b	R	The right end of the interval to search

Returns

Returns (found?, r) such that $s = f(r)$ if found? is *true*. Otherwise, found? is *false* and:

- if $r < a$ then $s < f(a)$ or $b < a$;
- if $r \geq b$ then $s > f(b)$;
- if $a \leq r < b$, then $f(r) < s < f(r + 1)$;

Usage

import from CommandLine

Description

CommandLine provides utilities for command-line processing.

Exports

`arguments`:	`Array String`	command line arguments
`command`:	`String`	command line command
`flag`:	`Character → List String`	value of a command line flag
`flag?`:	`Character → Boolean`	test for a command line flag

Usage
arguments

Signature
arguments: `Array String`

Returns
Returns an array containing all the arguments of the command line, the command itself is not included.

Signature

command: `String`

Returns

Returns the command, *i.e.*the first word of the command line.

Usage

flag c
flag(c, s)
flag? c
flag?(c, s)

Signatures

flag: `Character` → `List String`
flag: (`Character`,`String`)→`List String`
flag?: `Character` → `Boolean`
flag?: (`Character`, `String`) → `Boolean`

Parameter	Type	Description
c	`Character`	flag code to look for
s	`String`	special flag codes (optional)

Returns

flag(c) returns all the values of the flag c each time it is present in the command line, an empty list otherwise.
flag?(c) returns *true* if the flag c is present in the command line, *false* otherwise.
In both functions, the optional argument s contains a list of flag codes which cause the rest of the argument to be skipped.

Example

If the command line to the program was `myprog -lsalli -l gmp -v`, then `flag?(char "a")` and `flag?(char "v")` both return *true*, while `flag?(char "b")` and `flag?(char "a", "l")` both return *false*. In addition, `flag(char "l")` returns the list `[‘‘salli’’, ‘‘gmp’’]`.

CopyableType

Usage

CopyableType: Category

Description

CopyableType is the category of types whose objects can be copied.

Exports

`copy`:	% → %	Make a copy
`copy!`:	(%, %) → %	In-place copy

Usage

copy y
copy!(x, y)

Signatures

copy: % → %
copy!: (%, %) → %

Parameter	Type	Description
x,y	%	Element of the type

Returns

copy(y) returns a copy of y, while copy!(x, y) returns a copy of y, where the storage used by x is allowed to be destroyed or reused, so x is lost after this call.

Remarks

Use copy before making in–place operations on a parameter. The call copy!(x, y) may cause x to be destroyed, so do not use it unless x has been locally allocated, and is guaranteed not to share space with other elements. Some functions are not necessarily copying their arguments and can thus create memory aliases.

Usage

import from Generator T

Parameter	Type	Description
T	Type	the type of the elements generated

Description

Generator T is a type which allows values of type T to be obtained serially in a 'repeat' or 'collect' form.

Exports

`next!`: $\% \to \mathrm{T}$ get the next element

Usage

next! g

Signature

next!: % → T

Parameter	Type	Description
g	%	a generator

Returns

Returns the next element produced by *g*, updating *g*. If *g* is empty, then next!(g) throws the exception `GeneratorException`.

GeneratorException

Usage

throw GeneratorException
try ... catch E in { E has `GeneratorExceptionType` =¿ ... }

Description

GeneratorException is an exception type thrown by stepping through an empty generator.

GeneratorExceptionType

Usage

GeneratorExceptionType: Category

Description

GeneratorExceptionType is the category of exceptions thrown by generators.

Usage

import from Partial T

Parameter	Type	Description
T	Type	a type

Description

Partial(T) implements a union of T and *failed*.

Exports

`[]`:	T → %	create an element
`failed`:	%	the element *failed*
`failed?`:	% → `Boolean`	check for the element *failed*
`retract`:	% → T	convert to an element of T

if T has `PrimitiveType` then
`PrimitiveType`

if T has `HashType` then
`HashType`

if T has `InputType` then
`InputType`

if T has `OutputType` then
`OutputType`

if T has `SerializableType` then
`SerializableType`

Usage

[t]

Signature

[]: T → %

Parameter	Type	Description
t	T	an element

Returns

Returns the element t converted to an element of %.

Usage

failed
failed? x

Signatures

failed: %
failed?: % → `Boolean`

Parameter	Type	Description
x	%	a partial element

Returns

failed returns the special element *failed*, while failed?(x) returns *true* if x is *failed*, *false* otherwise.

Usage

retract x

Signature

retract: $\% \to \mathrm{T}$

Parameter	Type	Description
x	$\%$	a partial element

Returns

Returns the element x converted to an element of T, provided that x is not *failed*.

Usage

import from Pointer

Description

Pointer implements machine pointers.

Exports

`HashType`
`InputType`
`OutputType`
`SerializableType`

`coerce`:	% → `MachineInteger`	conversion to an integer
`coerce`:	`MachineInteger` → %	conversion from an integer
`nil`:	%	the nil pointer
`nil?`:	% → `Boolean`	test for the nil pointer

Usage

nil
nil? p

Signatures

nil: → %
nil?: % → `Boolean`

Parameter	Type	Description
p	%	a pointer

Returns

nil returns the nil pointer, while nil?(p) returns *true* if p is the nil pointer, *false* otherwise.

Usage

import from Timer

Description

Timer is a type whose elements are stopwatch timers, which can be used to time precisely various sections of code, including garbage collection. The precision can be up to 1 millisecond but depends on the operating system. The times returned are CPU times (user + gc) used by the process that created the timer.

Exports

`gc`:	% → `MachineInteger`	read a timer
`read`:	% → `MachineInteger`	read a timer
`reset!`:	% → %	reset a timer to 0
`start!`:	% → `MachineInteger`	start or restart a timer
`stop!`:	% → `MachineInteger`	stop a timer
`timer`:	() → %	create a new timer

Signature

gc: % → `MachineInteger`

Parameter	Type	Description
t	%	The timer to read

Description

Reads the timer t without stopping it.

Returns

Returns the total accumulated garbage collection time in milliseconds by all the start/stop cycles since t was created or last reset. If t is running, the garbage collection time since the last start is added in, and t is not stopped or affected.

See also

`read`

Usage
read t

Signature
read: % → MachineInteger

Parameter	Type	Description
t	%	The timer to read

Description
Reads the timer t without stopping it.

Returns
Returns the total accumulated time in milliseconds by all the start/stop cycles since t was created or last reset. This times includes any eventual garbage collection time (see gc to extract this information). If t is running, the time since the last start is added in, and t is not stopped or affected.

Example
The following function takes a positive MachineInteger n as input, computes and prints a machine approximation of $\sum_{i=1}^{n} 1/i$, and returns the CPU time needed to compute it, but not the time needed to print it.

```
timeHarmonic(n:MachineInteger):MachineInteger == {
        import from MachineInteger, SingleFloat, Timer, Character, TextWriter;
        t := timer();
        m:SingleFloat := 1;
        start! t;
        for i in 2..n repeat m := m + 1 / (i::SingleFloat);
        stop! t;
        stdout << "H" << n << " = " << m << newline;
        read t;
}
```

See also
gc, start!, stop!

Usage

reset! t

Signature

reset!: % → %

Parameter	Type	Description
t	%	The timer to reset

Description

Resets the timer t to 0 and stops it if it is running.

Returns

Returns the timer t after it is reset.

Usage
start! t

Signature
start!: % → `MachineInteger`

Parameter	Type	Description
t	%	The timer to start

Description
Starts or restarts t, without resetting it to 0, It has no effect on t if it is already running.

Returns
Returns 0 if t was already running, the absolute time at which the start/restart was done otherwise.

See also
`read`, `stop!`

Usage

stop! t

Signature

stop!: % → `MachineInteger`

Parameter	Type	Description
t	%	The timer to stop

Description

Stops t without resetting it to 0. It has no effect on t if it is not running.

Returns

Returns the elapsed time in milliseconds since the last time t was restarted, 0 if t was not running.

See also

`read`, `start!`

Usage
timer()

Signature
timer: () → %

Description
Creates a timer, set to 0 and stopped.

Returns
Returns the timer that has been created.

See also
`reset!`

Index