
Algebra User Guide and Reference Manual

Manuel Bronstein and Marc Moreno Maza

Version 1.1.0 – April 19, 2005

Reconstruction from sources (extract) – April 16, 2022
<https://github.com/nilqed/aldor/tree/kfp/docs>

Algebra Contributors

Algebra is the result of extracting the central subset of the Σ^{it} library¹ 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

1	Introduction	16
2	User Guide	17
	Basic algebraic categories	18
	Arithmetic	18
	Data structures	20
	Input/Output	20
	Linear algebra	20
	Univariate polynomials and series	21
	Multivariate polynomials	23
	Compatibility with C types	25
	Using GMP	25
	Exceptions	25
	Profiling and debugging	26
3	Reference Manual	29
	Basic Categories	
	AbelianGroup	30
	AbelianMonoid	31
	Algebra	37
	CharacteristicZero	38
	CommutativeRing	39
	DecomposableRing	45
	DifferentialExtension	48
	DifferentialRing	50
	EuclideanDomain	53
	Field	62
	FiniteCharacteristic	63
	FiniteSet	65
	FreeAlgebra	71
	FreeLinearArithmeticType	72
	FreeLinearCombinationType	73
	FreeModule	75
	FreeRRing	83
	GcdDomain	85
	Group	88
	IndexedFreeAlgebra	91
	IndexedFreeLinearArithmeticType	92
	IndexedFreeLinearCombinationType	94
	IndexedFreeModule	98

IndexedFreeRRing	102
IntegerCategory	103
IntegralDomain	107
LinearArithmeticType	114
Module	117
Monoid	118
NonCommutativeIntegralDomain	124
Ring	127
RittRing	132
RRing	135
Specializable	136
ExpressionType	138

Commutative Algebra

Automorphism	141
CanonicalSimplification	145
ChineseRemaindering	146
Complex	149
Derivation	150
Fraction	154
FractionalRoot	155
FractionBy	162
FractionByCategory	163
FractionByCategory0	164
FractionCategory	167
FractionFieldCategory	170
FractionFieldCategory0	171
LinearCombinationFraction	173
ModularComputation	179
Product	180
ResidueClassRing	189
ReducibleModulusException	190
ReducibleModulusExceptionType	191
SimpleAlgebraicExtension	192
SimpleAlgebraicExtensionCategory	193
SourceOfPrimes	194
UnivariatePolynomialMod	195
UnivariatePolynomialModSqfr	196
UnivariatePolynomialQuotient	197
UnivariatePolynomialQuotientSqfr	205

Finite Fields

SmallPrimes	208
LazyHalfWordSizePrimes	209
HalfWordSizePrimes	210

WordSizePrimes	211
PrimeCollection	212
PrimeField2	221
PrimeFieldCategory	222
PrimeFieldCategory0	224
PrimitiveRoots	226
PthPowering	229
SmallPrimeField	231
SmallPrimeField0	232
SmallPrimeFieldCategory	233
SmallPrimeFieldCategory0	234
ZechPrimeField	237

Linear Algebra

Backsolve	238
DenseMatrix	241
DivisionFreeGaussElimination	242
FractionFreeGaussElimination	243
HermiteGaussElimination	244
LinearAlgebra	245
LinearAlgebraRing	260
LinearEliminationCategory	274
MatrixCategory	287
MatrixCategory2	309
MatrixCategoryOverFraction	311
ModulopGaussElimination	315
OrdinaryGaussElimination	328
OverdeterminedLinearSystemSolver	329
SpecializationLinearAlgebra	331
TwoStepFractionFreeGaussElimination	333
UnivariatePolynomialCRTLinearAlgebra	334
UnivariatePolynomialPopovLinearAlgebra	337
Vector	339
VectorOverFraction	344

Univariate Polynomials and Series

DenseUnivariatePolynomial	345
DenseUnivariateTaylorSeries	346
FactorizationRing	347
FFTRing	350
GenericModularPolynomialGcdPackage	354
HeuristicGcd	355
ModularUnivariateGcd	359
ModulopUnivariateGcd	361
PrimeFieldUnivariateFactorizer	363

RationalRootRing	368
Resultant	370
SparseUnivariatePolynomial0	376
SparseUnivariatePolynomial1	377
SparseUnivariatePolynomial	378
UnivariateFactorialPolynomial	379
UnivariateFreeLinearArithmeticType	382
UnivariateFreeRing	388
UnivariateFreeRing2	395
UnivariateFreeRingOverFraction	397
UnivariateGcdRing	398
UnivariateIntegralFactorizer	401
UnivariateMonomial	404
UnivariatePolynomialAlgebra	413
UnivariatePolynomialKaratsuba	436
UnivariatePolynomialRing	437
UnivariatePolynomialSquareFree	440
UnivariateTaylorSeriesType	443
UnivariateTaylorSeriesType2Poly	447
UnivariateTaylorSeriesNewtonSolver	454

Multivariate Polynomials

BivariateUtilitiesPackage	457
DirectProduct	458
DirectProductCategory	459
DistributedMultivariatePolynomial0	460
DistributedMultivariatePolynomial1	461
ExponentCategory	462
FiniteAbelianMonoidRing0	463
FiniteVariableType	464
GeneralExponentCategory	466
IntegerExponentVectorCategory	469
IntegerPolynomial	470
MachineIntegerDegreeLexicographicalExponent	471
MachineIntegerDegreeReverseLexicographicalExponent	472
MachineIntegerExponentVectorCategory	473
MachineIntegerLexicographicalExponent	474
OrderedSymbol	475
OrderedVariableList	476
OrderedVariableTuple	477
PolynomialRing	478
PolynomialRing0	479
PolynomialTypeRing	480
RecursiveMultivariatePolynomial0	481
RecursiveMultivariatePolynomialCategory0	482

ResultantOfBivariatePolynomialsOverSmallPrimeField	483
SparseIntegerMultivariatePolynomial	484
SparseMultivariatePolynomial	485
StandardFilteredRing	486
VariableType	487

Expression Trees

ExpressionTree	488
ExpressionTreeAnd	499
ExpressionTreeAssign	500
ExpressionTreeBigO	501
ExpressionTreeCase	502
ExpressionTreeComplex	503
ExpressionTreeEqual	504
ExpressionTreeExpt	505
ExpressionTreeFactorial	506
ExpressionTreeGreaterEqual	507
ExpressionTreeGreaterThan	508
ExpressionTreeIf	509
ExpressionTreeLeaf	510
ExpressionTreeLessEqual	525
ExpressionTreeLessThan	526
ExpressionTreeLispList	527
ExpressionTreeList	528
ExpressionTreeMatrix	529
ExpressionTreeMinus	530
ExpressionTreeNotEqual	531
ExpressionTreeOperator	532
ExpressionTreeOperatorTools	538
ExpressionTreePlus	542
ExpressionTreePrefix	543
ExpressionTreeQuotient	544
ExpressionTreeSubscript	545
ExpressionTreeTimes	546
ExpressionTreeVector	547

Shells

Evaluator	548
InfixExpressionParser	549
LispExpressionParser	551
MakePartialRing	552
Maple	553
Parsable	557
Parser	559
ParserReader	563

ParsingTools	565
PartialRing	568
Scanner	569
Shell	571
Token	574

Utilities

AlgebraLibraryInformation	584
IndexedVariable	585
Permutation	588
Sequence	593

4 libaldor Reference Manual 603

Basic Categories

AbelianGroup	604
AbelianMonoid	605
Algebra	611
CharacteristicZero	612
CommutativeRing	613
DecomposableRing	619
DifferentialExtension	622
DifferentialRing	624
EuclideanDomain	627
Field	636
FiniteCharacteristic	637
FiniteSet	639
FreeAlgebra	645
FreeLinearArithmeticType	646
FreeLinearCombinationType	647
FreeModule	649
FreeRRing	657
GcdDomain	659
Group	662
IndexedFreeAlgebra	665
IndexedFreeLinearArithmeticType	666
IndexedFreeLinearCombinationType	668
IndexedFreeModule	672
IndexedFreeRRing	676
IntegerCategory	677
IntegralDomain	681
LinearArithmeticType	688
Module	691
Monoid	692
NonCommutativeIntegralDomain	698

Ring	701
RittRing	706
RRing	709
Specializable	710
ExpressionType	712

Commutative Algebra

Automorphism	715
CanonicalSimplification	719
ChineseRemaindering	720
Complex	723
Derivation	724
Fraction	728
FractionalRoot	729
FractionBy	736
FractionByCategory	737
FractionByCategory0	738
FractionCategory	741
FractionFieldCategory	744
FractionFieldCategory0	745
LinearCombinationFraction	747
ModularComputation	753
Product	754
ResidueClassRing	763
ReducibleModulusException	764
ReducibleModulusExceptionType	765
SimpleAlgebraicExtension	766
SimpleAlgebraicExtensionCategory	767
SourceOfPrimes	768
UnivariatePolynomialMod	769
UnivariatePolynomialModSqfr	770
UnivariatePolynomialQuotient	771
UnivariatePolynomialQuotientSqfr	779

Finite Fields

SmallPrimes	782
LazyHalfWordSizePrimes	783
HalfWordSizePrimes	784
WordSizePrimes	785
PrimeCollection	786
PrimeField2	795
PrimeFieldCategory	796
PrimeFieldCategory0	798
PrimitiveRoots	800
PthPowering	803

SmallPrimeField	805
SmallPrimeField0	806
SmallPrimeFieldCategory	807
SmallPrimeFieldCategory0	808
ZechPrimeField	811

Linear Algebra

Backsolve	812
DenseMatrix	815
DivisionFreeGaussElimination	816
FractionFreeGaussElimination	817
HermiteGaussElimination	818
LinearAlgebra	819
LinearAlgebraRing	834
LinearEliminationCategory	848
MatrixCategory	861
MatrixCategory2	883
MatrixCategoryOverFraction	885
ModulopGaussElimination	889
OrdinaryGaussElimination	902
OverdeterminedLinearSystemSolver	903
SpecializationLinearAlgebra	905
TwoStepFractionFreeGaussElimination	907
UnivariatePolynomialCRTLinearAlgebra	908
UnivariatePolynomialPopovLinearAlgebra	911
Vector	913
VectorOverFraction	918

Univariate Polynomials and Series

DenseUnivariatePolynomial	919
DenseUnivariateTaylorSeries	920
FactorizationRing	921
FFTRing	924
GenericModularPolynomialGcdPackage	928
HeuristicGcd	929
ModularUnivariateGcd	933
ModulopUnivariateGcd	935
PrimeFieldUnivariateFactorizer	937
RationalRootRing	942
Resultant	944
SparseUnivariatePolynomial0	950
SparseUnivariatePolynomial1	951
SparseUnivariatePolynomial	952
UnivariateFactorialPolynomial	953
UnivariateFreeLinearArithmeticType	956

UnivariateFreeRing	962
UnivariateFreeRing2	969
UnivariateFreeRingOverFraction	971
UnivariateGcdRing	972
UnivariateIntegralFactorizer	975
UnivariateMonomial	978
UnivariatePolynomialAlgebra	987
UnivariatePolynomialKaratsuba	1010
UnivariatePolynomialRing	1011
UnivariatePolynomialSquareFree	1014
UnivariateTaylorSeriesType	1017
UnivariateTaylorSeriesType2Poly	1021
UnivariateTaylorSeriesNewtonSolver	1028

Multivariate Polynomials

BivariateUtilitiesPackage	1031
DirectProduct	1032
DirectProductCategory	1033
DistributedMultivariatePolynomial0	1034
DistributedMultivariatePolynomial1	1035
ExponentCategory	1036
FiniteAbelianMonoidRing0	1037
FiniteVariableType	1038
GeneralExponentCategory	1040
IntegerExponentVectorCategory	1043
IntegerPolynomial	1044
MachineIntegerDegreeLexicographicalExponent	1045
MachineIntegerDegreeReverseLexicographicalExponent	1046
MachineIntegerExponentVectorCategory	1047
MachineIntegerLexicographicalExponent	1048
OrderedSymbol	1049
OrderedVariableList	1050
OrderedVariableTuple	1051
PolynomialRing	1052
PolynomialRing0	1053
PolynomialTypeRing	1054
RecursiveMultivariatePolynomial0	1055
RecursiveMultivariatePolynomialCategory0	1056
ResultantOfBivariatePolynomialsOverSmallPrimeField	1057
SparseIntegerMultivariatePolynomial	1058
SparseMultivariatePolynomial	1059
StandardFilteredRing	1060
VariableType	1061

Expression Trees

ExpressionTree	1062
ExpressionTreeAnd	1073
ExpressionTreeAssign	1074
ExpressionTreeBigO	1075
ExpressionTreeCase	1076
ExpressionTreeComplex	1077
ExpressionTreeEqual	1078
ExpressionTreeExpt	1079
ExpressionTreeFactorial	1080
ExpressionTreeGreaterEqual	1081
ExpressionTreeGreaterThan	1082
ExpressionTreeIf	1083
ExpressionTreeLeaf	1084
ExpressionTreeLessEqual	1099
ExpressionTreeLessThan	1100
ExpressionTreeLispList	1101
ExpressionTreeList	1102
ExpressionTreeMatrix	1103
ExpressionTreeMinus	1104
ExpressionTreeNotEqual	1105
ExpressionTreeOperator	1106
ExpressionTreeOperatorTools	1112
ExpressionTreePlus	1116
ExpressionTreePrefix	1117
ExpressionTreeQuotient	1118
ExpressionTreeSubscript	1119
ExpressionTreeTimes	1120
ExpressionTreeVector	1121

Shells

Evaluator	1122
InfixExpressionParser	1123
LispExpressionParser	1125
MakePartialRing	1126
Maple	1127
Parsable	1131
Parser	1133
ParserReader	1137
ParsingTools	1139
PartialRing	1142
Scanner	1143
Shell	1145
Token	1148

Utilities

AlgebraLibraryInformation 1158

IndexedVariable 1159

Permutation 1162

Sequence 1167

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 “`tar -xzvf algebra.tar.gz`” 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 `sh makealgebra; shell`;
- if you want to build the GMP version of the library, execute `sh makealgebra-gmp`. 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 `sh makealgebrad`. 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 environment variable `ALGEBRAROOT` should be set to the main `algebra` directory;
- `$ALGEBRAROOT/include` should be appended to your `INCPATH` environment variable;
- `$ALGEBRAROOT/lib` should be appended to your `LIBPATH` environment 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
```

to your compiler command line, or

```
-lalgebrad -laldord -dDEBUG -y$ALGEBRAROOT/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 `algebra/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 need to be mentioned.

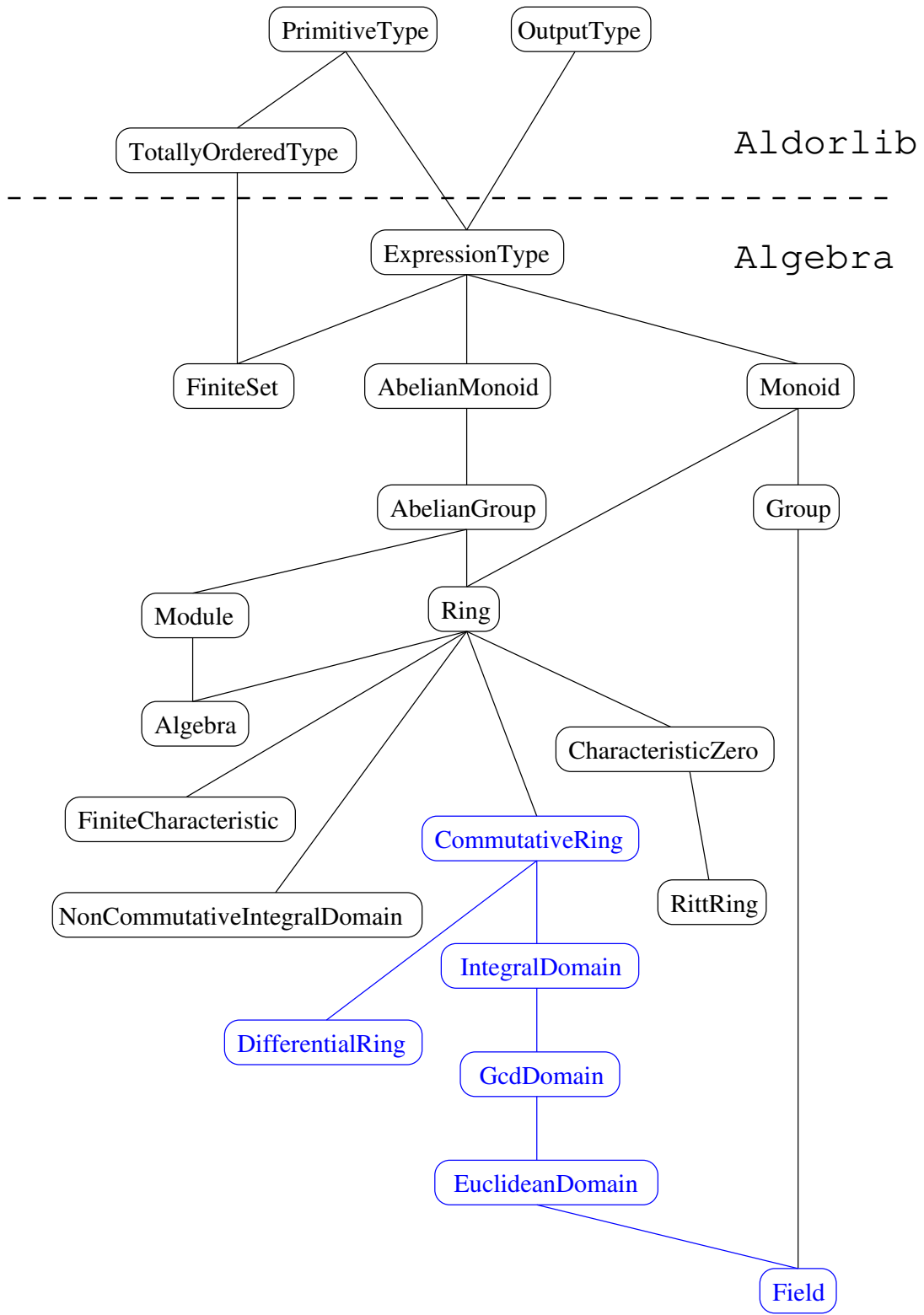
- The multiplication `*` is in general not assumed to be commutative, except in the subtree starting at **CommutativeRing**, which appears in blue in Figure 1. This means that code written for the other categories should be careful and not assume commutativity of `*`. Addition is on the other hand always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

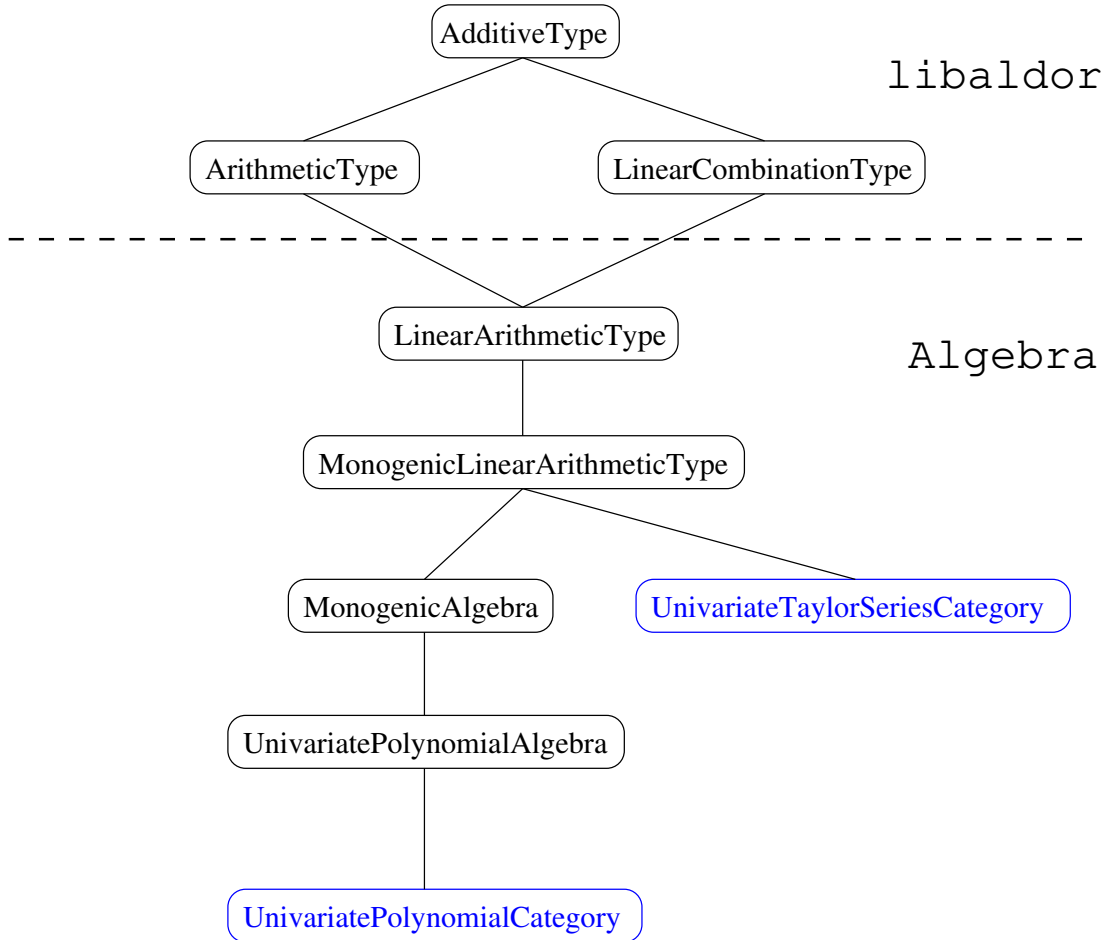


Figure 2: The `Algebra` univariate polynomial category hierarchy

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:UnivariateTaylorSeriesType R): with {
  hypergeometricSeries: (R, R) -> Rx;
} == add {
  hypergeometricSeries(a0:R, c:R):Rx == {
    import from Sequence R;
    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.

To construct a multivariate polynomial type with `SparseMultivariatePolynomial` one needs a `Ring` as a coefficient type and a `VariableType` as a variable type.

```
#include "algebra"
import from String, Symbol;
macro V == OrderedVariableTuple("-x","-y","-z");
import from MachineInteger, V;
macro P == SparseMultivariatePolynomial(Integer,V);
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;
```

To construct a multivariate polynomial type with `DistributedMultivariatePolynomial1` one needs a `Ring` as a coefficient type, a `VariableType` as a variable type, and `ExponentCategory(V)` as an exponent type. So we can continue the above sample program as follows

```
macro E1 == MachineIntegerDegreeLexicographicalExponent(V);
macro E2 == MachineIntegerLexicographicalExponent(V);
macro Q1 == DistributedMultivariatePolynomial1(Integer, V, E1);
macro Q2 == DistributedMultivariatePolynomial1(Integer, V, E2);
q1 := expand(E1)(Q1)(p);
q2 := expand(E2)(Q2)(p);
```

The above polynomials appear respectively as follows in an interpreter session.

```
%20 >> q1 := expand(E1)(Q1)(p)
x*y^3 + z^4 + x^2*y + x^2*z + x*z @ Q1
```

```
%21 >> q2 := expand(E2)(Q2)(p);
x^2*y + x^2*z + x*y^3 + x*z + z^4 @ Q2
```

Do not forget to enter

```
#include "aldorinterp"
```

after

```
#include "algebra"
```

in every interpreter session.

There are other multivariate polynomial types like `RecursiveMultivariatePolynomial0`. It is similar to `SparseMultivariatePolynomial` but takes a third argument which is a univariate polynomial type constructor:

```
UP: (R: Join(ArithmeticType, ExpressionType), avar: Symbol == new()
    ) -> UnivariatePolynomialCategory(R),
```

This third argument is used for the internal representation.

Another useful type is `IntegerPolynomial` which provides multivariate polynomial with integer coefficient and variables from `OrderedSymbol`. This multivariate polynomial type is easy to use as shown by the following interpreter session.

```
%1 >> #include "algebra"
                                     Comp: 130 msec, Interp: 0 msec
%2 >> #include "aldorinterp"
                                     Comp: 70 msec, Interp: 0 msec
%3 >> macro P == IntegerPolynomial;
                                     Comp: 0 msec, Interp: 0 msec
%4 >> import from String, P, Integer;
                                     Comp: 210 msec, Interp: 0 msec
%5 >> x: P := "x" :: P
```



```

x @ IntegerPolynomial
                                     Comp: 0 msec, Interp: 520 msec
%6 >> y: P := "y" :: P
y @ IntegerPolynomial
                                     Comp: 0 msec, Interp: 0 msec
%7 >> z: P := "z" :: P
z @ IntegerPolynomial
                                     Comp: 0 msec, Interp: 0 msec
%8 >> p := (y + z)*x^2 + (y^3 + z)*x + z^4 + 3::P;
z^4+(x^2+x)*z+x*y^3+x^2*y+3 @ IntegerPolynomial

```

Observe the `3::P`. Indeed, it is necessary to convert the integer 3 into a polynomial. Observe also the ordering on symbols is reversed w.r.t. the dictionary one.

Compatibility with C types

All the types of `libaldor` 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
```

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

`-laldebrad -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

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** $+: (\%,\%) \rightarrow \%$

Parameter	Type	Description
x, y	$\%$	elements of the monoid

ReturnsReturns the sum $x + y$.

Usage $n * x$ **Signature** $*: (\text{Integer}, \%) \rightarrow \%$

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

ReturnsReturns the product nx .

Usage

add!(x, y)

Signature

add!: ($\%$, $\%$) \rightarrow $\%$

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?: 'a → Boolean

Parameter	Type	Description
x	'a	an element of the monoid

Returns

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

Algebra

Usage

Algebra R:Category

Parameter	Type	Description
R	Ring	The coefficient ring

Description

Algebra R is the category of algebras over R , *i.e.* R -rings such that $R \cdot 1$ is included in the center (in other word, multiplication by R is bilinear).

Exports

RRing R

CharacteristicZero

Usage

CharacteristicZero: Category

Description

CharacteristicZero is the category of rings of characteristic 0.

Exports

Ring

CommutativeRing

Usage

CommutativeRing: Category

Description

CommutativeRing is the category of commutative rings.

Exports

Ring

canonicalUnitNormal?:	$\rightarrow \text{Boolean}$	Check if <code>unitNormal</code> is canonical
cutoff:	$\mathbb{Z} \rightarrow \mathbb{Z}$	Cutoffs for various fast algorithms
reciprocal:	$\% \rightarrow \text{Partial } \%$	Inverse
unitNormal:	$\% \rightarrow (\%, \%, \%)$ $(\%, \%) \rightarrow (\%, \%)$	Representative of the associates
unit?:	$\% \rightarrow \text{Boolean}$	Test whether an element is a unit

where

$\mathbb{Z} == \text{MachineInteger}$

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 (y, u, u^{-1}) and `unitNormal(x')` returns (y', u', u'^{-1}) , then $y = y'$.

See also

`unitNormal`, `unit?`

Usage

cutoff t

Signature

cutoff: `MachineInteger` \rightarrow `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: % \rightarrow 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?: % \rightarrow Boolean

Parameter	Type	Description
x	%	An element of the ring

Returns

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

See also

canonicalUnitNormal?, unitNormal

DecomposableRing

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?: % \rightarrow Boolean

someFactors: % \rightarrow List % Get some factors

Usage

provablyIrreducible? x

Signature

provablyIrreducible?: % \rightarrow 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: $\% \rightarrow \text{List } \%$

Parameter	Type	Description
x	$\%$	A ring element

Returns

Returns $[x_1, \dots, x_n]$ 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.

DifferentialExtension

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 \rightarrow Derivation % Extension of a derivation

if R has DifferentialRing then
DifferentialRing

Usage

lift *D*

Signature

lift: Derivation *R* \rightarrow Derivation %

Parameter	Type	Description
<i>D</i>	Derivation <i>R</i>	A derivation on <i>R</i>

Returns

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

DifferentialRing

Usage

DifferentialRing: Category

Description

DifferentialRing is the category of commutative differential rings.

Exports

CommutativeRing

derivation: \rightarrow Derivation % The derivation

differentiate: $(\%, \text{Integer}) \rightarrow \%$ 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: $(\%, \text{Integer}) \rightarrow \%$

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)

EuclideanDomain

Usage

EuclideanDomain: Category

Description

EuclideanDomain is the category of commutative Euclidean domains.

Exports

GcdDomain		
diophantine:	$(\%, \%, \%) \rightarrow \partial \%$	Linear diophantine solver
divide:	$(\%, \%) \rightarrow (\%, \%)$	Euclidean division
divide!:	$(\%, \%, \%) \rightarrow (\%, \%)$	In-place Euclidean division
euclid:	$(\%, \%) \rightarrow \%$	Euclidean gcd
euclid!:	$(\%, \%) \rightarrow \%$	In-place Euclidean gcd
euclideanSize:	$\% \rightarrow \text{Integer}$	Size function of the domain
extendedEuclidean:	$(\%, \%) \rightarrow (\%, \%, \%)$	Extended Euclidean Algorithm
	$(\%, \%, \%) \rightarrow \partial \text{Cross}(\%, \%)$	
quo:	$(\%, \%) \rightarrow \%$	Quotient
rem:	$(\%, \%) \rightarrow \%$	Remainder
remainder!:	$(\%, \%) \rightarrow \%$	In-place remainder
rationalReconstruction:	$(\%, \%, \text{Z}, \text{Z}) \rightarrow \partial \text{Cross}(\%, \%)$	Rational reconstruction

where

∂ == Partial
Z == Integer

Usage

diophantine(a, b, m)

Signature

diophantine: (% , % , %) \rightarrow **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

Signaturesdivide: ($\%$, $\%$) \rightarrow ($\%$, $\%$)quo,rem: ($\%$, $\%$) \rightarrow $\%$

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

Returns

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

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: (% , %) \rightarrow %

Parameter	Type	Description
x, y	%	Elements of the ring

Returns

Returns $\text{gcd}(x, y)$ 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: $\% \rightarrow \text{Integer}$

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: $(\%, \%) \rightarrow (\%, \%, \%)$
 extendedEuclidean: $(\%, \%, \%) \rightarrow \mathbf{Partial}(\%, \%)$

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

rationalReconstruction(u, m, n, d)

Signature

rationalReconstruction: (`%`, `%`, `Integer`, `Integer`) \rightarrow `Partial Cross`(`%`, `%`)

Parameter	Type	Description
u	<code>%</code>	An element of the ring
m	<code>%</code>	A nonzero modulus
n, d	<code>Integer</code>	Bounds for the size of the result

Returns

Returns either (a, b) such that $a/b = u \pmod{m}$, $|a| \leq n$ and $|b| \leq d$, or *failed* if no such a, b exist.

Remarks

The resulting a and b might not be unique, depending on the values of the bounds n and d .

Usage

remainder!(x, y)

Signature

remainder!: (% , %) → %

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

FiniteCharacteristic

Usage

FiniteCharacteristic: Category

Description

FiniteCharacteristic is the category of finite characteristic rings.

Exports

Ring

pthPower: % \rightarrow % Exponentiation to the characteristic

pthPower!: % \rightarrow % In-place exponentiation to the characteristic

Usage

pthPower x
 pthPower! x

Signature

pthPower: % \rightarrow %

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`.

FiniteSet

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
<i>p</i>	<i>%</i>	An element
<i>x</i>	ExpressionTree	A name for the variables

Returns

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

Usage

index *p*

Signature

index: $\% \rightarrow \text{Integer}$

Parameter	Type	Description
<i>p</i>	$\%$	An element

Returns

Returns the index of *p*.

See also

lookup(FiniteSet)

Usage

lookup *j*

Signature

lookup: Integer \rightarrow %

Parameter	Type	Description
<i>j</i>	Integer	An index

Returns

Returns the element with index *j*.

See also

index(FiniteSet)

Usage

random()

Signature

random: $() \rightarrow \%$

Returns

Returns a random element.

FreeAlgebra

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

FreeRRing R

If R has Ring then
Algebra R

FreeLinearArithmeticType

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 Algebra R even when R is a Ring.

Exports

FreeLinearCombinationType R
LinearArithmeticType R

FreeLinearCombinationType

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 **Module** R even when R is a **Ring**.

Exports

ExpressionType

LinearCombinationType R

map: $(R \rightarrow R) \rightarrow \% \rightarrow \%$ Lift a mapping

map!: $(R \rightarrow R) \rightarrow \% \rightarrow \%$ Lift a mapping

Usage

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

Signature

```
map: (R → R) → % → %
```

Parameter	Type	Description
f	$R \rightarrow R$	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 $m \rightarrow f(m)$. In both cases, map! does not make a copy of m but modifies it in place.

FreeModule

Usage

FreeModule R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

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		
leadingCoefficient:	$\% \rightarrow R$	The leading coefficient
nonZeroCoefficients:	$\% \rightarrow \text{Generator } R$	Iterate over the coefficients
reductum:	$\% \rightarrow \%$	All terms except the leading one
reductum!:	$\% \rightarrow \%$	All terms except the leading one
support:	$\% \rightarrow \text{Generator Cross}(R, \%)$	Make an iterator
term?:	$\% \rightarrow \text{Boolean}$	Test for a monomial
trailingCoefficient:	$\% \rightarrow R$	The trailing coefficient

If R has Ring then
Module R

If R has GcdDomain then		
content:	$\% \rightarrow R$	Content
primitive:	$\% \rightarrow (R, \%)$	Content and primitive part
primitive!:	$\% \rightarrow (R, \%)$	Content and primitive part
primitivePart:	$\% \rightarrow \%$	Primitive part
primitivePart!:	$\% \rightarrow \%$	Primitive part

If R has Field then		
monic:	$\% \rightarrow \%$	Make monic
monic!:	$\% \rightarrow \%$	Make monic

Usage

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

Signatures

content,content!: $\% \rightarrow R$
 primitive,primitive!: $\% \rightarrow (R, \%)$
 primitivePart,primitivePart!: $\% \rightarrow \%$

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

term? m

Signature

term?: % \rightarrow Boolean

Parameter	Type	Description
m	%	An element of the module

Returns

Returns *true* if $m = re$ for $r \in R$ and e an element of the basis, *false* otherwise.

Remarks

term?(0) returns *true*.

Usage

leadingCoefficient m
trailingCoefficient p

Signature

leadingCoefficient, trailingCoefficient: $\% \rightarrow R$

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

Usagemonic *m*monic! *m***Signature**monic: % \rightarrow %

Parameter	Type	Description
<i>m</i>	%	An element of the module

ReturnsReturns $r^{-1}m$ 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: % \rightarrow 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: $\% \rightarrow \%$

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: % \rightarrow 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 .

FreeRRing

Usage

FreeRRing R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

FreeRRing R is a category for free R-rings 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

ground?: % \rightarrow Boolean Test for a ground element

If R has CharacteristicZero then

CharacteristicZero

If R has FiniteCharacteristic then

FiniteCharacteristic

If R has Ring then

RRing R

if R has RittRing then

RittRing

Usage

ground? m

Signature

ground?: % \rightarrow Boolean

Parameter	Type	Description
m	%	An element of the module

Returns

Returns *true* if $m = r \cdot 1$ for $r \in R$, *false* otherwise.

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 [ $x_1, \dots, x_n$ ]

```

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

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

Signatures

gcdquo: $(\%, \%) \rightarrow (\%, \%, \%)$
gcdquo: **List** $\% \rightarrow (\%, \text{List } \%)$

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($[x_1, \dots, x_n]$) returns $(g, [y_1, \dots, y_n])$ where $g = \gcd(x_1, \dots, x_n)$ 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

`/:` $(\%, \%) \rightarrow \%$ quotient

`inv:` $\% \rightarrow \%$ inverse

Usage

x/y

Signature

$/: \quad \% \rightarrow \%$

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

Returns

Returns xy^{-1} .

Usage

inv x

Signature

inv: % \rightarrow %

Parameter	Type	Description
x	%	An element of the group

Returns

Returns x^{-1} .

IndexedFreeAlgebra

Usage

IndexedFreeAlgebra(R,E): 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

IndexedFreeRRing(R , E)

IndexedFreeLinearArithmeticType

Usage

`IndexedFreeLinearArithmeticType(R, E): Category`

Parameter	Type	Description
R	<code>ExpressionType</code> <code>ArithmeticType</code>	The coefficient domain
E	<code>ExpressionType</code>	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, E)`

`FreeLinearArithmeticType R`

`add!:` `(%, R, E, %) → %` Add a shifted element

Usage

add!(p, c, e, q)

Signature

add!: ($\%$, R, E, $\%$) \rightarrow $\%$

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.

IndexedFreeLinearCombinationType

Usage

IndexedFreeLinearCombinationType(R , E): Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain
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: (% , E) → 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) \rightarrow $\%$

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
<i>p</i>	<i>%</i>	An element of the module
<i>e</i>	<i>E</i>	An exponent

Returns

Returns the coefficient of *e* in *p*.

Usage

monomial(e)
term(r , e)

Signatures

monomial: $E \rightarrow \%$
term: $(R, E) \rightarrow \%$

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,E): Category

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

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:	% $\rightarrow E$	degree
generator:	% \rightarrow Generator Cross(R , E)	iterate through all the terms
leadingMonomial:	% \rightarrow %	leading monomial
leadingTerm:	% $\rightarrow (R, E)$	leading term
terms:	% \rightarrow Generator Cross(R , E)	iterate through all the terms
trailingDegree:	% $\rightarrow E$	trailing degree
trailingMonomial:	% \rightarrow %	trailing monomial
trailingTerm:	% $\rightarrow (R, E)$	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: % \rightarrow 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 $a_n \neq 0 \neq a_m$. When $p = 0$, both functions return an arbitrary value in E .

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`

IndexedFreeRRing

Usage

IndexedFreeRRing(R,E): Category

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

Description

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

Exports

FreeRRing R

IndexedFreeModule(R , E)

IndexedFreeLinearArithmeticType(R , E)

IntegerCategory

Usage

IntegerCategory: Category

Description

IntegerCategory is the category of integer-like rings.

Exports

CharacteristicZero

EuclideanDomain

IntegerType

Parsable

Specializable

integer: $\% \rightarrow \text{Integer}$ Conversion to an integer

rationalReconstruction: $\% \rightarrow \% \rightarrow \text{Partial Cross}(\%, \%)$ Rational reconstruction

Usage

binomial(n, m)

Signature

binomial: (%,%) \rightarrow %

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: % \rightarrow Integer

Parameter	Type	Description
<i>n</i>	%	An integer

Returns

Returns n as an integer.

Usage

rationalReconstruction m
 rationalReconstruction(m)(u)

Signature

rationalReconstruction: $\% \rightarrow \% \rightarrow \text{Partial Cross}(\%, \%)$

Parameter	Type	Description
m	$\%$	A nonzero modulus
u	$\%$	An Integer

Returns

rationalReconstruction(m)(u) returns either (a, b) such that $a/b = u \pmod{m}$, $|a| \leq \sqrt{(m-1)/2}$ and $|b| \leq \sqrt{(m-1)/2}$, or *failed* if no such a, b exist.

Remarks

The resulting a and b are guaranteed to be unique.

See also

rationalReconstruction

IntegralDomain

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: (% , % , % , % , %) \rightarrow %

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: (% , % , % , % , % , % , %) \rightarrow %

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	<code>%</code>	The numerator
y	<code>%</code>	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:  % → % → 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 $a^{n+1} \nmid b$, while `order(a)` returns the map $b \rightarrow \nu_a(b)$.

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 $l := [p_1, \dots, p_n]$:

```
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 $a \nmid c$, while orderquo(a) returns the map $b \rightarrow \text{orderquo(a)}(b)$.

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 $x = qy$ if such a q exists, while quotientBy(y) returns the map $x \rightarrow \text{quotient}(x, y)$ and quotientBy!(y) returns the map $x \rightarrow \text{quotient!}(x, y)$. 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
```

LinearArithmeticType

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
 \wedge : ($\%$, Integer) \rightarrow % exponentiation
coerce: $R \rightarrow \%$ Natural embedding

Usage $x \wedge n$ **Signatures** $\wedge: (\%, \text{Integer}) \rightarrow \%$

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

ReturnsReturns x to the power n .

Usage

coerce r

Signature

coerce: $R \rightarrow \%$

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
^:	(%, Integer) → %	exponentiation
one?:	% → Boolean	test for 1
times!:	(%, %) → %	In-place product

Usage

1

Signature

1: %

Returns

Returns the constant 1.

Usage $x * y$ **Signature** $*: (\%,\%) \rightarrow \%$

Parameter	Type	Description
x, y	$\%$	elements of the monoid

ReturnsReturns the product xy .

Usage $x \wedge n$ **Signatures** $\wedge: (\%, \text{Integer}) \rightarrow \%$

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

ReturnsReturns x^n .

Usage

one? x

Signature

one?: % → Boolean

Parameter	Type	Description
<i>x</i>	%	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.

NonCommutativeIntegralDomain

Usage

NonCommutativeIntegralDomain: Category

Description

NonCommutativeIntegralDomain is the category of non-commutative integral domains.

Exports

Ring

leftExactQuotient: $(\%, \%) \rightarrow \text{Partial } \%$ Left exact quotient

rightExactQuotient: $(\%, \%) \rightarrow \text{Partial } \%$ Right exact quotient

Usage

leftExactQuotient(x, y)

Signature

leftExactQuotient: (`%`, `%`) \rightarrow `Partial %`

Parameter	Type	Description
x	<code>%</code>	The numerator
y	<code>%</code>	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: (`%`, `%`) \rightarrow `Partial %`

Parameter	Type	Description
x	<code>%</code>	The numerator
y	<code>%</code>	The denominator

Returns

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

See also

leftExactQuotient(NonCommutativeIntegralDomain)

Ring

Usage

Ring: Category

Description

Ring is the category of rings, not necessarily commutative.

Exports

AbelianGroup

ArithmeticType

Monoid

characteristic: Integer

characteristic

coerce: Integer \rightarrow %

embedding of the integers

factorial: (% , % , MachineInteger) \rightarrow %

Generalized factorial

random: () \rightarrow %

Get a random element

Usage

characteristic

Signature

characteristic: `Integer`

Returns

Returns the characteristic of the ring.

Usage`coerce n``n::%`**Signature**`coerce: Integer \rightarrow %`

Parameter	Type	Description
<i>n</i>	Integer	an integer

Returns

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

Usage

factorial(a, s, n)

Signature

factorial: (% , % , MachineInteger) \rightarrow %

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: $() \rightarrow \%$

Returns

Returns a random element.

RittRing

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

$/:$ $(\%, \text{Integer}) \rightarrow \%$ Division by a nonzero integer

inv: $\text{Integer} \rightarrow \%$ Inversion of a nonzero integer

Usage x / n **Signature** $/: (\%, \text{Integer}) \rightarrow \%$

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 \rightarrow %

Parameter	Type	Description
n	Integer	A nonzero integer

Returns

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

RRing

Usage

RRing R:Category

Parameter	Type	Description
R	Ring	The coefficient ring

Description

RRing R is the category of R-rings, *i.e.* rings that are also R-modules.

Exports

LinearArithmeticType R

Module R

Ring

Specializable

Usage

Specializable: Category

Description

Specializable is the category of specializable types.

Exports

specialization: (Image:CommutativeRing) \rightarrow PartialFunction(% , Image) Morphism

Usage

specialization R

Signature

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

Parameter	Type	Description
<i>R</i>	CommutativeRing	A ring

Returns

Returns a partial map from % into R.

ExpressionType

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: % \rightarrow ExpressionTree

Parameter	Type	Description
x	%	The element to convert

Description

Converts x to an expression tree.

Usage

relativeSize x

SignaturerelativeSize: % \rightarrow 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 automorphisms on R .

Exports

Group

apply: $(\%, R, \text{Integer}) \rightarrow R$ Apply a morphism to an element of R

function: $\% \rightarrow (R \rightarrow R)$ Action of a morphism

morphism: $(R \rightarrow R) \rightarrow \%$ Create a morphism

$(R \rightarrow R, R \rightarrow R) \rightarrow \%$

$((R, \text{Integer}) \rightarrow R) \rightarrow \%$

Usage

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

Signature

`apply: (% , R, n:Integer == 1) → R`

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

Returns

Returns $\sigma^n x$.

Usage

function σ

Signature

function: $\% \rightarrow (R \rightarrow R)$

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: $(R \rightarrow R) \rightarrow \%$
 morphism: $(R \rightarrow R, R \rightarrow R) \rightarrow \%$
 morphism: $((R, \text{Integer}) \rightarrow R) \rightarrow \%$

Parameter	Type	Description
f	$R \rightarrow R$	A function
f^{-1}	$R \rightarrow R$	The inverse function of f
g	$(R, \text{Integer}) \rightarrow R$	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

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

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 $g: (R, \text{Integer}) \rightarrow R$ should be used to avoid repeated iterations of σ , which is the default behavior.

Usage

CanonicalSimplification: Category

Description

CanonicalSimplification is the category of domains supporting a canonical simplifier. This means that for any p such that p equals its canonical associate and for any a , the element $a \bmod p$ is a canonical representative of the residue class of a by p . That is, for any b the relation $a \bmod p = b \bmod p$ is equivalent to p divides $b - a$. In addition $a \bmod p$ equals its canonical associate. If the domain is Euclidean, then it must support also a symmetric canonical simplifier. See the paper *On the genericity of the Modular Gcd Algorithm* by Kaltofen and Monagan in the proc. of ISSAC 1999.

Exports

CommutativeRing

mod:	(%, %) → %	residue class representative
mod_+:	(%, %) → %	modular sum
mod_-:	(%, %) → %	modular difference
mod_*:	(%, %) → %	modular product
recipMod	(%, %) → Partial %	modular reciprocal

if % has EuclideanDomain then

symmetricMod: (%, %) → % symmetric residue class representative

ChineseRemaindering

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: $(R,R) \rightarrow (R,R) \rightarrow R$ combine interpolated result with new modulus.
interpolate: $(\text{List } R, \text{List } R) \rightarrow R$ interpolate given residues and moduli.

if R has IntegerType then

combine: $(R, \text{MachineInteger}) \rightarrow (R, \text{MachineInteger}) \rightarrow R$ 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	A new modulus.
	MachineInteger	
A	R	The interpolated value modulo M .
a	R	The residue modulo m .
	MachineInteger	

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) \rightarrow 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.

Complex

Usage

import from Complex R

Parameter	Type	Description
R	ArithmeticType ExpressionType	Type to be extended

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

Derivation

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: $(\%, R, \text{Integer}) \rightarrow R$ Differentiate an element of R

derivation: $(R \rightarrow R) \rightarrow \%$ Create a derivation

function: $\% \rightarrow (R \rightarrow R)$ 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: $(R \rightarrow R) \rightarrow \%$

Parameter	Type	Description
f	$R \rightarrow R$	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: % \rightarrow ($R \rightarrow R$)

Parameter	Type	Description
D	%	A derivation

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

Fraction

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

FractionalRoot

Usage

```
import from FractionalRoot
```

Description

FractionalRoot(R) provides fractions of R with multiplicities.

Parameter	Type	Description
R	CommutativeRing	A ring

Exports

ExpressionType

fractionalRoot:	$(R, R, \text{Integer}) \rightarrow \%$	Create a root
integral?:	$\% \rightarrow \text{Boolean}$	Test whether root is integral
integralRoot:	$(R, \text{Integer}) \rightarrow \%$	Create a root
integralValue:	$\% \rightarrow R$	Value of an integral root
multiplicity:	$\% \rightarrow \text{Integer}$	Multiplicity of a root
setMultiplicity!:	$(\%, \text{Integer}) \rightarrow \%$	Change a multiplicity
value:	$\% \rightarrow (R, R)$	Value of a root

Signature

integral?: % \rightarrow Boolean

Usage

integral? r

Parameter	Type	Description
<i>r</i>	%	A root

Returns

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

UsagefractionalRoot(*a*, *b*, *n*)integralRoot(*a*, *n*)**Signatures**

fractionalRoot: (R, R, Integer) → %

integralRoot: (R, Integer) → %

Parameter	Type	Description
<i>a</i>	R	A numerator
<i>b</i>	R	A denominator
<i>n</i>	Integer	A multiplicity

ReturnsReturn the root *a* or *a/b* with multiplicity *n*.

Usage

integralValue r

Signature

integralValue: % \rightarrow 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: $\% \rightarrow \text{Integer}$

Parameter	Type	Description
<i>r</i>	$\%$	A root

Returns

Return the multiplicity of *r*.

Usage

setMultiplicity!(*r*, *m*)

Signature

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

Parameter	Type	Description
<i>r</i>	<i>%</i>	A root
<i>m</i>	Integer	Its new multiplicity

Description

Sets the multiplicity of *r* to *m* and returns *r*.

Usage

(n, d) := value r

Signature

value: % \rightarrow (Integer, Integer)

Parameter	Type	Description
<i>r</i>	%	A root

Returns

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

See also

integralValue

FractionBy

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

FractionByCategory

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

FractionByCategory0

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: % \rightarrow Integer Valuation at p
shift: (% , Integer) \rightarrow % Multiplication by a power of p

Usage

order x

Signature

order: % \rightarrow Integer

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

Returns

Returns n such that $x = ap^n$ and $a \in R, p \nmid a$.

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 $x p^n$.

FractionCategory

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

LinearAlgebraRing

denominator: $\% \rightarrow R$ Denominator of a fraction

normalize: $\% \rightarrow \%$ Normalize a fraction

numerator: $\% \rightarrow R$ 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: % \rightarrow R

Parameter	Type	Description
x	%	A fraction

Returns

Returns respectively the denominator and the numerator of a fraction.

Usage

normalize x

Signature

normalize: $\% \rightarrow \%$

Parameter	Type	Description
x	$\%$	A fraction

Description

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

FractionFieldCategory

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

FractionFieldCategory0

Usage

FractionFieldCategory0 R: Category

Parameter	Type	Description
<i>R</i>	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

$/: (R,R) \rightarrow \%$

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

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

if Q has FractionByCategory0 R

makeIntegralBy: $LQ \rightarrow (Integer, LR)$ Conversion to an integral element

if Q has FractionFieldCategory0 R

normalize: $LR \rightarrow (R, LQ)$ 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

$(a, A) := \text{makeIntegral } B$

Signature

$\text{makeIntegral}: \text{LQ} \rightarrow (\text{R}, \text{LR})$

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

$(\mu, A) := \text{makeIntegralBy } B$

Signature

$\text{makeIntegralBy}: \text{LQ} \rightarrow (\text{Integer}, \text{LR})$

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: $LR \rightarrow LQ$

Parameter	Type	Description
A	LR	An integral element

Returns

Returns A as a rational element.

Usage

$(a, B) := \text{normalize } A$

Signature

normalize: $LR \rightarrow (R, LQ)$

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 is 1, for vectors that the first coordinate is 1, etc.

Usage

ModularComputation: Category

Description

ModularComputation is the category of domains that support modular algorithm such as the modular gcd algorithm. More precisely, for every element p of $\%$ the operation `residueClassRing` returns a domain implementing the residue class ring R/p . In addition, if $\%$ is an Euclidean domain, the operation `combine` implements the Chinese Remaindering algorithm.

Exports

`CanonicalSimplification`

`residueClassRing: (p: %) → ResidueClassRing (% ,p)` Residue class ring

if $\%$ has `EuclideanDomain` then

`combine: (% , %) → (% , %) → %` Chinese remaindering algorithm

Product

Usage

import from Product R

Parameter	Type	Description
R	<code>CommutativeRing</code>	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`

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

Usage

p

Signatures

#: % → MachineInteger

Parameter	Type	Description
<i>p</i>	%	A product

Returns

Returns the number of terms in the product p.

Usage

for d in divisors p repeat { ... }

Signature

divisors: % \rightarrow 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: % \rightarrow R

Parameter	Type	Description
<i>p</i>	%	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: $\% \rightarrow (\mathbb{R}, \mathbb{R})$

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	$R \rightarrow M$	a logarithmic function on R
p	%	A product

Description

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

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) \rightarrow %

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.

ResidueClassRing

Usage

ResidueClassRing (R,p): Category

Parameter	Type	Description
R	CommutativeRing	The ring
p	R	The modulo

Description

ResidueClassRing (R,p) specifies an implementation of the residue class ring R/p . The canonical pre-image of an element x of % is the element r of R such that x is a canonical representative of r and r equals its canonical associate.

Exports

CommutativeRing

modularRep: $R \rightarrow \%$ Residue class

canonicalPreImage: $\% \rightarrow R$ Canonical pre-image

if R has EuclideanDomain then

symmetricPreImage: $\% \rightarrow R$ Symmetric pre-image

ReducibleModulusException

Usage

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

Parameter	Type	Description
R	<code>CommutativeRing</code>	A commutative ring
m	<code>R</code>	The modulus
a	<code>R</code>	A proper factor of m

Description

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

ReducibleModulusExceptionType

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	UnivariatePolynomialAlgebra 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)
```


SimpleAlgebraicExtensionCategory

Usage

`SimpleAlgebraicExtensionCategory(R, Rx):Category`

Parameter	Type	Description
R	<code>IntegralDomain</code>	The coefficient ring of the polynomials
Rx	<code>UnivariatePolynomialAlgebra R</code>	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 `UnivariatePolynomialQuotientSqfr` when p is known to be reducible but squarefree, and `UnivariatePolynomialQuotient` when p is known to be reducible and not squarefree.

Remarks

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

Exports

`IntegralDomain`

`UnivariatePolynomialQuotientSqfr(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`

SourceOfPrimes

Usage

SourceOfPrimes: Category

Description

SourceOfPrimes is the category of domains supporting a (partial) primality test and a source of primes (which may finite or infinite).

Exports

CommutativeRing

prime?: % → Partial Boolean Primality test

prime?: % → Boolean Primality test

getPrime: () → Partial % Prime source seed

nextPrime: () → Partial % Prime source next

if % has EuclideanDomain then

getPrimeOfSize: () → Partial % Prime source seed

Usage

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

Parameter	Type	Description
R	<code>CommutativeRing</code>	The coefficient ring of the polynomials
Rx	<code>UnivariatePolynomialAlgebra R</code>	The type of the modulus
p	<code>Rx</code>	The modulus
x	<code>Symbol</code>	The generator name (optional)

Description

`UnivariatePolynomialMod(R, Rx, p)` implements the univariate polynomials modulo p , *i.e.* the ring $Rx/(p)$, where p is assumed to be monic, but with no other restrictions. Use instead the types `UnivariatePolynomialModSqfr` when p is also known to be squarefree, and `SimpleAlgebraicExtension` when p is also known to be irreducible.

Exports

```
UnivariatePolynomialQuotient(R, Rx)
```

Usage

```
import from UnivariatePolynomialModSqfr(R, Rx, p)
import from UnivariatePolynomialModSqfr(R, Rx, p, x)
```

Parameter	Type	Description
R	<code>CommutativeRing</code>	The coefficient ring of the polynomials
Rx	<code>UnivariatePolynomialAlgebra R</code>	The type of the modulus
p	<code>Rx</code>	The modulus
x	<code>Symbol</code>	The generator name (optional)

Description

`UnivariatePolynomialModSqfr(R, Rx, p)` implements the univariate polynomials modulo p , *i.e.* the ring $Rx/(p)$, where p is assumed to be monic and squarefree, but not necessarily irreducible. Use instead the types `UnivariatePolynomialMod` when p is not squarefree, and `SimpleAlgebraicExtension` when p is known to be irreducible.

Exports

```
UnivariatePolynomialQuotientSqfr(R, Rx)
```

Usage

UnivariatePolynomialQuotient(R, Rx):Category

Parameter	Type	Description
R	CommutativeRing	The coefficient ring of the polynomials
Rx	UnivariatePolynomialAlgebra R	A polynomial type over R

Description

UnivariatePolynomialQuotient(R, Rx) is the category of extensions of R of the form $Rx/(p)$ for some $p \in Rx$, where p is assumed to be monic, but with no other restrictions. Use instead the categories `UnivariatePolynomialQuotientSqfr` when p is also known to be squarefree, and `SimpleAlgebraicExtensionCategory` when p is also known to be irreducible.

Exports

Algebra R		
CommutativeRing		
coefficient:	$(\%, \mathbb{Z}) \rightarrow R$	Extraction of a coefficient
compose:	$\% \rightarrow \% \rightarrow \%$	Modular composition
definingPolynomial:	Rx	Defining polynomial
generator:	$\% \rightarrow \text{Generator Cross}(R, \mathbb{Z})$	iterate through all the terms
knownIrreducible?:	Boolean	Irreducible modulus?
lift:	$\% \rightarrow Rx$	Conversion to a polynomial
map:	$(R \rightarrow R) \rightarrow \% \rightarrow \%$	Lift a mapping
map!:	$(R \rightarrow R) \rightarrow \% \rightarrow \%$	Lift a mapping
monom:	$\%$	Generator of the algebra
reduce:	$Rx \rightarrow \%$	Reduction of a polynomial
value:	$(P:\text{POLY } \%) \rightarrow (P, \mathbb{Z}, \mathbb{Z}) \rightarrow \%$	Evaluation at a rational number

where

Z == Integer
POLY == UnivariatePolynomialAlgebra

if R has CharacteristicZero then

CharacteristicZero

if R has FiniteCharacteristic then

FiniteCharacteristic

if R has FiniteSet then

FiniteSet

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 $q_1(p), \dots, q_k(p)$ 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: $R[x]$

Returns

Returns the polynomial p such that the extension is $R[x]/(p)$.

Usage

knownIrreducible?

Signature

knownIrreducible?: `Boolean`

Returns

Returns *true* if the modulus is known to be irreducible, *false* otherwise. Note that the modulus could be irreducible, even if `knownIrreducible?` is *false*.

Usage

lift q

Signaturelift: % \rightarrow Rx

Parameter	Type	Description
q	%	An element of the algebraic extension

ReturnsReturns 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

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:UnivariatePolynomialAlgebra %) → (P, Integer, Integer) → %

Parameter	Type	Description
P	UnivariatePolynomialAlgebra %	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

UnivariatePolynomialQuotientSqfr(R, Rx):Category

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

Description

UnivariatePolynomialQuotientSqfr(R, Rx) is the category of extensions of R of the form $Rx/(p)$ for some $p \in Rx$, where p is assumed to be monic and squarefree, but not necessarily irreducible. Use instead the categories **UnivariatePolynomialQuotient** when p is not squarefree, and **SimpleAlgebraicExtensionCategory** when p is known to be irreducible.

Exports

```

UnivariatePolynomialQuotient(R, Rx)
newtonSeries: Rxx          Newton series of the generator
norm:         % → R        Norm
              (P:POLY %) → P → Rx
trace:        % → R        Trace
              (P:POLY %) → P → Rx

```

where

```

POLY == UnivariatePolynomialAlgebra
Rxx  == DenseUnivariateTaylorSeries R

```

if R has RationalRootRing then

```
RationalRootRing
```

if R has CharacteristicZero and R has Field then

```
DifferentialExtension R
```

Usage

newtonSeries

Signature

newtonSeries: DenseUnivariateTaylorSeries R

Returns

Returns the series

$$\sum_{n \geq 0} Tr(\alpha^n) x^n$$

where α is the image in the quotient of the term x from Rx , and Tr is the trace from the quotient into R .

See also

monom, trace

Usage

```

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

```

Signatures

```

norm,trace:  % → R
norm,trace:  (P:UnivariatePolynomialAlgebra %) → P → Rx

```

Parameter	Type	Description
q	<code>%</code>	An element of the algebraic extension
P	<code>UnivariatePolynomialAlgebra %</code>	A polynomial type
p	<code>P</code>	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.

SmallPrimes

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 than half-word-size, which allows for accumulation before reduction. Only a specific set \mathcal{P} of half-word-size primes is available.

Exports

```
PrimeCollection
```

HalfWordSizePrimes

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
```

WordSizePrimes

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
```

PrimeCollection

Usage

PrimeCollection: Category

Description

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

Exports

allPrimes:	$() \rightarrow \text{Generator } Z$	Generate all the primes
fourierPrime:	$Z \rightarrow (Z, Z)$	Fourier prime
maxPrime:	$\rightarrow Z$	Largest prime
nextPrime:	$Z \rightarrow Z$	First prime above a given number
previousPrime:	$Z \rightarrow Z$	First prime below a given number
primeInCollection?:	$Z \rightarrow \text{Boolean}$	Check a prime
primRoot:	$Z \rightarrow Z$	Modular primitive root
randomPrime:	$() \rightarrow Z$	Random prime

where

$Z == \text{MachineInteger}$

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` \rightarrow (`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
<i>n</i>	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
<i>n</i>	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

primeInCollection? *n*

Signature

primeInCollection?: MachineInteger → Boolean

Parameter	Type	Description
<i>n</i>	MachineInteger	An integer

Returns

Returns *true* if *n* is a prime in the collection, *false* otherwise (*n* could still be prime in that case).

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

PrimeField2 implements the finite field with two elements.

Exports

```
SmallPrimeFieldCategory
```

PrimeFieldCategory

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 %) \rightarrow P \rightarrow List Cross(% , Integer) In-field roots

rootsSqfr: (P:POLY %) \rightarrow P \rightarrow List % In-field roots

where

POLY == UnivariatePolynomialAlgebra0

Usage

rootsSqfr P
rootsSqfr(P)(p)

Signature

rootsSqfr: (P:UnivariatePolynomialAlgebra0 %) → P → Generator %

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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.

PrimeFieldCategory0

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

FiniteField

SerializableType

lift: % \rightarrow Integer Conversion to an integer

Usage

lift x

Signature

lift: % \rightarrow Integer

Parameter	Type	Description
<i>x</i>	%	an element of the field

Returns

Return x seen as an integer.

PrimitiveRoots

Usage

import from PrimitiveRoots

Description

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

Exports

factors: $Z \rightarrow \text{List } Z$ List of prime factors
primitiveRoot: $Z \rightarrow Z$ Modular primitive root

where

$Z == \text{MachineInteger}$

Usage

factors n

Signature

factors: `MachineInteger` \rightarrow `List MachineInteger`

Parameter	Type	Description
n	<code>MachineInteger</code>	An integer

Returns

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

Usage

primitiveRoot p

Signature

primitiveRoot: MachineInteger \rightarrow 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: $(T, \text{Integer}) \rightarrow T$ p^{th} -powering
pExponentiation!: $(T, \text{Integer}) \rightarrow T$ In-place p^{th} -powering

Usage

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

Signature

pExponentiation: $(R, \text{Integer}) \rightarrow R$

Parameter	Type	Description
a	<code>R</code>	The element to exponentiate
n	<code>Integer</code>	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.

SmallPrimeField

Usage

import from SmallPrimeField p

Parameter	Type	Description
<i>p</i>	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

SmallPrimeField0

Usage

import from SmallPrimeField0 p

Parameter	Type	Description
<i>p</i>	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

SmallPrimeFieldCategory

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

SmallPrimeFieldCategory0

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 \rightarrow % conversion to a field element

discreteLogTable: Cross(A, A, Boolean) discrete log table if available

machine: % \rightarrow MachineInteger conversion to a machine integer

where

A == PrimitiveArray MachineInteger

Usage

n::%
machine m

Signatures

coerce: MachineInteger \rightarrow %
machine: % \rightarrow MachineInteger

Parameter	Type	Description
<i>m</i>	%	an element of the field
<i>n</i>	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`).

ZechPrimeField

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
```

Backsolve

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 \rightarrow Z,ARR Z,Z,Z,R) \rightarrow V R Backsolve a triangular system
backsolve: (M,Z \rightarrow Z,ARR Z,Z,M,R) \rightarrow (M, V R) Backsolve a triangular system

if R has GcdDomain then

backsolve: (M,Z \rightarrow Z,ARR Z,Z,Z) \rightarrow V R Backsolve a triangular system
backsolve: (M,Z \rightarrow Z,ARR Z,Z,M) \rightarrow (M, V R) Backsolve a triangular system
backsolve2: (M,Z \rightarrow Z,ARR Z,Z,Z,R) \rightarrow V R Backsolve a triangular system
backsolve2: (M,Z \rightarrow Z,ARR Z,Z,M,R) \rightarrow (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
<i>a</i>	M	A matrix representing a Row Echelon Form (REF)
<i>p</i>	$Z \rightarrow Z$	A permutation of the rows of <i>a</i>
<i>st</i>	PrimitiveArray Z	The stairs of the REF
<i>r</i>	Z	The number of leading columns (before the c^{th} column)
<i>c</i>	Z	The column to be backsolved
<i>b</i>	M	A matrix whose columns have to be backsolved
<i>d</i>	R	A maximal denominator needed for a dependence relation

where

$Z == \text{MachineInteger}$

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
<i>a</i>	M	A matrix representing a Row Echelon Form (REF)
<i>p</i>	$Z \rightarrow Z$	A permutation of the rows of <i>a</i>
<i>st</i>	PrimitiveArray Z	The stairs of the REF
<i>r</i>	Z	The number of leading columns (before the c^{th} column)
<i>c</i>	Z	The column to be backsolved
<i>b</i>	M	A matrix whose columns have to be backsolved
<i>d</i>	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.

DenseMatrix

Usage

import from DenseMatrix R

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

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

Exports

MatrixCategory R

DivisionFreeGaussElimination

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.

FractionFreeGaussElimination

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.

HermiteGaussElimination

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	<code>CommutativeRing</code>	The coefficient domain
M	<code>MatrixCategory R</code>	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 $([a_0, \dots, a_n], m)$ 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 $v, f(v), \dots, f^n(v)$ (resp. $v, Av, \dots, A^n v$).

Remarks

The relation is as small as possible, meaning that $v, f(v), \dots, f^{n-1}(v)$ (resp. $v, Av, \dots, A^{n-1}v$) 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: $M \rightarrow R$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns the determinant of *a*.

See also

factorOfDeterminant

Usage

factorOfDeterminant a

Signature

factorOfDeterminant: $M \rightarrow (\text{Boolean}, R)$

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
<i>gen</i>	Generator Vector R	A generator of vectors
<i>n</i>	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

Signatureinverse: $M \rightarrow (M, \text{Vector } R)$

Parameter	Type	Description
a	M	A matrix

ReturnsReturns $(b, [d_1, \dots, d_n])$ 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, \dots, 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 $\text{transpose}(a)$. 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: $M \rightarrow (\text{Boolean}, \text{Array MachineInteger}, \text{Array MachineInteger})$

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(\text{max?}, [r_1, \dots, r_r], [c_1, \dots, c_r])$ 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: $M \rightarrow M$

Parameter	Type	Description
<i>a</i>	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: $M \rightarrow (\text{Array MachineInteger}, \text{Array MachineInteger})$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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)

SignatureparticularSolution: (M, M) → (M, **Vector** R)

Parameter	Type	Description
a, b	M	Matrices

ReturnsReturns $(m, [d_1, \dots, d_n])$ 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: $M \rightarrow \text{MachineInteger}$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns the rank of *a*.

See also

rankLowerBound, span

Usage

rankLowerBound a

Signature

rankLowerBound: $M \rightarrow (\text{Boolean}, \text{MachineInteger})$

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(\text{rank?}, r)$ 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: $M \rightarrow \text{Array MachineInteger}$

Parameter	Type	Description
a	M	A matrix

Returns

Returns $[c_1, \dots, c_r]$ where r is the rank of a and the span of a is generated by its columns c_1, \dots, 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 $(w, m, [d_1, \dots, d_n])$ 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
<i>a</i>	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

LinearAlgebraRing: 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 $(b, [d_1, \dots, d_n])$ 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 $(max?, [r_1, \dots, r_r], [c_1, \dots, c_r])$ 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) \rightarrow Vector R

Parameter	Type	Description
<i>gen</i>	Generator Vector R	A generator of vectors
<i>n</i>	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 $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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 $(m, [d_1, \dots, d_n])$ 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
<i>M</i>	MatrixCategory %	A matrix type
<i>a</i>	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	<code>MatrixCategory %</code>	A matrix type
a, b	<code>M</code>	Matrices

Returns

Returns $(w, m, [d_1, \dots, d_n])$ 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 $[c_1, \dots, c_r]$ where r is the rank of a and the span of a is generated by its columns c_1, \dots, 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 computing a row echelon form (REF) of a matrix and the first dependence relation among vectors.

Exports

extendedRowEchelon:	$M \rightarrow (M, Z \rightarrow Z, Z, \text{ARR } Z, Z, M)$	REF of a matrix
extendedRowEchelon!:	$M \rightarrow (Z \rightarrow Z, Z, \text{ARR } Z, Z, M)$	REF of a matrix
extendedRowEchelonForm:	$M \rightarrow (M, M)$	REF of a matrix
maxInvertibleSubmatrix:	$M \rightarrow (\text{Array } Z, \text{Array } Z)$	Maximal minor
maxInvertibleSubmatrix!:	$M \rightarrow (\text{Array } Z, \text{Array } Z)$	Maximal minor
pivot:	$(M, Z \rightarrow Z, Z, Z) \rightarrow Z$	Select a pivot
rowEchelon:	$M \rightarrow (M, Z \rightarrow Z, Z, \text{ARR } Z, Z)$	REF of a matrix
rowEchelon!:	$M \rightarrow (Z \rightarrow Z, Z, \text{ARR } Z, Z)$	REF of a matrix
rowEchelonForm:	$(M, M) \rightarrow (Z \rightarrow Z, Z, \text{ARR } Z, Z)$	REF of a matrix
span:	$M \rightarrow M$	REF of a matrix
span!:	$M \rightarrow \text{Array } Z$	Span of a matrix
span!:	$M \rightarrow \text{Array } Z$	Span of a matrix

if R has IntegralDomain then

denominators:	$(M, Z \rightarrow Z, Z, \text{ARR } Z) \rightarrow \text{ARR } R$	Maximal denominators
dependence:	$(\text{Generator } V, Z) \rightarrow (M, Z \rightarrow Z, Z, R)$	First linear dependence
deter:	$(M, Z \rightarrow Z, Z, Z) \rightarrow R$	Determinant
determinant:	$M \rightarrow R$	Determinant
determinant!:	$M \rightarrow R$	Determinant
rank!:	$M \rightarrow Z$	Rank of a matrix
rank:	$M \rightarrow Z$	Rank of a matrix

where

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

Usage

denominators(*a*,*p*,*r*,*st*)

Signature

denominators: (M,Z \rightarrow Z,Z, PrimitiveArray Z) \rightarrow PrimitiveArray R

where

Z == MachineInteger

Parameter	Type	Description
<i>a</i>	M	A matrix in REF form
<i>p</i>	MachineInteger \rightarrow MachineInteger	A permutation of the rows of <i>a</i>
<i>r</i>	MachineInteger	The number of stairs of the REF
<i>st</i>	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) \rightarrow (M,Z \rightarrow Z,Z,R)

where

Z == MachineInteger

Parameter	Type	Description
<i>gen</i>	Generator Vector R	A generator of vectors
<i>n</i>	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
<i>a</i>	M	A matrix in REF form
<i>p</i>	MachineInteger → MachineInteger	A permutation of the rows of <i>a</i>
<i>r</i>	MachineInteger	The number of stairs of the REF
<i>d</i>	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: $M \rightarrow R$

Parameter	Type	Description
-----------	------	-------------

<i>a</i>	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: $M \rightarrow (M, Z \rightarrow Z, Z, \text{PrimitiveArray } Z, Z, M)$
 extendedRowEchelon!: $M \rightarrow (Z \rightarrow Z, Z, \text{PrimitiveArray } Z, Z, M)$

where

$Z == \text{MachineInteger}$

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 $j_1 < j_2 < \dots < j_r$ (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: $M \rightarrow (M, M)$

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 $j_1 < j_2 < \dots < j_r$ (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: $M \rightarrow (\text{Array MachineInteger}, \text{Array MachineInteger})$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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
<i>a</i>	M	A matrix
<i>p</i>	MachineInteger → MachineInteger	A permutation
<i>c</i>	MachineInteger	A column index
<i>r</i>	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: $M \rightarrow \text{MachineInteger}$

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 $j_1 < j_2 < \dots < j_r$ (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: $M \rightarrow M$

Parameter	Type	Description
<i>a</i>	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 $j_1 < j_2 < \dots < j_r$ (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: $M \rightarrow \text{Array MachineInteger}$

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

Returns

Returns $[c_1, \dots, c_r]$ where r is the rank of a and the span of a is generated by its columns c_1, \dots, 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	
*	$(\%, V) \rightarrow V$	multiplication by a vector
[]	$\text{Tuple } V \rightarrow \%$ $\text{Generator } V \rightarrow \%$	create a matrix
apply:	$(\%, I, I) \rightarrow R$ $(\%, I, I, I, I) \rightarrow \%$ $(\%, \text{Array } I, \text{Array } I) \rightarrow \%$	extract an entry extract a submatrix
colCombine!:	$(\%, R, I, R, I) \rightarrow \%$ $(\%, R, I, R, I, I, I) \rightarrow \%$ $(\%, (R, R) \rightarrow R, I, I) \rightarrow \%$ $(\%, (R, R) \rightarrow R, I, I, I, I) \rightarrow \%$	In-place linear combination of columns
colSwap!:	$(\%, I, I) \text{ to } \%$ $(\%, I, I, I, I) \text{ to } \%$	Swap columns in-place
column:	$(\%, I) \rightarrow V$	extraction of a column
columns:	$\% \rightarrow \text{Generator } V$	iteration over the columns
companion:	$V \rightarrow \%$ $(V, R) \rightarrow \%$	creates a companion matrix
diagonal:	$V \rightarrow \%$ $(R, I) \rightarrow \%$	creates a diagonal matrix
diagonal?:	$\% \rightarrow \text{Boolean}$	test for a diagonal matrix
dimensions:	$\% \rightarrow (I, I)$	get row and column dimensions
map:	$(R \rightarrow R) \rightarrow V \rightarrow \%$	lift a mapping
map:	$(R \rightarrow R) \rightarrow \% \rightarrow \%$	lift a mapping
map!:	$(R \rightarrow R) \rightarrow \% \rightarrow \%$	lift a mapping
numberOfColumns:	$\% \rightarrow I$	number of columns of the matrix
numberOfRows:	$\% \rightarrow I$	number of rows of the matrix
one:	$I \rightarrow \%$	identity matrix
one?:	$\% \rightarrow \text{Boolean}$	test for an identity matrix
row:	$(\%, I) \rightarrow V$	extraction of a row

<code>rowCombine!:</code>	$(\%, R, I, R, I) \rightarrow \%$ $(\%, R, I, R, I, I, I) \rightarrow \%$ $(\%, (R, R) \rightarrow R, I, I) \rightarrow \%$ $(\%, (R, R) \rightarrow R, I, I, I, I) \rightarrow \%$	In-place linear combination of rows
<code>rowSwap!:</code>	$(\%, I, I) \text{ to } \%$ $(\%, I, I, I, I) \text{ to } \%$	Swap rows in-place
<code>rows:</code>	$\% \rightarrow \text{Generator } V$	iteration over the rows
<code>scalar?:</code>	$\% \rightarrow \text{Boolean}$	test for a scalar matrix
<code>set!:</code>	$(\%, I, I, R) \rightarrow R$	set an entry in the matrix
<code>setMatrix!:</code>	$(\%, I, I, \%) \rightarrow \%$	modify a submatrix of a matrix
<code>square?:</code>	$\% \rightarrow \text{Boolean}$	test for a square matrix
<code>tensor:</code>	$(\%, \%) \rightarrow \%$	tensor product
<code>transpose:</code>	$V \rightarrow \%$	transpose a vector
<code>transpose:</code>	$\% \rightarrow \%$	transpose a matrix
<code>transpose!:</code>	$\% \rightarrow \%$	transpose a matrix in-place
<code>zero:</code>	$(I, I) \rightarrow \%$	create a zero matrix
<code>zero!:</code>	$\% \rightarrow ()$	make all the entries zero
<code>zero?:</code>	$\% \rightarrow \text{Boolean}$	test if all entries are zero

if R has Ring then

`random:` $() \rightarrow \%$ create a random matrix
 $(I, I) \rightarrow \%$

if R has DifferentialRing then

`wronskian:` $V \rightarrow \%$ Wronskian matrix

where

`I` == MachineInteger

`V` == Vector R

if R has SerializableType then

SerializableType

Usage $A * v$ **Signature** $*: (\%, \text{Vector } R) \rightarrow \text{Vector } R$

Parameter	Type	Description
A	$\%$	a matrix
v	$\text{Vector } R$	a vector

ReturnsReturns the vector Av .

Usage

$[v_1, \dots, v_n]$
 $[v \text{ for } v \text{ in } g]$

Signatures

$[]$: Tuple Vector R \rightarrow %
 $[]$: Generator Vector R \rightarrow %

Parameter	Type	Description
v_1, \dots, 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

$A(n, m)$
`apply(A,n,m)`
 $A(n, m, r, c)$
`apply(A,n,m,r,c)`
 $A(a, b)$
`apply(A, a, b)`

Signatures

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

Parameter	Type	Description
A	<code>%</code>	A matrix
n,m,r,c	<code>MachineInteger</code>	indices
a,b	<code>Array MachineInteger</code>	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. For more general submatrices, $A([a_1, \dots, a_r], [b_1, \dots, b_s])$ returns the submatrix of A consisting of the intersection of the rows a_1, \dots, a_r and columns b_1, \dots, b_r of A .

Usage

```

colCombine!(A, c1, j1, c2, j2)
colCombine!(A, c1, j1, c2, j2, i1, i2)
colCombine!(A, f, j1, j2)
colCombine!(A, f, j1, j2, i1, i2)
rowCombine!(A, c1, i1, c2, i2)
rowCombine!(A, c1, i1, c2, i2, j1, j2)
rowCombine!(A, f, i1, i2)
rowCombine!(A, f, i1, i2, j1, j2)

```

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	$(R,R) \rightarrow R$	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^{th} column (resp. i_1^{th} row) of A is replaced by the result of applying f pointwise to its j_1^{th} and j_2^{th} columns (resp. i_1^{th} and i_2^{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, j1, j2)
colSwap!(A, j1, j2, i1, i2)
rowSwap!(A, i1, i2)
rowSwap!(A, i1, i2, j1, j2)
```

Signatures

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

where

```
I == MachineInteger
```

Parameter	Type	Description
A	<code>%</code>	A matrix
$j1, j2$	<code>MachineInteger</code>	column indices
$i1, i2$	<code>MachineInteger</code>	row indices

Description

The j_1^{th} and j_2^{th} columns (resp. i_1^{th} and i_2^{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)

SignaturecolSwap!,rowSwap!: (%MachineInteger) \rightarrow Vector R

Parameter	Type	Description
A	%	A matrix
n	MachineInteger	An index

ReturnsReturns 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: % \rightarrow Generator Vector R

Parameter	Type	Description
<i>A</i>	%	A matrix

Description

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

Usage

```
companion [r1,...,rn]
companion([r1,...,rn],a)
```

Signature

```
companion: (Vector R, R) -> %
```

Parameter	Type	Description
r_1, \dots, 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 [r1, ..., rn]`
`diagonal(r, n)`
`diagonal? A`
`scalar? A`
`square? A`

Signatures

`diagonal:` `Vector R → %`
`diagonal:` `(R, MachineInteger) → %`
`diagonal?, scalar?, square?:` `% → Boolean`

Parameter	Type	Description
r, r_1, \dots, r_n	<code>R</code>	Entries
n	<code>MachineInteger</code>	A size
A	<code>%</code>	A matrix

Description

`diagonal([r1, ..., rn])` returns a diagonal matrix whose diagonal elements are r_1, \dots, r_n , while `diagonal(r, n)` returns an $n \times n$ diagonal matrix with r on its diagonal, and `square?(A)` (resp. `diagonal A` and `scalar? A`) return *true* if A is a square (resp. diagonal and diagonal with the same entry on its diagonal) matrix, *false* otherwise.

See also

`companion, one?`

Usage

dimensions A

Signature

dimensions: % \rightarrow (MachineInteger,MachineInteger)

Parameter	Type	Description
<i>A</i>	%	A matrix

Returns

The number of rows and columns in *A*

See also

numberOfColumns,numberOfRows

Usage

```
map f
map! f
map(f)([v1, ..., vn])
map(f)(A)
map!(f)(A)
```

Signatures

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

Parameter	Type	Description
f	$R \rightarrow R$	a map
v_i	R	Entries of a vector
A	$\%$	A matrix

Description

$\text{map}(f)([v_1, \dots, v_n])$ returns the square matrix

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

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

Usage

numberOfColumns A
numberOfRows A

Signature

numberOfColumns,numberOfRows: % \rightarrow 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
<i>n,m</i>	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
<i>A</i> , <i>B</i>	<i>%</i>	Matrices
<i>n</i> , <i>m</i>	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	<code>Vector R</code>	A vector
A	<code>%</code>	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 [v₁, ..., v_n]

Signature

wronskian: Vector R → %

Parameter	Type	Description
v _i	R	Entries of a vector

Description

wronskian([v₁, ..., v_n]) returns the square matrix

$$\begin{pmatrix} v_1 & \dots & v_n \\ v'_1 & \dots & v'_n \\ \vdots & & \vdots \\ v_1^{(n-1)} & \dots & v_n^{(n-1)} \end{pmatrix}$$

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 MatrixCategory2(R, MR, S, MS)
```

Parameter	Type	Description
R, S	ExpressionType ArithmeticType	Coefficient domains
MR	MatrixCategory R	a matrix type over R
MS	MatrixCategory S	a matrix type over S

Description

MatrixCategory2(R,MR,S,MS) provides tools for lifting maps $R \rightarrow S$ to maps $MR \rightarrow MS$.

Exports

```
map: (R → S) → MR → MS  Lift a mapping
```

Usage

```
map f
map(f)(m)
```

Signature

```
map: (R → S) → MR → MS
```

Parameter	Type	Description
f	$R \rightarrow S$	A map
m	MR	A matrix with entries in R

Description

`map(f)(m)` returns a matrix whose $(i, j)^{\text{th}}$ entry is $f(m_{ij})$, while `map(f)` returns the mapping $m \rightarrow f(m)$.

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

$(v, A) := \text{makeColIntegral } B$
 $(a, v) := \text{makeColIntegral}(B, i)$

Signatures

$\text{makeColIntegral}: \text{MQ} \rightarrow (\text{Vector } R, \text{MR})$
 $\text{makeColIntegral}: (\text{MQ}, \text{MachineInteger}) \rightarrow (R, \text{Vector } R)$

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

Returns

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

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

$\text{makeColIntegral}(B, i)$ 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

$(v, A) := \text{makeRowIntegral } B$
 $(a, v) := \text{makeRowIntegral}(B, i)$

Signatures

$\text{makeRowIntegral}: \text{MQ} \rightarrow (\text{Vector } R, \text{MR})$
 $\text{makeRowIntegral}: (\text{MQ}, \text{MachineInteger}) \rightarrow (R, \text{Vector } R)$

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

Returns

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

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

$\text{makeRowIntegral}(B, i)$ 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
<i>B</i>	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:	$(M, Z, V, Z, Z, Z) \rightarrow Z$	Determinant
determinant!:	$(M, Z, Z) \rightarrow Z$	Determinant
extendedRowEchelon!:	$(M, Z, Z, Z) \rightarrow (V, Z, V, M)$	REF of a matrix
firstDependence!:	$(\text{Generator } V, Z, Z, M, M) \rightarrow Z$	First dependence relation
inverse!:	$(M, Z, Z, M, V) \rightarrow ()$	Inverse
kernel!:	$(M, Z, Z, M) \rightarrow Z$	Kernel
maxInvertibleSubmatrix!:	$(M, Z, Z) \rightarrow (V, V)$	Maximal minor
particularSolution!:	$(M, Z, Z, M, Z, Z, M, V) \rightarrow ()$	A solution
rank!:	$(M, Z, Z, Z) \rightarrow Z$	Rank
rowEchelon!:	$(M, Z, Z, Z) \rightarrow (V, Z, V)$	REF of a matrix
solve!:	$(M, Z, Z, M, Z, Z, M, V, M) \rightarrow Z$	All solutions
span!:	$(M, Z, Z) \rightarrow V$	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 \mathbb{Z}, \mathbb{Z} , PrimitiveArray $\mathbb{Z}, \mathbb{Z}, \mathbb{Z}$) $\rightarrow \mathbb{Z}$

where

$\mathbb{Z} == \text{MachineInteger}$

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) \rightarrow (A Z, Z, A Z, Z, A A Z)

where

Z == MachineInteger

A == PrimitiveArray

Parameter	Type	Description
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	MachineInteger	A prime

Description

We say that a matrix *a* is in REF if there are *r* (the rank) and $j_1 < j_2 < \dots < j_r$ (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
<i>gen</i>	Generator A MachineInteger	A generator of vectors
<i>n</i>	MachineInteger	The dimension of the vectors generated
<i>p</i>	MachineInteger	a prime
<i>a</i>	A A MachineInteger	A work matrix
<i>d</i>	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
<i>a</i> , <i>b</i>	PrimitiveArray PrimitiveArray MachineInteger	Square matrices
<i>n</i>	MachineInteger	The size of <i>a</i> , <i>b</i> and <i>d</i>
<i>p</i>	MachineInteger	a prime
<i>d</i>	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
<i>a</i> , <i>w</i>	PrimitiveArray PrimitiveArray MachineInteger	Matrices
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	MachineInteger	a prime

Returns

Returns $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	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 $j_1 < j_2 < \dots < j_r$ (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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	MachineInteger	a prime

Returns

Returns $[c_1, \dots, c_r]$ 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, \dots, c_r .

Remarks

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

OrdinaryGaussElimination

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.

OverdeterminedLinearSystemSolver

Usage

```
import from OverdeterminedLinearSystemSolver(R, M)
```

Parameter	Type	Description
R	<code>IntegralDomain</code>	A domain
M	<code>MatrixCategory R</code>	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: $M \rightarrow \text{Partial Cross (Boolean, MachineInteger)}$ Probable rank

Usage

rankLowerBound a

Signature

rankLowerBound: $M \rightarrow \text{Partial Cross}(\text{Boolean}, \text{MachineInteger})$

Parameter	Type	Description
a	M	A matrix

Returns

Returns *failed* if specialization failed, otherwise $(\text{rank?}, r)$ 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.

TwoStepFractionFreeGaussElimination

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	UnivariatePolynomialAlgebra 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: $M \rightarrow \text{Integer}$ Degree bound for the determinant
 determinant: $M \rightarrow RX$ Determinant
 $(M, RX) \rightarrow RX$
 $(M, RX, \text{Integer}) \rightarrow RX$

Usage

degreeBound a

Signature

degreeBound: M → Integer

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

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

Usage

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

Signatures

determinant: $M \rightarrow RX$
determinant: $(M, RX) \rightarrow RX$
determinant: $(M, RX, Integer) \rightarrow RX$

Parameter	Type	Description
<i>a</i>	<i>M</i>	A matrix
<i>d</i>	<i>RX</i>	A known factor of $ a $ (optional)
<i>n</i>	<i>Integer</i>	A known degree bound on $ a $ (optional)

Returns

All calls to `determinant` return 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	UnivariatePolynomialAlgebra 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:	$M \rightarrow FX$	Determinant
hermite:	$M \rightarrow M$	Hermite form
kernel:	$M \rightarrow M$	Kernel
maxInvertibleSubmatrix:	$M \rightarrow (AZ, AZ)$	Maximal minor
popov:	$M \rightarrow M$	weak Popov form
rank:	$M \rightarrow \text{MachineInteger}$	Rank
span:	$M \rightarrow AZ$	Span

where

```
AZ == Array MachineInteger
```

Usage

hermite a

Signature

hermite: M → M

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns the Hermite form of *a*.

Vector

Usage

import from Vector R

Parameter	Type	Description
R	ExpressionType AdditiveType	The coefficient domain

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 \rightarrow % zero vector

zero!: % \rightarrow () make all the entries zero

zero?: % \rightarrow Boolean test if all entries are zero

if R has ArithmeticType then

LinearCombinationType R

dot: (% , %) \rightarrow R dot product

tensor: (% , %) \rightarrow % tensor product

if R has Ring then

random: () \rightarrow % random vector

MachineInteger \rightarrow %

Usage`dot(u,v)`**Signature**`dot: (%,%) \rightarrow R`

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

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

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
<i>u, v</i>	%	Vectors with coefficients from R.

Returns

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

Usage

```
zero n
zero! v
zero? v
```

Signatures

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

Parameter	Type	Description
n	<code>MachineInteger</code>	The dimension of the new vector
v	<code>%</code>	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.

VectorOverFraction

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

```
UnivariatePolynomialAlgebra 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

```
UnivariateTaylorSeriesType R
```

FactorizationRing

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 == UnivariatePolynomialAlgebra0

Usage

factor(P)(p)

Signature

factor: (P: UnivariatePolynomialAlgebra0 %) → P → (% , Product P)

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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 ==  UnivariatePolynomialAlgebra0
```

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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.

FFTRing

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 == UnivariatePolynomialAlgebra0

Usage

fft(P)(p,q)

Signature

fft: (P: UnivariatePolynomialAlgebra0 %) → (P, P) → Partial P

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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: UnivariatePolynomialAlgebra0 %) → (P, P, P) → Boolean

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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 GenericModularPolynomialGcdPackage (R, U)

Parameter	Type	Description
R	EuclideanDomain SourceOfPrimes ModularComputation	
U	UnivariatePolynomialAlgebra (R)	

Description

GenericModularPolynomialGcdPackage (R, U) provides a generic modular gcd algorithm. See the paper *On the genericity of the Modular Gcd Algorithm* by Kaltofen and Monagan in the proc. of ISSAC 1999.

Exports

modularGcd: $(U, U) \rightarrow \text{Partial } (U)$ gcd (failure if not enough primes)

HeuristicGcd

Usage

```
import from HeuristicGcd(Z, P)
```

Parameter	Type	Description
Z	IntegerCategory	An integer-like ring
P	UnivariatePolynomialAlgebra0 Z	Polynomials over Z

Description

HeuristicGcd provides an implementation of the HEUGCD algorithm for univariate polynomials over the integers.

Exports

balancedRemainder:	$(Z, Z) \rightarrow Z$	Symmetric Remainder
heuristicGcd:	$(P, P) \rightarrow (\text{Partial } P, P, P)$	the Heuristic gcd algorithm
radixInterpolate:	$(Z, Z) \rightarrow P$	Radix interpolation

Usage

balancedRemainder(*n*, *m*)

Signature

balancedRemainder: $(\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z}$

Parameter	Type	Description
<i>n</i>	\mathbb{Z}	An integer
<i>m</i>	\mathbb{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: $(P, P) \rightarrow (\text{Partial } P, P, P)$

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 $(\text{failed}, p_1, p_2)$ is returned, although this has never been reported.

Usage

radixInterpolate(n, m)

Signature

radixInterpolate: $(\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{P}$

Parameter	Type	Description
n	\mathbb{Z}	A point
m	\mathbb{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	UnivariatePolynomialAlgebra 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: $(P, P) \rightarrow (\text{Partial } P, P, P)$ The Modular Gcd algorithm

Usage

`modularGcd(p_1, p_2)`

Signatures

`modularGcd`: $(P, P) \rightarrow (\text{Partial } P, P, P)$

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.

ModulopUnivariateGcd

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, [l1, ..., lp-1], [e1, ..., ep-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
$[l_0, \dots, l_{p-1}]$	PrimitiveArray MachineInteger	log table
$[e_0, \dots, e_{p-1}]$	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

```
import from PrimeFieldUnivariateFactorizer(F, P)
```

Parameter	Type	Description
F	PrimeFieldCategory0	Coefficient ring of the polynomials
P	UnivariatePolynomialAlgebra F	A polynomial ring

Description

PrimeFieldUnivariateFactorizer provides implementations of various univariate factorization algorithms over a prime field.

Exports

berlekamp:	$P \rightarrow \text{List } P$	Berlekamp's algorithm
cantorZassenhaus:	$P \rightarrow \text{List } P$	Cantor-Zassenhaus algorithm
factor:	$P \rightarrow (F, \text{PROD})$	Factor (default algorithm)
factor:	$(P \rightarrow \text{List } P) \rightarrow (P \rightarrow (F, \text{PROD}))$	Factor (given algorithm)
roots:	$(P, \text{Boolean}) \rightarrow \text{Generator FR } F$	Roots of a polynomial

where

```
FR    == FractionalRoot
PROD  == Product P
```

Usage

berlekamp p

Signature

berlekamp: $P \rightarrow \text{List } P$

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: $P \rightarrow \text{List } P$

Parameter	Type	Description
<i>p</i>	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
<i>p</i>	P	A polynomial.
<i>sqrfree?</i>	Boolean	Indicates whether <i>p</i> 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 .

RationalRootRing

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 == UnivariatePolynomialAlgebra0

Usage

integerRoots(P)(p)
rationalRoots(P)(p)

Signature

integerRoots,rationalRoots: (P: POL %) → P → Generator RationalRoot
where
POL == UnivariatePolynomialAlgebra0

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	A polynomial type
p	P	A polynomial

Returns

Return $[(r_1, e_1), \dots, (r_n, e_n)]$ where the r_i 's are the integer or rational roots of p and have multiplicity e_i .

Resultant

Usage

```
import from Resultant(R,P)
```

Parameter	Type	Description
R	IntegralDomain	Coefficient ring for the polynomials
P	UnivariatePolynomialAlgebra0 R	A polynomial ring

Exports

lastSPRS:	$(P,P) \rightarrow P$	last non-zero remainder in the SPRS
extendedLastSPRS:	$(P,P) \rightarrow (P,P,P)$	extended last non-zero remainder in the SPRS
resultant:	$(P,P) \rightarrow R$	polynomial resultant
SPRS:	$(P,P) \rightarrow \text{List } P$	Subresultant Polynomial Remainder Sequence
subResultantGcd:	$(P,P) \rightarrow P$	GCD

Description

Resultant(R,P) implements polynomial resultant computations.

Usage

lastSPRS(a,b)

Signature

lastSPRS: (P,P) \rightarrow P

Parameter	Type	Description
<i>a, b</i>	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: $(P, P) \rightarrow (P, P, P)$

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: $(P,P) \rightarrow R$

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.

UsageSPRS(*a*,*b*)**Signature**

SPRS: (P,P) → List P

Parameter	Type	Description
<i>a</i> , <i>b</i>	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: $(P,P) \rightarrow P$

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

```
UnivariatePolynomialRing R
```

SparseUnivariatePolynomial

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

```
UnivariatePolynomialAlgebra R
```

Usage

```
import from UnivariateFactorialPolynomial(R, Rx)
```

Parameter	Type	Description
R	Ring	The coefficient domain
Rx	UnivariatePolynomialAlgebra 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 $(x^n)_{n \geq 0}$, where $x^n = x(x-1)\dots(x-n+1)$. Rx is used for representing the factorial polynomials, so you can choose between sparse and dense representations.

Exports

```
UnivariateFreeRing 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 $(x^n)_{n \geq 0}$, while $\text{expand}(q)$ performs the reverse conversion and $\text{trailExpand}(q)$ returns (n, h) such that $q = x^n h$.

Usage

integerRoots p
rationalRoots p

Signature

integerRoots,rationalRoots: % → List FractionalRoot Integer

Parameter	Type	Description
p	%	A factorial polynomial

Returns

Return $[(r_1, e_1), \dots, (r_n, e_n)]$ where the r_i 's are the integer or rational roots of p and have multiplicity e_i .

Usage

UnivariateFreeLinearArithmeticType R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariateFreeLinearArithmeticType 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 `RRing 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	<code>%</code>	A polynomial or series
t	<code>ExpressionTree</code>	An expression tree
x	<code>Symbol</code>	A name for the variables
$port$	<code>TextWriter</code>	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^3+4*x-2
to the standard stream stdout.

```

Usage

for c in coefficients m repeat { ... }

Signature

coefficients: % \rightarrow 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	<code>%</code>	A polynomial or series
m	<code>Integer</code>	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

UnivariateFreeRing R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariateFreeRing 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
IndexedFreeRRing(R, Integer)
UnivariateFreeLinearArithmeticType 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 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	$\text{Vector } R$	A coefficient vector

Description

$[v_1, \dots, v_n]::\%$ returns the polynomial $\sum_{i=0}^{n-1} v_{i+1} P_i$, while $\text{vectorize!}(v)(\sum_{j=0}^d a_j P_j)$ fills v with $[a_0, \dots, a_d, 0, \dots]$ and returns v .

Remarks

If $d > \#v - 1$, then the high coefficients of p are simply ignored.

Usage

companion p

Signature

companion: % \rightarrow 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

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	$() \rightarrow R$	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:  % → %
```

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) \rightarrow %

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 i th coefficient of s equals the $l + i$ th coefficient of pq . Here $l \leq h$ should hold.

Usage

import from UnivariateFreeRing2(*R*,*R**x*,*S*,*S**y*)

Parameter	Type	Description
<i>R</i> , <i>S</i>	ExpressionType ArithmeticType	Coefficient domains
<i>R</i> <i>x</i>	UnivariateFreeRing <i>R</i>	A monogenic algebra over <i>R</i>
<i>S</i> <i>y</i>	UnivariateFreeRing <i>S</i>	A monogenic algebra over <i>R</i>

Description

UnivariateFreeRing2(*R*,*R**x*,*S*,*S**y*) provides tools for lifting maps $R \rightarrow S$ to maps $Rx \rightarrow Sy$.

Exports

map: $(R \rightarrow S) \rightarrow Rx \rightarrow Sy$ Lift a mapping

Usage

```
map f
map(f)(p)
```

Signature

```
map:  (R → S) → Rx → Sy
```

Parameter	Type	Description
f	$R \rightarrow S$	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 $p \rightarrow f(p)$.

UnivariateFreeRingOverFraction

Usage

import from UnivariateFreeRingOverFraction(R , PR , Q , PQ)

Parameter	Type	Description
R	IntegralDomain	an integral domain
PR	UnivariateFreeRing R	a univariate free finite algebra type over R
Q	FractionCategory R	a fraction domain of R
PQ	UnivariateFreeRing R	a univariate free finite algebra type over Q

Description

UnivariateFreeRingOverFraction(R , PR , Q , PQ) provides useful conversions between polynomials with integral and rational coefficients.

Exports

LinearCombinationFraction(R , PR , Q , PQ)

UnivariateGcdRing

Usage

UnivariateGcdRing: Category

Description

UnivariateGcdRing is the category of rings which export a gcd algorithm for univariate polynomials over themselves.

Exports

GcdDomain

gcdUP: (P:UnivariatePolynomialAlgebra0) \rightarrow (P, P) \rightarrow P Gcd

gcdUP!: (P:UnivariatePolynomialAlgebra0) \rightarrow (P, P) \rightarrow P Gcd

gcdquoUP: (P:UnivariatePolynomialAlgebra0) \rightarrow (P, P) \rightarrow (P,P,P) Gcd

Usage

gcdUP(P)(p, q)
 gcdUP!(P)(p, q)

Signature

gcdUP: (P: UnivariatePolynomialAlgebra0 %) → (P, P) → P

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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: UnivariatePolynomialAlgebra0 %) → (P,P) → (P,P,P)

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	A polynomial type
p, q	P	Polynomials

Returns

Returns (g, y, z) such that $g = \gcd(p, q)$, $p = gy$ and $q = gz$.

UnivariateIntegralFactorizer

Usage

import from UnivariateIntegralFactorizer(Z, P)

Parameter	Type	Description
Z	IntegerCategory	An integer-like ring
P	UnivariatePolynomialAlgebra0 Z	A polynomial type over Z

Description

UnivariateIntegralFactorizer(Z , P) implements a factorizer for polynomials with integer coefficients.

Exports

factor:	$P \rightarrow (Z, \text{Product } P)$	Factor
integerRoots:	$P \rightarrow \text{Generator FractionalRoot Integer}$	Integer roots
rationalRoots:	$P \rightarrow \text{Generator FractionalRoot Integer}$	Rational roots

Usage

factor p

Signature

factor: P → (Z,Product P)

Parameter	Type	Description
p	P	A polynomial with integer coefficients

ReturnsReturns $(c, p_1^{e_1} \cdots p_n^{e_n})$ 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: $P \rightarrow \text{Generator FractionalRoot Integer}$

Parameter	Type	Description
p	P	A polynomial with integer coefficients

Returns

integerRoots(p) (resp. rationalRoots(p)) return $[(r_1, e_1), \dots, (r_n, e_n)]$ 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 ArithmeticType	Coefficients of the monomials
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: % \rightarrow 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: % \rightarrow 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 → R) → % → %
```

Parameter	Type	Description
f	$R \rightarrow R$	A map
p	$\%$	A monomial

Description

`map(f)(p)` returns $f(a)x^n$ where $p = ax^n$, while `map(f)` returns the mapping $p \rightarrow f(p)$. 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: % \rightarrow %

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) \rightarrow 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	<code>%</code>	A monomial
n	<code>Integer</code>	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 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

UnivariatePolynomialRing R

IndexedFreeAlgebra(R, Integer)

apply: $(\%, R) \rightarrow R$ Evaluate a polynomial

apply: $(\%, \%) \rightarrow \%$ Evaluate a polynomial

equal?: $(\%, \%, \%, \text{Integer}) \rightarrow \text{Boolean}$ Truncated equality

Horner: $(\%, R) \rightarrow (\%, R)$ Horner division by $x - a$

if R has CharacteristicZero then

ordinaryPoint: $\% \rightarrow \text{Integer}$ Point where a polynomial is nonzero

if R has CommutativeRing then

DifferentialRing

lift: $(\text{Derivation } R, \%) \rightarrow \text{Derivation } \%$ Extend a derivation

monicDivide: $(\%, \%) \rightarrow (\%, \%)$ Polynomial division

monicDivide!: $(\%, \%) \rightarrow (\%, \%)$ Polynomial division

monicDivideBy: $\% \rightarrow \% \rightarrow (\%, \%)$ Polynomial division

monicDivideBy!: $\% \rightarrow \% \rightarrow (\%, \%)$ Polynomial division

monicQuotient: $(\%, \%) \rightarrow \%$ Quotient

monicQuotient!: $(\%, \%) \rightarrow \%$ Quotient

monicQuotientBy: $\% \rightarrow \% \rightarrow \%$ Quotient

monicQuotientBy!: $\% \rightarrow \% \rightarrow \%$ Quotient

monicRemainder: $(\%, \%) \rightarrow \%$ Remainder

monicRemainder!: $(\%, \%) \rightarrow \%$ Remainder

monicRemainderBy: $\% \rightarrow \% \rightarrow \%$ Remainder

monicRemainderBy!: $\% \rightarrow \% \rightarrow \%$ Remainder

if R has CommutativeRing and R has RittRing then

integrate: $\% \rightarrow \%$ Integration

$(\%, \text{Integer}) \rightarrow \%$

if R has FactorizationRing then

factor: $\% \rightarrow (R, \text{Product } \%)$ Factorisation into irreducibles

fractionalRoots: $\% \rightarrow \text{Generator FractionalRoot } R$ Roots in the fraction field

```

roots:  % → Generator FractionalRoot R  Roots in the coefficient ring

if R has Field then
  EuclideanDomain
  rationalReconstruction:  % → % → Partial Cross(%, %)  Rational reconstruction
  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: % \rightarrow 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: $(\%, R) \rightarrow (\%, R)$

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: $\% \rightarrow \%$
 integrate: $(\%, \text{Integer}) \rightarrow \%$

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 $\int \dots \int s(x)dx^n$.

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 $a \rightarrow \text{monicXXX}(a, b)$, while monicXXXBy!(b) returns the map $a \rightarrow \text{monicXXX}!(a, b)$.

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: % \rightarrow 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: $(\%, \%) \rightarrow (\%, \%)$

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: $(\%, \%) \rightarrow \%$

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

rationalReconstruction m
 rationalReconstruction(m)(u)

Signature

rationalReconstruction: $\% \rightarrow \% \rightarrow \text{Partial Cross}(\%, \%)$

Parameter	Type	Description
m	$\%$	A modulus of positive degree
u	$\%$	A polynomial

Returns

rationalReconstruction(m)(u) returns either (a, b) such that $a/b = u \pmod{m}$, $\deg(a) \leq (\deg(m) - 1)/2$ and $\deg(b) \leq (\deg(m) - 1)/2$, or *failed* if no such a, b exist.

Remarks

The resulting a and b are guaranteed to be unique.

See also

rationalReconstruction

Usage

resultant(p, q)

Signature

resultant: (% , %) \rightarrow R

Parameter	Type	Description
p	%	A polynomial
q	%	A polynomial

Returns

Returns the resultant of p and q.

Usage

sparseMultiple(p, n)

Signature

sparseMultiple: $(\%, \text{Integer}) \rightarrow \%$

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: % \rightarrow (R, Product %)
 squareFreePart: % \rightarrow %

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) \rightarrow Generator R

Parameter	Type	Description
p	%	A polynomial
a	R	A scalar

Returns

Returns a generator generating the sequence $p(a), p(a + 1), p(a + 2), \dots$

Remarks

values uses arrays of differences and can be more efficient than 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: % \rightarrow 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: % \rightarrow Generator FractionalRoot Integer

Parameter	Type	Description
p	%	A polynomial

Returns

Return $[(r_1, e_1), \dots, (r_n, e_n)]$ 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:  %  $\rightarrow$  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: (%,%) \rightarrow List Cross(% , Integer)

Parameter	Type	Description
a, b	%	Nonzero polynomials

Returns

Return $[(p_1, e_1), \dots, (p_n, e_n)]$ 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 $u = \prod_{i=1}^n \prod_{j=0}^{e_i} p_i(x - j)$.

UnivariatePolynomialKaratsuba

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

UnivariatePolynomialRing

Usage

UnivariatePolynomialRing R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariatePolynomialRing 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

UnivariateFreeRing R

add!: (% , R, Z, % , Z, Z) \rightarrow % In-place partial product and sum

compose: (% , %) \rightarrow % Compose polynomials

translate: (% , R) \rightarrow % 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 threshold

Description

add!(p, c, m, q, n, N) computes all the terms of degree at least n and at most N of

$$p + cx^mq = \sum_{i=0}^{d+m} (a_i + cb_{i-m})x^i,$$

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

import from UnivariatePolynomialSquareFree(R, P)

Parameter	Type	Description
R	GcdDomain	Coefficient ring of the polynomials
P	UnivariatePolynomialAlgebra0 R	A polynomial ring

Description

UnivariatePolynomialSquareFree provides implementations of various squarefree factorization algorithms.

Exports

- musser: $P \rightarrow (R, \text{Product } P)$ Musser's algorithm
- yun: $P \rightarrow (R, \text{Product } P)$ 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

UnivariateTaylorSeriesType

Usage

UnivariateTaylorSeriesType R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariateTaylorSeriesType R is the category of univariate Taylor series with coefficients in R.

Exports

UnivariateFreeLinearArithmeticType R

degree: % \rightarrow Partial Integer upper bound on the degree
dot: Vector R \rightarrow Vector % \rightarrow % linear combination
finite?: % \rightarrow Boolean check whether the support is finite
series: Sequence R \rightarrow % creation of a series

where

UPC == UnivariatePolynomialAlgebra

if R has CommutativeRing then

differentiate: % \rightarrow % Differentiation
 (%, Integer) \rightarrow %
reciprocal: % \rightarrow Partial % Inverse

if R has CommutativeRing and R has RittRing then

integrate: % \rightarrow % Integration
 (%, Integer) \rightarrow %

if R has FloatType then

reciprocal: % \rightarrow Partial % Inverse

Usage

degree s
finite? s

Signatures

degree: % \rightarrow Partial Integer
finite?: % \rightarrow Boolean

Parameter	Type	Description
<i>s</i>	%	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: $\% \rightarrow \%$
differentiate,integrate: $(\%, \text{Integer}) \rightarrow \%$

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 $\int \dots \int s(x)dx^n$.

Usage

series s

Signature

series: Sequence R \rightarrow %

Parameter	Type	Description
s	Sequence R	a coefficient sequence

Returns

Returns s viewed as a series.

Usage

import from UnivariateTaylorSeriesType2Poly(R, RXX, RX)

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain
RXX	UnivariateTaylorSeriesType R	A series type over R
RX	UnivariatePolynomialAlgebra R	A polynomial type over R

Description

UnivariateTaylorSeriesType2Poly(R, RXX, RX) provides conversion tools between polynomials and series.

Exports

expand: $R \rightarrow RX \rightarrow RXX$ series expansion at a point
truncate: $(RXX, \text{Integer}) \rightarrow RX$ truncation of a series

if R has Field then

expandFraction: $R \rightarrow \text{Fraction } RX \rightarrow (\text{Integer}, RXX)$ series expansion at a point

if R has FloatType then

expandFraction: $R \rightarrow \text{Fraction } RX \rightarrow (\text{Integer}, RXX)$ series expansion at a point

if R has CommutativeRing then

monicNewtonSeries: $RX \rightarrow RXX$ Newton series

tryExpandFraction: $R \rightarrow (RX, RX) \rightarrow (\text{Integer}, \text{Partial } RXX)$ try series expansion

if R has IntegralDomain then

polynomials: $(V \text{ } RXX, \text{Integer}) \rightarrow (V \text{ } RX, M \text{ } R)$ interpolation
 $(V \text{ } RXX, \text{Integer}, \text{Integer}) \rightarrow (V \text{ } RX, M \text{ } R)$

where

V == Vector

M == DenseMatrix

Usage

expand a
expand(a)(p)

Signature

expand: $R \rightarrow RX \rightarrow RXX$

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 $n \leq 0$ 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, tryExpandFraction
```

Usage

polynomials($[s_1, \dots, s_n]$, N)
 polynomials($[s_1, \dots, s_n]$, N, M)

Signatures

polynomials: (Vector RXX, Integer) \rightarrow (Vector RX, DenseMatrix R)
 polynomials: (Vector RXX, Integer, Integer) \rightarrow (Vector RX, DenseMatrix R)

Parameter	Type	Description
s_i	RXX	series
N	Integer	a degree bound
M	Integer	an upper bound

Description

polynomials($[s_1, \dots, s_n]$, N , M) returns $([p_1, \dots, p_s], A)$ such that the series $[s_1, \dots, s_n]A$ all have coefficients 0 from x^{N+1} to x^M , and $[p_1, \dots, p_s]$ are the truncations to order N of $[s_1, \dots, s_n]A$. If the upper bound M is not given, then the series returned have coefficients 0 from x^{N+1} up to an order that is determined heuristically.

Usage

monicNewtonSeries p

Signature

monicNewtonSeries: $RX \rightarrow RXX$

Parameter	Type	Description
p	RX	A monic polynomial

Returns

Returns the series

$$\sum_{n \geq 0} (y_1 + \dots + y_d)^n x^n$$

where y_1, \dots, y_d are all the roots of p .

Usage

```
tryExpandFraction a
tryExpandFraction(a)(p,q)
```

Signature

```
tryExpandFraction:  R → (RX, RX) → (Integer, Partial RXX)
```

Parameter	Type	Description
a	R	the expansion point
p, q	RX	numerator and denominator of a rational function

Returns

If $q(a)$ is a unit in R , then `tryExpandFraction(a)(p,q)` returns (n, s) where $n \leq 0$ and the series expansion of p/q around a is $(x-a)^n s(x-a)$. In addition, $s(a) \neq 0$ whenever $n < 0$. Otherwise, it returns (n, failed) where n is the order of p/q at $x = a$.

See also

```
expand, expandFraction
```

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 ArithmeticType	The coefficient domain
Rx	UnivariateTaylorSeriesType R	Series over R
RxY	UnivariatePolynomialAlgebra Rx	Polynomials over Rx
RX	UnivariatePolynomialAlgebra R	Polynomials over R
RXY	UnivariatePolynomialAlgebra 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:  $RxY \rightarrow RxY$       Differentiation
  root:           $(RXY, R) \rightarrow Rx$     Root of a polynomial
                $(RxY, R) \rightarrow Rx$ 

if  $R$  has FloatType then
  root:  $(RXY, RXY, R) \rightarrow Rx$     Root of a polynomial
        $(RxY, RxY, R) \rightarrow Rx$ 
```

Usage

differentiate p

Signature

differentiate: RxY \rightarrow 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 BivariateUtilitiesPackage (U,V)
```

Parameter	Type	Description
U	UnivariatePolynomialAlgebra Integer	
V	UnivariatePolynomialAlgebra U	

Description

BivariateUtilitiesPackage (U,V) provides basic operations for bivariate polynomials over the integer numbers. In particular, it provides support for modular methods. In the above description, the smallest variable of a polynomial from V refers to the variable of U and its degree refers to the its degree as a univariate polynomial w.r.t. the variable of V.

Exports

maxNorm:	$U \rightarrow Z$	maximum absolute value of a coefficient
maxNorm:	$V \rightarrow Z$	maximum absolute value of a coefficient
degree_U:	$V \rightarrow Z$	degree w.r.t. the smallest variable
resultantCoefficientBound:	$(V, V) \rightarrow Z$	upper bound for the resultant maxNorm
resultantDegreeBound:	$(V, V) \rightarrow Z$	degree bound for the resultant
primeBad?:	$(V, V, I) \rightarrow B$	true iff the degree of one of the two polynomials drops modulo the number

where

```
I == MachineInteger
Z == Integer
B == Boolean
```

DirectProduct

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)

DirectProductCategory

Usage

DirectProductCategory (dim, T): Category

Parameter	Type	Description
<i>dim</i>	MachineInteger	The length of a direct product
<i>T</i>	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 (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

DistributedMultivariatePolynomial0

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 can be multiplied only by a constant from R . 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)

ExponentCategory

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 additive ordered monoid (with cancellation) by associating to every product of variables its sequence of degrees.

Exports

GeneralExponentCategory

exponent:	$V \rightarrow \%$	The exponent of a variable
exponent:	$(V, Z) \rightarrow \%$	The exponent of a power of a variable
exponent:	Generator Cross $(V, Z) \rightarrow \%$	The exponent of a power product
exponent:	$(\text{List } V, \text{List } Z) \rightarrow \%$	The exponent of a power product
terms:	$\% \rightarrow \text{Generator Cross } (V, Z)$	Inverse map of exponent
mainVariable:	$\% \rightarrow \text{Partial } V$	The biggest variable, if any
variables:	$\% \rightarrow \text{List } V$	The list of variables of a monomial
degree:	$(\%, V) \rightarrow Z$	The degree w.r.t. a variable
totalDegree:	$\% \rightarrow Z$	The sum of the degrees of an exponent
totalDegree:	$(\%, \text{List } V) \rightarrow Z$	The sum of the degrees w.r.t. a list
gcd:	$(\%, \%) \rightarrow \%$	Monomial gcd
lcm:	$(\%, \%) \rightarrow \%$	Monomial lcm
syzygy:	$(\%, \%) \rightarrow (\%, \%)$	Cofactors w.r.t. lcm

where

$Z == \text{Integer}$

Usage

FiniteAbelianMonoidRing0 (R,V,E): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	VariableType	The variable type
E	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 $c_1 x + \mathbf{term}(c_2, e)y$ 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

FiniteVariableType

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

<code>variable:</code>	<code>MachineInteger → %</code>	Retuns the n -th variable of the type, if any
<code>index:</code>	<code>% → MachineInteger</code>	Retuns the associated machine integer
<code>#:</code>	<code>MachineInteger</code>	The number of elements of the type
<code>max:</code>	<code>%</code>	The greatest element of the type
<code>min:</code>	<code>%</code>	The smallest element of the type
<code>minToMax:</code>	<code>List %</code>	The elements of the type sorted in increasing order
<code>maxToMin:</code>	<code>List %</code>	The elements of the type sorted in decreasing order

Usage

variable *i*

Signature

variable: MachineInteger \rightarrow %

Parameter	Type	Description
<i>i</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 $a \geq b$ implies $a + c \geq b + c$. 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 $a \geq b$ and *failed* otherwise. For multivariate monomials $f(a, b)$ is c if $a = bc$ and *failed* if 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
<i>n</i>	Integer	An integer
<i>x</i>	%	An element of the type

Returns

Returns the product *nx*.

IntegerExponentVectorCategory

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 non-negative integers.

Exports

CopyableType

ExponentCategory V

HashType

free!: % \rightarrow ()

Asserts that the input will no longer be used

exponent: Tuple Integer \rightarrow %

Creation from a tuple

exponent: PrimitiveArray Integer \rightarrow %

Creation from a primitive array

Degrees start at slot 1

Slot 0 must be the total degree

parray: % \rightarrow PrimitiveArray Integer

Inverse mapping of exponent

IntegerPolynomial

Usage

```
import from IntegerPolynomial
```

Description

`IntegerPolynomial` provides a basic domain for multivariate polynomials with `Integer` coefficients and variables from `OrderedSymbol`. The representation is sparse and recursive by means of the `SparseUnivariatePolynomial` univariate polynomial domain constructor.

Exports

```
RecursiveMultivariatePolynomialCategory0(Z,OS)
```

```
coerce:      String → %           Conversion.
```

```
univariate:  % → SUP(%)           Convert to univariate w.r.t. the greatest variable
```

```
multivariate: (SUP(%), OS) → %    Convert to multivariate with given variable
```

where

```
OS == OrderedSymbol
```

```
Z  == Integer
```

```
SUP == SparseUnivariatePolynomial
```

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 primitive 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 primitive 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 non-negative machine integers.

Exports

CopyableType

ExponentCategory V

HashType

free!: % \rightarrow () Asserts that the input will no longer be used

exponent: Tuple I \rightarrow % Creation from a tuple

exponent: PrimitiveArray I \rightarrow % Creation from a primitive array

Degrees start at slot 1

Slot 0 must be the total degree

parray: % \rightarrow PrimitiveArray I Inverse mapping of **exponent**

where

I == MachineInteger

MachineIntegerLexicographicalExponent

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 primitive 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` \rightarrow % Conversion to an element of the type.

OrderedVariableList

Usage

import from OrderedVariableList t

Parameter	Type	Description
<i>t</i>	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 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

OrderedVariableTuple

Usage

import from OrderedVariableTuple t

Parameter	Type	Description
<i>t</i>	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 ExpressionType	The coefficient domain
V	TotallyOrderedType ExpressionType	The variable domain

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)

eval: $(\%, V, R) \rightarrow \%$ eval(x, v, r) evaluates x at $v = r$

eval: $(\%, V, \%) \rightarrow \%$ eval(x, v, y) evaluates x at $v = y$

where

GEN == Generator

PZ == Cross($\%$, Integer)

if R has CommutativeRing then

differentiate: $(\%, V) \rightarrow \%$ Differentiation w.r.t. a variable

differentiate: $(\%, V, \text{Integer}) \rightarrow \%$ Iterated differentiation w.r.t. a variable

if R has CommutativeRing then

CommutativeRing

if R has IntegralDomain then

IntegralDomain

if R has GcdDomain then

GcdDomain

Usage

PolynomialRing0 (R,V): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	TotallyOrderedType ExpressionType	The variable domain

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 $\{v_1, \dots, v_l\}$ this polynomial ring is just $R[v_1, \dots, v_l]$. 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 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 set of algebraically independent 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

PolynomialTypeRing(R, V)
FreeAlgebra R

Usage

PolynomialTypeRing (R,V): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	ExpressionType	The variable domain

Description

PolynomialTypeRing (R,V) is the category of standard filtered rings generated over R by power products of the form $v_1^{e_1} \cdots v_n^{e_n}$ for $v_i \in V$ and $e_i \in \mathbb{N}$. No commutativity is assumed either between the variables in V, or between R and V.

Exports

```
StandardFilteredRing(R, V)
coefficient:      (% , V, Z) → %      Coefficient w.r.t. a power  $v^n$ .
coefficient:      (% , L V, L Z) → %   Coefficient w.r.t. a power product
degree:           (% , V) → Z          Degree w.r.t. a variable
degrees:          % → GEN VZ          Degrees w.r.t. all variables
leadingCoefficient: (% , V) → %        Leading coefficient w.r.t. a variable
multivariate:     (P:IFRR R) → (P, V) → % Conversion from univariate view
                  (P:IFRR P) → (P, V) → %
reductum:         (% , V) → %          Reductum w.r.t. a variable
univariate:       (P:IFRR P) → (% , V) → P Conversion to univariate view
```

If V has TotallyOrderedType then

```
initial:          % → %                Leading coefficient w.r.t. main variable
univariate:       (P:IFRR P) → % → P   Conversion to univariate view
```

where

```
GEN == Generator
IFRR == IndexedFreeRRing
L == List
Z == Integer
VZ == Cross (V, Integer)
```


Usage

import from RecursiveMultivariatePolynomial0 (UP,R,V)

Parameter	Type	Description
UP	$(T: \text{Join}(\text{ArithmeticType}, \text{ExpressionType})) \rightarrow \text{POL } T$	Univariate functor
R	Ring	The coefficient ring
V	VariableType	The variables

where

$\text{POL} == \text{UnivariatePolynomialAlgebra}$

Description

RecursiveMultivariatePolynomial0 (UP,R,V) provides a basic domain for multivariate polynomials with coefficients in R and variables in V . The representation is recursive by means of UP .

Exports

RecursiveMultivariatePolynomialCategory0(R,V)

univariate: $\% \rightarrow UP \%$ Convert to univariate w.r.t. the greatest variable

multivariate: $(UP(\%), V) \rightarrow \%$ Convert to multivariate with given variable

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 recursive vision of a multivariate polynomial (as a univariate polynomial w.r.t. its main variable).

Exports

`PolynomialRing(R,V)`

<code>variables:</code>	<code>% → SortedSetV</code>	The sorted set of variables of a polynomial
<code>mvar:</code>	<code>% → V</code>	Greatest variable of a polynomial
<code>mdeg:</code>	<code>% → Integer</code>	Degree w.r.t. greatest variable
<code>rank:</code>	<code>% → %</code>	Leading monomial as univariate w.r.t. greatest variable
<code>init:</code>	<code>% → %</code>	Leading coefficient as univariate w.r.t. greatest variable
<code>tail:</code>	<code>% → %</code>	Reductum as univariate w.r.t. greatest variable
<code>head:</code>	<code>% → %</code>	Leading term as univariate w.r.t. greatest variable

Usage

import from ResultantOfBivariatePolynomialsOverSmallPrimeField (Kp, Up, Vp)

Parameter	Type	Description
Kp	SmallPrimeFieldCategory	
Up	UnivariatePolynomialAlgebra Kp	
Vp	UnivariatePolynomialAlgebra Up	

Description

ResultantOfBivariatePolynomialsOverSmallPrimeField (Kp, Up, Vp) provides resultant computations for two bivariate polynomials over a small prime field.

Exports

evaluationResultant:	$(Vp, Vp, Z) \rightarrow Up$	resultant computed by an evaluation and interpolation scheme. Arg 3 is a degree bound.
evaluationReduction:	$(Vp, Kp) \rightarrow Up$	evaluation at a point
interpolation:	$(A\ Kp, A\ Kp) \rightarrow Up$	Lagrange interpolant where the arrays give points and values resp.
integerImage:	$Up \rightarrow DUP\ I$	conversion.

where

DUP	==	DenseUnivariatePolynomial
I	==	MachineInteger
Z	==	Integer
A	==	Array

SparseIntegerMultivariatePolynomial

Usage

import from SparseIntegerMultivariatePolynomial (V)

Parameter	Type	Description
V	VariableType	The variables

Description

SparseIntegerMultivariatePolynomial (V) provides a basic domain for multivariate polynomials with coefficients over the `Integer` ring and variables in V . The representation is sparse and recursive by means of *SUP*.

Exports

`RecursiveMultivariatePolynomialCategory0(Z,V)`

`univariate: % \rightarrow SUP(%) Convert to univariate w.r.t. the greatest variable`

`multivariate: (SUP(%), V) \rightarrow % Convert to multivariate with given variable`

where

`Z == Integer`

`SUP == SparseUnivariatePolynomial`

SparseMultivariatePolynomial

Usage

import from SparseMultivariatePolynomial (R,V)

Parameter	Type	Description
R	Ring	The coefficient ring
V	VariableType	The variables

Description

SparseMultivariatePolynomial (R,V) provides a basic domain for multivariate polynomials with coefficients in R and variables in V . The representation is sparse and recursive by means of *SUP*.

Exports

RecursiveMultivariatePolynomialCategory0(R,V)

univariate: % \rightarrow SUP(%) Convert to univariate w.r.t. the greatest variable

multivariate: (SUP(%), V) \rightarrow % Convert to multivariate with given variable

where

SUP == SparseUnivariatePolynomial

Usage

StandardFilteredRing (R,V): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	ExpressionType	The variable domain

Description

StandardFilteredRing (R,V) is the category of R-rings with a standard filtration for which R is the ring of elements of degree 0, and V generates the elements of degree 1. It is the set of finite linear combinations of the form $\sum r_i w_i$ where the r_i are in R and the w_i are words of V.

Exports

CopyableType

FreeRRing R

coerce:	$V \rightarrow \%$	Conversion of a variable to a polynomial
ground?:	$\% \rightarrow \text{Boolean}$	Membership test to the coefficient ring
totalDegree:	$\% \rightarrow \mathbb{Z}$	Greatest total degree among all the monomials
term:	$(R, V, \mathbb{Z}) \rightarrow \%$	term (r, v, n) returns $r v^n$ provided $n \geq 0$
term:	$(R, \text{GEN } VZ) \rightarrow \%$	Term from a power product and a coeff.
term:	$(R, \text{L } V, \text{L } \mathbb{Z}) \rightarrow \%$	Term from a power product and a coeff.
times:	$(\%, R, V, \mathbb{Z}) \rightarrow \%$	times (x, r, v, n) is $x (r v^n)$
times!:	$(\%, R, V, \mathbb{Z}) \rightarrow \%$	times (x, r, v, n) is $x (r v^n)$ and may modify x
univariate?:	$\% \rightarrow \text{Boolean}$	Check whether elt is univariate
variable:	$\% \rightarrow V$	Conversion of a polynomial to a variable
variable?:	$\% \rightarrow \text{Boolean}$	Membership test to the variable domain
variableProduct:	$\% \rightarrow \text{GEN } VZ$	variableProduct x is g s.t. term (1, g) is x
variables:	$\% \rightarrow \text{GEN } V$	The variables occuring in a polynomial

where

GEN == Generator

L == List

Z == Integer

VZ == Cross (V, Integer)

if R has Parsable and V has Parsable then

Parsable

if V has TotallyOrderedType then

mainVariable: $\% \rightarrow V$ Greatest variable occuring

VariableType

Usage

VariableType: Category

Description

VariableType is a category for multivariate polynomial variables looked as symbols with additional properties such as a total order.

Exports

TotallyOrderedType

ExpressionType

Parsable

SerializableType

HashType

variable: $\text{Symbol} \rightarrow \text{Partial } \%$ Associated variable, if any

symbol: $\% \rightarrow \text{Symbol}$ Associated symbol

ExpressionTree

Usage

import from ExpressionTree

Description

ExpressionTree is a type whose elements are expression trees.

Exports

OutputType

PrimitiveType

aldor:	(TEXT, %) → TEXT	Conversion to <i>A</i> [#] 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
infix:	(TEXT, %) → TEXT	Conversion to one-dim infix output
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 L ^A T _E X
texParen?:	(MachineInteger, %) → Boolean	Check whether to parenthesize

where

TEXT == TextWriter

OP == ExpressionTreeOperator

Usage*format*(p, t)**Signature**

aldor,axiom,C,fortran,infix,lisp,maple,tex: (TextWriter, %) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>t</i>	%	An expression tree

Description

Writes to *p* the expression corresponding to the tree *t* in the requested format.

Usage

apply(op, [t₁, ..., t_n])
op [t₁, ..., t_n]

Signature

apply: (ExpressionTreeOperator, List %) → %

Parameter	Type	Description
<i>op</i>	ExpressionTreeOperator	An operator
<i>t_i</i>	%	Expression trees

Returns

Returns the tree whose root is *op*, with arguments *t*₁, ..., *t*_{*n*}.

Usage

arguments t

Signature

arguments: % \rightarrow List %

Parameter	Type	Description
<i>t</i>	%	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 \rightarrow %

Parameter	Type	Description
<i>a</i>	ExpressionTreeLeaf	A leaf

Returns

extree a returns *a* as an expression tree.

Usage

is?(t, op)

Signature

is?: (% , ExpressionTreeOperator) → Boolean

Parameter	Type	Description
<i>t</i>	%	An expression tree
<i>op</i>	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
<i>t</i>	%	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: % \rightarrow %

Parameter	Type	Description
<i>t</i>	%	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?

Usagenegative? *t***Signature**negative?: % \rightarrow Boolean

Parameter	Type	Description
<i>t</i>	%	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: % \rightarrow ExpressionTreeOperator

Parameter	Type	Description
<i>t</i>	%	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
<i>prec</i>	MachineInteger	An operator precedence.
<i>t</i>	%	An expression tree

Returns

Returns *true* if *t* should be parenthetized when appearing as argument of an operator of precedence *prec*, *false* otherwise.

ExpressionTreeAnd

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**.

ExpressionTreeAssign

Usage

import from ExpressionTreeAssign

Description

ExpressionTreeAssign is 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

ExpressionTreeCase

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**.

ExpressionTreeComplex

Usage

import from ExpressionTreeComplex

Description

ExpressionTreeComplex is the complex operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeEqual

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**.

ExpressionTreeExpt

Usage

import from ExpressionTreeExpt

Description

ExpressionTreeExpt is the exponentiation operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeFactorial

Usage

import from ExpressionTreeFactorial

Description

ExpressionTreeFactorial is the generalized factorial operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeGreaterEqual

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**.

ExpressionTreeGreaterThan

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**.

ExpressionTreeIf

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
infix:	(TEXT, %) → TEXT	Conversion to one-dim infix output
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 L ^A T _E X
texParen?:	% → Boolean	Check whether to parenthesize

where

```
TEXT == TextWriter
```

Usage

format(p, a)

Signature

aldor,axiom,C,fortran,infix,lisp,maple,tex: (TextWriter, %) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	Boolean DoubleFloat Float Integer MachineInteger SingleFloat String Symbol	A constant

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: $\% \rightarrow \%$

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?: % \rightarrow Boolean

Parameter	Type	Description
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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: % \rightarrow **Symbol**
symbol?: % \rightarrow **Boolean**

Parameter	Type	Description
<i>a</i>	%	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?: % \rightarrow Boolean

Parameter	Type	Description
<i>a</i>	%	A leaf

Returns

Returns *true* if the leaf *a* should be parenthetized, *false* otherwise.

ExpressionTreeLessEqual

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**.

ExpressionTreeLessThan

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**.

ExpressionTreeLispList

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**.

ExpressionTreeList

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**.

ExpressionTreeMatrix

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**.

ExpressionTreeMinus

Usage

import from ExpressionTreeMinus

Description

ExpressionTreeMinus is the unary/binary minus operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeNotEqual

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**.

ExpressionTreeOperator

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
infix:	(TEXT, List TREE) → TEXT	Conversion to one-dim infix output
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 L ^A T _E X
texParen?:	MachineInteger → Boolean	Check whether to parenthesize
uniqueId:	MachineInteger	A unique key per operator

where

TEXT == TextWriter
TREE == ExpressionTree

Usage

format(*p*, [*t*₁, ..., *t*_{*n*}])

Signature

aldor,axiom,C,fortran,infix,lisp,maple,tex: (TextWriter, List ExpressionTree) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>t</i> _{<i>i</i>}	ExpressionTree	The arguments of the operator

Description

Writes to *p* the expression corresponding to this operator applied to the arguments (*t*₁, ..., *t*_{*n*}) in the requested format.

Usage

arity

Signaturearity: \rightarrow 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: \rightarrow Symbol

Returns

Returns the name of this operator.

Usage

texParen? prec

Signature

texParen?: MachineInteger \rightarrow Boolean

Parameter	Type	Description
<i>prec</i>	MachineInteger	An operator precedence.

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: \rightarrow MachineInteger

Returns

Returns a integer key which is associated to this operator only. This is used for testing whether two operators are equal.

ExpressionTreeOperatorTools

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, [t1, ..., tn], paren?, farg, op, left, right)`

Signatures

`infix: (TEXT, List TREE, TREE → Boolean, (TEXT, TREE) → TEXT, String, String, String) → TEXT`

Parameter	Type	Description
<i>p</i>	TEXT	The port to write to
<i>[t₁, ..., t_n]</i>	List TREE	The arguments of the operator
<i>paren?</i>	TREE → Boolean	The parenthetization function
<i>farg</i>	(TEXT, TREE) → TEXT	The function for the arguments
<i>op</i>	String	The infix symbol
<i>left</i>	String	The left parenthesis (optional)
<i>right</i>	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, . . . , *t*_{*n*}])

Signature

lisp: (TEXT, String, List TREE) → TEXT

Parameter	Type	Description
<i>p</i>	TEXT	The port to write to
<i>s</i>	String	A Lisp operator name
[<i>t</i> ₁ , . . . , <i>t</i> _{<i>n</i>}]	List TREE	The arguments of the operator

Description

Writes (*st*₁ . . . *t*_{*n*}) to *p*, where each *t*_{*i*} is written in Lisp format.

Usage

```
prefix(p, t, paren?, farg, op, left, right)
prefix(p, [t1, ..., tn], 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
<i>p</i>	TEXT	The port to write to
<i>[t₁, ..., t_n]</i>	List TREE	The arguments of the operator
<i>paren?</i>	TREE → Boolean	The parenthetization function
<i>farg</i>	(TEXT, TREE) → TEXT	The function for the arguments
<i>op</i>	String	The prefix symbol
<i>left</i>	String	The left parenthesis (optional)
<i>right</i>	String	The right parenthesis (optional)

where

```
TEXT == TextWriter
TREE == ExpressionTree
```

Description

Writes $op(farg(t_1), \dots, farg(t_n))$ 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`

ExpressionTreePlus

Usage

import from ExpressionTreePlus

Description

ExpressionTreePlus is the addition operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreePrefix

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.

ExpressionTreeQuotient

Usage

import from ExpressionTreeQuotient

Description

ExpressionTreeQuotient is the division operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeSubscript

Usage

import from ExpressionTreeSubscript

Description

ExpressionTreeSubscript is the subscript operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeTimes

Usage

```
import from ExpressionTreeTimes
```

Description

ExpressionTreeTimes is the multiplication operator for expression trees.

Exports

```
ExpressionTreeOperator
```

ExpressionTreeVector

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**.

Evaluator

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!:	$(\text{TEXT}, \text{TREE}, \text{SYM } R) \rightarrow \overline{R}$	Interpret an arbitrary tree
evalLeaf!:	$(\text{LEAF}, \text{SYM } R) \rightarrow \overline{R}$	Interpret a leaf
evalOp!:	$(\text{TEXT}, \text{TREE}, \text{SYM } R) \rightarrow \overline{R}$	Interpret $op(args)$
evalPrefix!:	$(\text{String}, Z, \text{List } R, \text{SYM } R) \rightarrow \overline{R}$	Interpret $op(args)$, op is prefix

where

Z	==	MachineInteger
LEAF	==	ExpressionTreeLeaf
\overline{R}	==	Partial R
SYM	==	SymbolTable
TEXT	==	TextWriter
TREE	==	ExpressionTree

InfixExpressionParser

Usage

import from InfixExpressionParser

Description

InfixExpressionParser implements infix expression parsers.

Exports

ParserReader

precedences!: ($\%$, MachineInteger, MachineInteger) \rightarrow $\%$ Set operator precedences

Usage

precedences!(p, op, n)

Signature

precedences!: (% , MachineInteger, MachineInteger) → %

Parameter	Type	Description
<i>p</i>	%	A parser
<i>op</i>	MachineInteger	An operator code
<i>n</i>	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`.

LispExpressionParser

Usage

import from LispExpressionParser

Description

LispExpressionParser implements lisp expression parsers.

Exports

ParserReader

MakePartialRing

Usage

import from MakePartialRing(R, f)

Parameter	Type	Description
R	Ring	A ring
f	$R \rightarrow \text{Partial Integer}$	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: $R \rightarrow \%$ Convert an element of R to one of the partial ring

coerce: $\% \rightarrow R$ 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: % \rightarrow 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: % \rightarrow 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 5th 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)))
```

Parsable

Usage

Parsable: Category

Description

Parsable is the category of types that convert expression trees into themselves whenever possible.

Exports

InputType

eval: ExpressionTree \rightarrow Partial % Interpret a tree

eval: ExpressionTreeLeaf \rightarrow Partial % Interpret a leaf

eval: (MachineInteger, List ExpressionTree) \rightarrow Partial % Interpret a node

Usage

```
eval e
eval t
eval(op,[e1,...,en])
```

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,[e₁,...,e_n]) returns the result of evaluating $op(e_1, \dots, e_n)$ 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

<code>eof?:</code>	<code>% → Boolean</code>	Check for end of input
<code>lastError:</code>	<code>% → MachineInteger</code>	Code for last parsing error
<code>parse!:</code>	<code>% → Partial ExpressionTree</code>	Parse one expression

Usage

eof? p

Signature

eof?: % \rightarrow Boolean

Parameter	Type	Description
<i>p</i>	%	A parser

Returns

Returns *true* if the input is finished, *false* otherwise.

Usage

lastError p

Signature

lastError: % \rightarrow MachineInteger

Parameter	Type	Description
<i>p</i>	%	A parser

Returns

Returns the code for the last parsing error.

Usage

parse! p

Signature

parse!: % \rightarrow 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 \rightarrow % Create a parser

Usage

parser r

Signature

parser: TextReader → %

Parameter	Type	Description
<i>r</i>	TextReader	The input stream to parse

Returns

Returns a parser that takes its input on r.

ParsingTools

Usage

import from ParsingTools R

Parameter	Type	Description
<i>R</i>	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, \dots, e_n])`

Signature

`evalArith: (MachineInteger, List ExpressionTree) \rightarrow Partial R`

Parameter	Type	Description
op	MachineInteger	Code for an operator
e_i	ExpressionTree	Expression trees

Returns

Returns the result of evaluating $op(e_1, \dots, e_n)$ where op is a code from `include/algebrauid.as`. Provides support for the evaluation of the operators $+$, $-$, $*$ and \wedge , as well as $/$ when R has `CommutativeRing` or `FloatType`.

Usage

evalInt e

Signature

evalInt: ExpressionTree \rightarrow 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.

PartialRing

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

Scanner

Usage

import from Scanner

Description

Scanner provides a simple scanner for mathematical expressions.

Exports

scan: TextReader \rightarrow Token Scan a token

Usage

scan *p*

Signature

scan: `TextReader` \rightarrow `Token`

Parameter	Type	Description
<i>p</i>	<code>TextReader</code>	Text to scan

Returns

Returns the next token read from the reader *p*.

import from Shell(P, R, E)

Shell(P, R, E) provides a basic read-eval-loop that converts its input to expressions of type R.

center:	$(TW, \text{String}) \rightarrow TW$	center a line of text
shell!:	$(P, TW, TW, \text{SymbolTable } R, \text{Boolean}, \text{Boolean}, \text{String}) \rightarrow Z$	Read-eval loop

```

TW == TextWriter
Z  == MachineInteger

```

Usage

center(*p*, *s*)

Signature

center: (TextWriter, String) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>s</i>	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) \rightarrow 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
<i>verbose?</i>	Boolean	Select verbose or quiet mode
<i>time?</i>	Boolean	Select whether to return times in msec
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 $n = b_1b_0$ where b_0 is 1 if a syntax error occurred, 0 otherwise, and b_1 is 1 if any other error occurred, 0 otherwise.

Token

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) \rightarrow %

Parameter	Type	Description
$[d_0, \dots, d_n]$	List Character	A list of digits
$[e_0, \dots, e_m]$	List Character	A list of digits

Returns

float($[d_0, \dots, d_n], [e_0, \dots, e_m]$) returns the float $d_n d_{n-1} \dots d_0 . e_m e_{m-1} \dots e_0$ as a token.

See also

integer

Usage

integer l

Signature

integer: List Character \rightarrow %

Parameter	Type	Description
$[d_0, \dots, d_n]$	List Character	A list of digits

Returns

integer($[d_0, \dots, d_n]$) returns the integer $d_0 + 10d_1 + \dots + 10^nd_n$ as a token.

See also

float

Usage

leaf t
leaf? t

Signatures

leaf: % → ExpressionTreeLeaf
leaf?: % → Boolean

Parameter	Type	Description
<i>t</i>	%	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

Token	name
-------	------

Usage
 name l

Signature
 name: List Character → %

Parameter	Type	Description
$[c_0, \dots, c_n]$	List Character	A list of characters

Returns
 name($[c_0, \dots, c_n]$) returns the name $c_n c_{n-1} \dots c_0$ as a token representing a constant symbol.

See also
 prefix, string

Usage

operator *t*
operator? *t*

Signatures

operator: % → ExpressionTreeOperator
operator?: % → Boolean

Parameter	Type	Description
<i>t</i>	%	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 \rightarrow %

Parameter	Type	Description
$[c_0, \dots, c_n]$	List Character	A list of characters

Returns

prefix($[c_0, \dots, c_n]$) returns the name “ $c_n c_{n-1} \dots c_0''$ ” as a token representing a prefix function.

See also

name

Usage

special t
special? t

Signatures

special: % → MachineInteger
special?: % → Boolean

Parameter	Type	Description
<i>t</i>	%	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

string l

Signature

string: List Character \rightarrow %

Parameter	Type	Description
$[c_0, \dots, c_n]$	List Character	A list of characters

Returns

string($[c_0, \dots, c_n]$) returns the string “ $c_n c_{n-1} \dots c_0''$ ” as a token representing a constant.

See also

name, prefix

Usage

token a

Signatures

token: `Character` \rightarrow `Partial %`

token: `ExpressionTreeOperator` \rightarrow `Partial %`

token: `MachineInteger` \rightarrow `Partial %`

Parameter	Type	Description
<i>a</i>	<code>Character</code>	A single character token
	<code>ExpressionTreeOperator</code>	An operator
	<code>MachineInteger</code>	A code of a special token

Returns

Returns the token corresponding to the single character *a*, *failed* if there is none.

AlgebraLibraryInformation

Usage

import from AlgebraLibraryInformation

Description

AlgebraLibraryInformation provides version information about the **Algebra** library.

Exports

VersionInformationType

IndexedVariable

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:    %  $\rightarrow$  S  Index of a variable
variable: S  $\rightarrow$  %  Create an indexed variable
```

Usage

index *v*

Signature

index: % \rightarrow S

Parameter	Type	Description
<i>v</i>	%	An indexed variable

Returns

Returns the index of *v*.

Usage
variable s

Signature
variable: S → %

Parameter	Type	Description
<i>s</i>	S	An index

Returns
Returns the variable x_s .

Permutation

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:	$(\%, Z) \rightarrow Z$	Image of an element
mapping:	$\% \rightarrow (Z \rightarrow Z)$	Action of a permutation
sign:	$\% \rightarrow Z$	Sign
transpose:	$(Z, Z) \rightarrow \%$	Transposition
transpose!:	$(\%, Z, Z) \rightarrow \%$	Compose with a transposition

where

$Z == \text{MachineInteger}$

Usage

apply(σ , x)
 σx

Signature

apply: ($\%$, MachineInteger) \rightarrow MachineInteger

Parameter	Type	Description
σ	$\%$	A permutation
x	MachineInteger	An index

Returns

Returns σx .

Usage

mapping σ

Signature

mapping: $\% \rightarrow (\text{MachineInteger} \rightarrow \text{MachineInteger})$

Parameter	Type	Description
σ	$\%$	A permutation

Returns

Returns the map corresponding to σ .

See also

apply

Usage

sign σ

Signature

sign: $\% \rightarrow \text{MachineInteger}$

Parameter	Type	Description
σ	$\%$	A permutation

Returns

Returns the sign of σ , *i.e.* $(-1)^\epsilon$ where ϵ is the number of transpositions in the factorization of σ .

Usage

```
transpose(x,y)
transpose!( $\sigma$ ,x,y)
```

Signatures

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

Parameter	Type	Description
σ	%	A permutation
x,y	MachineInteger	Indices

Returns

transpose(x,y) returns the transposition (xy) , while transpose!(σ ,x,y) returns the composition $(xy) \circ \sigma$.

Remarks

When using transpose!, the storage used by σ is allowed to be destroyed or reused, so do not use it unless σ 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. Since there is no guarantee of reuse, you should always use the permutation returned by transpose! rather than σ after the call.

Sequence

Usage

import from Sequence R

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

Sequence R implements infinite sequences with coefficients in R.

Exports

LinearStructureType R

UnivariateFreeLinearArithmeticType R

#:	$\% \rightarrow \mathbb{Z}$	number of computed elements
bound:	$\% \rightarrow \text{MachineInteger}$	upper bound on the support size
dot:	$\text{Vector } R \rightarrow \text{Vector } \% \rightarrow \%$	linear combination
finite?:	$\% \rightarrow \text{Boolean}$	check whether the support is finite
interlacing:	$\text{List } \% \rightarrow \%$	interlacing
sequence:	$\text{Stream } R \rightarrow \%$	create a sequence

if R has Ring then

random: $() \rightarrow \%$ random sequence

Usage

s

Signatures

#: % → MachineInteger

Parameter	Type	Description
<i>s</i>	%	a sequence

Returns

Returns the number of elements of *s* that have been computed.

Usage

bound s

finite? s

Signaturesbound: % \rightarrow MachineIntegerfinite?: % \rightarrow 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

```
dot [r1, ..., rn]
dot([r1, ..., rn])([s1, ..., sn])
```

Signature

```
dot: Vector R → Vector % → %
```

Parameter	Type	Description
r_i	R	coefficients
s_i	%	sequences

Returns

dot(r)(s) returns the sequence $\sum_{i=1}^n r_i s_i$, while dot(r) returns the map $[s_1, \dots, s_n] \rightarrow \text{sum}_{i=1}^n r_i s_i$.

Remarks

Using dot is more efficient than building up the same linear combination via additions and multiplications, since the intermediate combinations are not storing their computed elements when using dot.

Usage

interlacing $[s_1, \dots, s_n]$

Signature

interlacing: List % \rightarrow %

Parameter	Type	Description
s_i	%	sequences

Description

Given $s_i = r_{i1}, r_{i2}, \dots$, returns the sequence $r_{11}, r_{21}, \dots, r_{n1}, r_{12}, r_{22}, \dots, r_{n2}, \dots$

Usage

random()

Signature

random: $() \rightarrow \%$

Returns

Returns a sequence with random entries.

Usage

sequence s

Signature

sequence: Stream R \rightarrow %

Parameter	Type	Description
s	Stream R	a stream

Returns

Returns s viewed as a sequence.

4 libaldor Reference Manual

AbelianGroup

Usage

AbelianGroup: Category

Description

AbelianGroup is the category of commutative groups.

Exports

AdditiveType

AbelianMonoid

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

$+: (\%,\%) \rightarrow \%$

Parameter	Type	Description
x,y	$\%$	elements of the monoid

Returns

Returns the sum $x + y$.

Usage $n * x$ **Signature** $*: (\text{Integer}, \%) \rightarrow \%$

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

ReturnsReturns the product nx .

Usage

add!(x, y)

Signature

add!: ($\%$, $\%$) \rightarrow $\%$

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?: 'a → Boolean

Parameter	Type	Description
<i>x</i>	'a	an element of the monoid

Returns

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

Algebra

Usage

Algebra R:Category

Parameter	Type	Description
R	Ring	The coefficient ring

Description

Algebra R is the category of algebras over R , *i.e.* R -rings such that $R \cdot 1$ is included in the center (in other word, multiplication by R is bilinear).

Exports

`RRing R`

CharacteristicZero

Usage

CharacteristicZero: Category

Description

CharacteristicZero is the category of rings of characteristic 0.

Exports

Ring

CommutativeRing

Usage

CommutativeRing: Category

Description

CommutativeRing is the category of commutative rings.

Exports

Ring

canonicalUnitNormal?:	$\rightarrow \text{Boolean}$	Check if <code>unitNormal</code> is canonical
cutoff:	$Z \rightarrow Z$	Cutoffs for various fast algorithms
reciprocal:	$\% \rightarrow \text{Partial } \%$	Inverse
unitNormal:	$\% \rightarrow (\%, \%, \%)$ $(\%, \%) \rightarrow (\%, \%)$	Representative of the associates
unit?:	$\% \rightarrow \text{Boolean}$	Test whether an element is a unit

where

$Z == \text{MachineInteger}$

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 (y, u, u^{-1}) and `unitNormal(x')` returns (y', u', u'^{-1}) , then $y = y'$.

See also

`unitNormal`, `unit?`

Usage

cutoff *t*

Signature

cutoff: `MachineInteger` \rightarrow `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?: % \rightarrow Boolean

Parameter	Type	Description
x	%	An element of the ring

Returns

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

See also

canonicalUnitNormal?, unitNormal

DecomposableRing

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?: % \rightarrow Boolean

someFactors: % \rightarrow List % Get some factors

Usage

provablyIrreducible? x

Signature

provablyIrreducible?: % \rightarrow 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: $\% \rightarrow \text{List } \%$

Parameter	Type	Description
x	$\%$	A ring element

Returns

Returns $[x_1, \dots, x_n]$ 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.

DifferentialExtension

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 \rightarrow 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.

DifferentialRing

Usage

DifferentialRing: Category

Description

DifferentialRing is the category of commutative differential rings.

Exports

CommutativeRing

derivation: \rightarrow Derivation % The derivation

differentiate: $(\%, \text{Integer}) \rightarrow \%$ 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: $(\%, \text{Integer}) \rightarrow \%$

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)

EuclideanDomain

Usage

EuclideanDomain: Category

Description

EuclideanDomain is the category of commutative Euclidean domains.

Exports

GcdDomain

diophantine:	$(\%, \%, \%) \rightarrow \partial \%$	Linear diophantine solver
divide:	$(\%, \%) \rightarrow (\%, \%)$	Euclidean division
divide!:	$(\%, \%, \%) \rightarrow (\%, \%)$	In-place Euclidean division
euclid:	$(\%, \%) \rightarrow \%$	Euclidean gcd
euclid!:	$(\%, \%) \rightarrow \%$	In-place Euclidean gcd
euclideanSize:	$\% \rightarrow \text{Integer}$	Size function of the domain
extendedEuclidean:	$(\%, \%) \rightarrow (\%, \%, \%)$	Extended Euclidean Algorithm
	$(\%, \%, \%) \rightarrow \partial \text{Cross}(\%, \%)$	
quo:	$(\%, \%) \rightarrow \%$	Quotient
rem:	$(\%, \%) \rightarrow \%$	Remainder
remainder!:	$(\%, \%) \rightarrow \%$	In-place remainder
rationalReconstruction:	$(\%, \%, \text{Z}, \text{Z}) \rightarrow \partial \text{Cross}(\%, \%)$	Rational reconstruction

where

∂ == Partial
Z == Integer

Usage

diophantine(a, b, m)

Signature

diophantine: (% , % , %) \rightarrow 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

Signaturesdivide: ($\%$, $\%$) \rightarrow ($\%$, $\%$)quo,rem: ($\%$, $\%$) \rightarrow $\%$

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

Returns

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

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: (% , %) \rightarrow %

Parameter	Type	Description
x, y	%	Elements of the ring

Returns

Returns $\text{gcd}(x, y)$ 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: $\% \rightarrow \text{Integer}$

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: $(\%, \%) \rightarrow (\%, \%, \%)$
 extendedEuclidean: $(\%, \%, \%) \rightarrow \mathbf{Partial}(\%, \%)$

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

rationalReconstruction(u, m, n, d)

Signature

rationalReconstruction: (% , % , Integer, Integer) → Partial Cross(% , %)

Parameter	Type	Description
u	%	An element of the ring
m	%	A nonzero modulus
n, d	Integer	Bounds for the size of the result

Returns

Returns either (a, b) such that $a/b = u \pmod{m}$, $|a| \leq n$ and $|b| \leq d$, or *failed* if no such a, b exist.

Remarks

The resulting a and b might not be unique, depending on the values of the bounds n and d .

Usage

remainder!(x, y)

Signature

remainder!: (% , %) → %

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

FiniteCharacteristic

Usage

FiniteCharacteristic: Category

Description

FiniteCharacteristic is the category of finite characteristic rings.

Exports

Ring

pthPower: % → % Exponentiation to the characteristic

pthPower!: % → % In-place exponentiation to the characteristic

Usage

pthPower x
 pthPower! x

Signature

pthPower: % \rightarrow %

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`.

FiniteSet

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
<i>p</i>	<i>%</i>	An element
<i>x</i>	ExpressionTree	A name for the variables

Returns

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

Usage

index *p*

Signature

index: $\% \rightarrow \text{Integer}$

Parameter	Type	Description
<i>p</i>	$\%$	An element

Returns

Returns the index of *p*.

See also

lookup(FiniteSet)

Usage

lookup *j*

Signature

lookup: Integer \rightarrow %

Parameter	Type	Description
<i>j</i>	Integer	An index

Returns

Returns the element with index *j*.

See also

index(FiniteSet)

Usage

random()

Signature

random: $() \rightarrow \%$

Returns

Returns a random element.

FreeAlgebra

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

FreeRRing R

If R has Ring then
Algebra R

FreeLinearArithmeticType

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 Algebra R even when R is a Ring.

Exports

FreeLinearCombinationType R
LinearArithmeticType R

FreeLinearCombinationType

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 **Module** R even when R is a **Ring**.

Exports

ExpressionType

LinearCombinationType R

map: $(R \rightarrow R) \rightarrow \% \rightarrow \%$ Lift a mapping

map!: $(R \rightarrow R) \rightarrow \% \rightarrow \%$ Lift a mapping

Usage

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

Signature

```
map: (R → R) → % → %
```

Parameter	Type	Description
f	$R \rightarrow R$	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 $m \rightarrow f(m)$. In both cases, map! does not make a copy of m but modifies it in place.

FreeModule

Usage

FreeModule R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

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		
leadingCoefficient:	$\% \rightarrow R$	The leading coefficient
nonZeroCoefficients:	$\% \rightarrow \text{Generator } R$	Iterate over the coefficients
reductum:	$\% \rightarrow \%$	All terms except the leading one
reductum!:	$\% \rightarrow \%$	All terms except the leading one
support:	$\% \rightarrow \text{Generator Cross}(R, \%)$	Make an iterator
term?:	$\% \rightarrow \text{Boolean}$	Test for a monomial
trailingCoefficient:	$\% \rightarrow R$	The trailing coefficient

If R has Ring then
Module R

If R has GcdDomain then		
content:	$\% \rightarrow R$	Content
primitive:	$\% \rightarrow (R, \%)$	Content and primitive part
primitive!:	$\% \rightarrow (R, \%)$	Content and primitive part
primitivePart:	$\% \rightarrow \%$	Primitive part
primitivePart!:	$\% \rightarrow \%$	Primitive part

If R has Field then		
monic:	$\% \rightarrow \%$	Make monic
monic!:	$\% \rightarrow \%$	Make monic

Usage

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

Signatures

content,content!: $\% \rightarrow R$
 primitive,primitive!: $\% \rightarrow (R, \%)$
 primitivePart,primitivePart!: $\% \rightarrow \%$

Parameter	Type	Description
<i>m</i>	$\%$	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

term? m

Signatureterm?: % \rightarrow Boolean

Parameter	Type	Description
m	%	An element of the module

ReturnsReturns *true* if $m = re$ for $r \in R$ and e an element of the basis, *false* otherwise.**Remarks**term?(0) returns *true*.

Usage

leadingCoefficient m
trailingCoefficient p

Signature

leadingCoefficient, trailingCoefficient: $\% \rightarrow R$

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: % \rightarrow %

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: % \rightarrow 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: % \rightarrow %

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: % \rightarrow 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 .

FreeRRing

Usage

FreeRRing R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

FreeRRing R is a category for free R-rings 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

ground?: % \rightarrow Boolean Test for a ground element

If R has CharacteristicZero then

CharacteristicZero

If R has FiniteCharacteristic then

FiniteCharacteristic

If R has Ring then

RRing R

if R has RittRing then

RittRing

Usage

ground? m

Signature

ground?: % \rightarrow Boolean

Parameter	Type	Description
m	%	An element of the module

Returns

Returns *true* if $m = r \cdot 1$ for $r \in R$, *false* otherwise.

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 [ $x_1, \dots, x_n$ ]

```

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($[x_1, \dots, x_n]$) 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, \dots, x_n]

Signatures

gcdquo: $(\%, \%) \rightarrow (\%, \%, \%)$
gcdquo: **List** $\% \rightarrow (\%, \text{List } \%)$

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($[x_1, \dots, x_n]$) returns $(g, [y_1, \dots, y_n])$ where $g = \gcd(x_1, \dots, x_n)$ 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

`/:` $(\%, \%) \rightarrow \%$ quotient

`inv:` $\% \rightarrow \%$ inverse

Usage
 x/y

Signature
 $/: \% \rightarrow \%$

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

Returns
Returns xy^{-1} .

Usage

inv x

Signature

inv: % → %

Parameter	Type	Description
<i>x</i>	%	An element of the group

Returns

Returns x^{-1} .

IndexedFreeAlgebra

Usage

IndexedFreeAlgebra(R,E): 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

IndexedFreeRRing(R , E)

IndexedFreeLinearArithmeticType

Usage

`IndexedFreeLinearArithmeticType(R, E): Category`

Parameter	Type	Description
R	<code>ExpressionType</code> <code>ArithmeticType</code>	The coefficient domain
E	<code>ExpressionType</code>	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, E)`

`FreeLinearArithmeticType R`

`add!:` `(%, R, E, %) → %` Add a shifted element

Usage

add!(p, c, e, q)

Signature

add!: ($\%$, R, E, $\%$) \rightarrow $\%$

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.

IndexedFreeLinearCombinationType

Usage

IndexedFreeLinearCombinationType(R , E): Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain
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: (% , E) → 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!: $(\%, R, E) \rightarrow \%$

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
<i>p</i>	%	An element of the module
<i>e</i>	E	An exponent

Returns

Returns the coefficient of *e* in *p*.

Usagemonomial(e)term(r , e)**Signatures**monomial: $E \rightarrow \%$ term: $(R, E) \rightarrow \%$

Parameter	Type	Description
r	R	A scalar
e	E	An exponent

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

Usage

IndexedFreeModule(R,E): Category

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

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:	% $\rightarrow E$	degree
generator:	% \rightarrow Generator Cross(R , E)	iterate through all the terms
leadingMonomial:	% \rightarrow %	leading monomial
leadingTerm:	% $\rightarrow (R, E)$	leading term
terms:	% \rightarrow Generator Cross(R , E)	iterate through all the terms
trailingDegree:	% $\rightarrow E$	trailing degree
trailingMonomial:	% \rightarrow %	trailing monomial
trailingTerm:	% $\rightarrow (R, E)$	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: % \rightarrow 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 $a_n \neq 0 \neq a_m$. When $p = 0$, both functions return an arbitrary value in E .

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

IndexedFreeRRing(R,E): Category

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

Description

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

Exports

FreeRRing R
IndexedFreeModule(R , E)
IndexedFreeLinearArithmeticType(R , E)

IntegerCategory

Usage

IntegerCategory: Category

Description

IntegerCategory is the category of integer-like rings.

Exports

CharacteristicZero

EuclideanDomain

IntegerType

Parsable

Specializable

integer: $\% \rightarrow \text{Integer}$ Conversion to an integer

rationalReconstruction: $\% \rightarrow \% \rightarrow \text{Partial Cross}(\%, \%)$ Rational reconstruction

Usage

binomial(n, m)

Signature

binomial: (%,%) \rightarrow %

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: % \rightarrow Integer

Parameter	Type	Description
<i>n</i>	%	An integer

Returns

Returns n as an integer.

Usage

rationalReconstruction m
 rationalReconstruction(m)(u)

Signature

rationalReconstruction: $\% \rightarrow \% \rightarrow \text{Partial Cross}(\%, \%)$

Parameter	Type	Description
m	$\%$	A nonzero modulus
u	$\%$	An Integer

Returns

rationalReconstruction(m)(u) returns either (a, b) such that $a/b = u \pmod{m}$, $|a| \leq \sqrt{(m-1)/2}$ and $|b| \leq \sqrt{(m-1)/2}$, or *failed* if no such a, b exist.

Remarks

The resulting a and b are guaranteed to be unique.

See also

rationalReconstruction

IntegralDomain

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
<i>a1</i> , <i>a2</i> , <i>b1</i> , <i>b2</i> , <i>q</i>	%	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
<i>a1, a2, b1, b2, c1, c2, q</i>	%	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: (`%`, `%`) \rightarrow `Partial %`

Parameter	Type	Description
x	<code>%</code>	The numerator
y	<code>%</code>	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:  % → % → 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 $a^{n+1} \nmid b$, while `order(a)` returns the map $b \rightarrow \nu_a(b)$.

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 $l := [p_1, \dots, p_n]$:

```
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 $a \nmid c$, while orderquo(a) returns the map $b \rightarrow \text{orderquo(a)}(b)$.

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 $x = qy$ if such a q exists, while `quotientBy(y)` returns the map $x \rightarrow \text{quotient}(x, y)$ and `quotientBy!(y)` returns the map $x \rightarrow \text{quotient!}(x, y)$. 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
```

LinearArithmeticType

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
 \wedge : (`%`, `Integer`) \rightarrow `%` exponentiation
`coerce`: `R` \rightarrow `%` Natural embedding

Usage

$x \wedge n$

Signatures

$\wedge: (\%, \text{Integer}) \rightarrow \%$

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

Returns

Returns x to the power n .

Usage

coerce r

Signature

coerce: R → %

Parameter	Type	Description
<i>r</i>	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
^:	(%, Integer) → %	exponentiation
one?:	% → Boolean	test for 1
times!:	(%, %) → %	In-place product

Usage

1

Signature

1: %

Returns

Returns the constant 1.

Usage
 $x * y$

Signature
 $*: (\%,\%) \rightarrow \%$

Parameter	Type	Description
x, y	$\%$	elements of the monoid

Returns
Returns the product xy .

Usage

$x \wedge n$

Signatures

$\wedge: (\%, \text{Integer}) \rightarrow \%$

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
<i>x</i>	%	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.

NonCommutativeIntegralDomain

Usage

NonCommutativeIntegralDomain: Category

Description

NonCommutativeIntegralDomain is the category of non-commutative integral domains.

Exports

Ring

leftExactQuotient: $(\%, \%) \rightarrow \text{Partial } \%$ Left exact quotient

rightExactQuotient: $(\%, \%) \rightarrow \text{Partial } \%$ Right exact quotient

Usage

leftExactQuotient(x, y)

Signature

leftExactQuotient: (`%`, `%`) \rightarrow `Partial %`

Parameter	Type	Description
x	<code>%</code>	The numerator
y	<code>%</code>	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: ($\%$, $\%$) \rightarrow **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)

Ring

Usage

Ring: Category

Description

Ring is the category of rings, not necessarily commutative.

Exports

AbelianGroup

ArithmeticType

Monoid

characteristic: Integer

characteristic

coerce: Integer \rightarrow %

embedding of the integers

factorial: (% , % , MachineInteger) \rightarrow %

Generalized factorial

random: () \rightarrow %

Get a random element

Usage

characteristic

Signature

characteristic: `Integer`

Returns

Returns the characteristic of the ring.

Usage

coerce n
n::%

Signature

coerce: Integer \rightarrow %

Parameter	Type	Description
<i>n</i>	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: $() \rightarrow \%$

Returns

Returns a random element.

RittRing

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

$/:$ $(\%, \text{Integer}) \rightarrow \%$ Division by a nonzero integer

inv: $\text{Integer} \rightarrow \%$ Inversion of a nonzero integer

Usage

x / n

Signature

$/: (\%, \text{Integer}) \rightarrow \%$

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

Signatureinv: Integer \rightarrow %

Parameter	Type	Description
n	Integer	A nonzero integer

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

RRing

Usage

RRing R:Category

Parameter	Type	Description
R	Ring	The coefficient ring

Description

RRing R is the category of R-rings, *i.e.* rings that are also R-modules.

Exports

LinearArithmeticType R

Module R

Ring

Specializable

Usage

Specializable: Category

Description

Specializable is the category of specializable types.

Exports

specialization: (Image:CommutativeRing) \rightarrow PartialFunction(% , Image) Morphism

Usage

specialization R

Signature

specialization: (Image:CommutativeRing) \rightarrow PartialFunction(% ,Image)

Parameter	Type	Description
<i>R</i>	CommutativeRing	A ring

Returns

Returns a partial map from % into R.

ExpressionType

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: % \rightarrow ExpressionTree

Parameter	Type	Description
<i>x</i>	%	The element to convert

Description

Converts *x* to an expression tree.

Usage

relativeSize x

Signature

relativeSize: % \rightarrow 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 automorphisms on R .

Exports

Group

apply: $(\%, R, \text{Integer}) \rightarrow R$ Apply a morphism to an element of R

function: $\% \rightarrow (R \rightarrow R)$ Action of a morphism

morphism: $(R \rightarrow R) \rightarrow \%$ Create a morphism

$(R \rightarrow R, R \rightarrow R) \rightarrow \%$

$((R, \text{Integer}) \rightarrow R) \rightarrow \%$

Usage

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

Signature

`apply: (% , R, n:Integer == 1) → R`

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

Returns

Returns $\sigma^n x$.

Usage

function σ

Signature

function: $\% \rightarrow (R \rightarrow R)$

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: $(R \rightarrow R) \rightarrow \%$
 morphism: $(R \rightarrow R, R \rightarrow R) \rightarrow \%$
 morphism: $((R, \text{Integer}) \rightarrow R) \rightarrow \%$

Parameter	Type	Description
f	$R \rightarrow R$	A function
f^{-1}	$R \rightarrow R$	The inverse function of f
g	$(R, \text{Integer}) \rightarrow R$	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

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

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 $g: (R, \text{Integer}) \rightarrow R$ should be used to avoid repeated iterations of σ , which is the default behavior.

Usage

CanonicalSimplification: Category

Description

CanonicalSimplification is the category of domains supporting a canonical simplifier. This means that for any p such that p equals its canonical associate and for any a , the element $a \bmod p$ is a canonical representative of the residue class of a by p . That is, for any b the relation $a \bmod p = b \bmod p$ is equivalent to p divides $b - a$. In addition $a \bmod p$ equals its canonical associate. If the domain is Euclidean, then it must support also a symmetric canonical simplifier. See the paper *On the genericity of the Modular Gcd Algorithm* by Kaltofen and Monagan in the proc. of ISSAC 1999.

Exports

CommutativeRing

mod:	(%, %) → %	residue class representative
mod_+:	(%, %) → %	modular sum
mod_-:	(%, %) → %	modular difference
mod_*:	(%, %) → %	modular product
recipMod	(%, %) → Partial %	modular reciprocal

if % has EuclideanDomain then

symmetricMod: (%, %) → % symmetric residue class representative

ChineseRemaindering

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: $(R,R) \rightarrow (R,R) \rightarrow R$ combine interpolated result with new modulus.
interpolate: $(\text{List } R, \text{List } R) \rightarrow R$ interpolate given residues and moduli.

if R has IntegerType then

combine: $(R, \text{MachineInteger}) \rightarrow (R, \text{MachineInteger}) \rightarrow R$ 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	A new modulus.
	MachineInteger	
A	R	The interpolated value modulo M .
a	R	The residue modulo m .
	MachineInteger	

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) \rightarrow 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 ExpressionType	Type to be extended

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

Derivation

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:	$(\%, R, \text{Integer}) \rightarrow R$ Differentiate an element of R
derivation:	$(R \rightarrow R) \rightarrow \%$ Create a derivation
function:	$\% \rightarrow (R \rightarrow R)$ 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: $(R \rightarrow R) \rightarrow \%$

Parameter	Type	Description
f	$R \rightarrow R$	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: % \rightarrow (R \rightarrow R)

Parameter	Type	Description
<i>D</i>	%	A derivation

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

Fraction

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

FractionalRoot

Usage

```
import from FractionalRoot
```

Description

FractionalRoot(R) provides fractions of R with multiplicities.

Parameter	Type	Description
R	CommutativeRing	A ring

Exports

ExpressionType		
fractionalRoot:	$(R, R, \text{Integer}) \rightarrow \%$	Create a root
integral?:	$\% \rightarrow \text{Boolean}$	Test whether root is integral
integralRoot:	$(R, \text{Integer}) \rightarrow \%$	Create a root
integralValue:	$\% \rightarrow R$	Value of an integral root
multiplicity:	$\% \rightarrow \text{Integer}$	Multiplicity of a root
setMultiplicity!:	$(\%, \text{Integer}) \rightarrow \%$	Change a multiplicity
value:	$\% \rightarrow (R, R)$	Value of a root

Signature

integral?: % \rightarrow Boolean

Usage

integral? r

Parameter	Type	Description
<i>r</i>	%	A root

Returns

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

UsagefractionalRoot(*a*, *b*, *n*)integralRoot(*a*, *n*)**Signatures**

fractionalRoot: (R, R, Integer) → %

integralRoot: (R, Integer) → %

Parameter	Type	Description
<i>a</i>	R	A numerator
<i>b</i>	R	A denominator
<i>n</i>	Integer	A multiplicity

ReturnsReturn the root *a* or *a/b* with multiplicity *n*.

Usage

integralValue r

Signature

integralValue: % \rightarrow Integer

Parameter	Type	Description
<i>r</i>	%	A root

Returns

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

See also

value

Usage

multiplicity *r*

Signature

multiplicity: $\% \rightarrow \text{Integer}$

Parameter	Type	Description
<i>r</i>	$\%$	A root

Returns

Return the multiplicity of *r*.

Usage

setMultiplicity!(*r*, *m*)

Signature

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

Parameter	Type	Description
<i>r</i>	<i>%</i>	A root
<i>m</i>	Integer	Its new multiplicity

Description

Sets the multiplicity of *r* to *m* and returns *r*.

Usage

(n, d) := value r

Signature

value: % \rightarrow (Integer, Integer)

Parameter	Type	Description
<i>r</i>	%	A root

Returns

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

See also

integralValue

FractionBy

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

FractionByCategory

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

FractionByCategory0

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: % \rightarrow Integer Valuation at p
shift: (% , Integer) \rightarrow % Multiplication by a power of p

Usage

order x

Signature

order: % \rightarrow Integer

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

Returns

Returns n such that $x = ap^n$ and $a \in R, p \nmid a$.

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 $x p^n$.

FractionCategory

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

LinearAlgebraRing

denominator: $\% \rightarrow R$ Denominator of a fraction

normalize: $\% \rightarrow \%$ Normalize a fraction

numerator: $\% \rightarrow R$ 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: % \rightarrow R

Parameter	Type	Description
x	%	A fraction

Returns

Returns respectively the denominator and the numerator of a fraction.

Usage

normalize x

Signature

normalize: % \rightarrow %

Parameter	Type	Description
x	%	A fraction

Description

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

FractionFieldCategory

Usage

FractionFieldCategory R: Category

Parameter	Type	Description
<i>R</i>	IntegralDomain	an integral domain

Description

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

Exports

FractionFieldCategory0 R

FractionFieldCategory0

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

$/: (R,R) \rightarrow \%$

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

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

if Q has FractionByCategory0 R

makeIntegralBy: $LQ \rightarrow (Integer, LR)$ Conversion to an integral element

if Q has FractionFieldCategory0 R

normalize: $LR \rightarrow (R, LQ)$ 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
<i>B</i>	LQ	A rational element

Returns

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

Usage

$(\mu, A) := \text{makeIntegralBy } B$

Signature

$\text{makeIntegralBy}: \text{LQ} \rightarrow (\text{Integer}, \text{LR})$

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: $LR \rightarrow LQ$

Parameter	Type	Description
A	LR	An integral element

Returns

Returns A as a rational element.

Usage

$(a, B) := \text{normalize } A$

Signature

normalize: $LR \rightarrow (R, LQ)$

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 is 1, for vectors that the first coordinate is 1, etc.

Usage

ModularComputation: Category

Description

ModularComputation is the category of domains that support modular algorithm such as the modular gcd algorithm. More precisely, for every element p of $\%$ the operation `residueClassRing` returns a domain implementing the residue class ring R/p . In addition, if $\%$ is an Euclidean domain, the operation `combine` implements the Chinese Remaindering algorithm.

Exports

`CanonicalSimplification`

`residueClassRing: (p: %) → ResidueClassRing (%p)` Residue class ring

if $\%$ has `EuclideanDomain` then

`combine: (% , %) → (% , %) → %` Chinese remaindering algorithm

Product

Usage

import from Product R

Parameter	Type	Description
R	<code>CommutativeRing</code>	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`

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

Usage
p

Signatures
#: % → MachineInteger

Parameter	Type	Description
<i>p</i>	%	A product

Returns
Returns the number of terms in the product p.

Usage

for d in divisors p repeat { ... }

Signature

divisors: % \rightarrow 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: % \rightarrow R

Parameter	Type	Description
<i>p</i>	%	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: $\% \rightarrow (\mathbf{R}, \mathbf{R})$

Parameter	Type	Description
<i>p</i>	$\%$	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	$R \rightarrow M$	a logarithmic function on R
p	%	A product

Description

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

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) \rightarrow %

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.

ResidueClassRing

Usage

ResidueClassRing (R,p): Category

Parameter	Type	Description
R	CommutativeRing	The ring
p	R	The modulo

Description

ResidueClassRing (R,p) specifies an implementation of the residue class ring R/p . The canonical pre-image of an element x of % is the element r of R such that x is a canonical representative of r and r equals its canonical associate.

Exports

CommutativeRing

modularRep: $R \rightarrow \%$ Residue class

canonicalPreImage: $\% \rightarrow R$ Canonical pre-image

if R has EuclideanDomain then

symmetricPreImage: $\% \rightarrow R$ Symmetric pre-image

ReducibleModulusException

Usage

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

Parameter	Type	Description
R	<code>CommutativeRing</code>	A commutative ring
m	<code>R</code>	The modulus
a	<code>R</code>	A proper factor of m

Description

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

ReducibleModulusExceptionType

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	UnivariatePolynomialAlgebra 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)
```

SimpleAlgebraicExtensionCategory

Usage

`SimpleAlgebraicExtensionCategory(R, Rx):Category`

Parameter	Type	Description
R	<code>IntegralDomain</code>	The coefficient ring of the polynomials
Rx	<code>UnivariatePolynomialAlgebra R</code>	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 `UnivariatePolynomialQuotientSqfr` when p is known to be reducible but squarefree, and `UnivariatePolynomialQuotient` when p is known to be reducible and not squarefree.

Remarks

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

Exports

`IntegralDomain`

`UnivariatePolynomialQuotientSqfr(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`

SourceOfPrimes

Usage

SourceOfPrimes: Category

Description

SourceOfPrimes is the category of domains supporting a (partial) primality test and a source of primes (which may finite or infinite).

Exports

CommutativeRing

prime?: % → Partial Boolean Primality test

prime?: % → Boolean Primality test

getPrime: () → Partial % Prime source seed

nextPrime: () → Partial % Prime source next

if % has EuclideanDomain then

getPrimeOfSize: () → Partial % Prime source seed

Usage

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

Parameter	Type	Description
R	<code>CommutativeRing</code>	The coefficient ring of the polynomials
Rx	<code>UnivariatePolynomialAlgebra R</code>	The type of the modulus
p	<code>Rx</code>	The modulus
x	<code>Symbol</code>	The generator name (optional)

Description

`UnivariatePolynomialMod(R, Rx, p)` implements the univariate polynomials modulo p , *i.e.* the ring $Rx/(p)$, where p is assumed to be monic, but with no other restrictions. Use instead the types `UnivariatePolynomialModSqfr` when p is also known to be squarefree, and `SimpleAlgebraicExtension` when p is also known to be irreducible.

Exports

```
UnivariatePolynomialQuotient(R, Rx)
```

Usage

```
import from UnivariatePolynomialModSqfr(R, Rx, p)
import from UnivariatePolynomialModSqfr(R, Rx, p, x)
```

Parameter	Type	Description
R	<code>CommutativeRing</code>	The coefficient ring of the polynomials
Rx	<code>UnivariatePolynomialAlgebra R</code>	The type of the modulus
p	<code>Rx</code>	The modulus
x	<code>Symbol</code>	The generator name (optional)

Description

`UnivariatePolynomialModSqfr(R, Rx, p)` implements the univariate polynomials modulo p , *i.e.* the ring $Rx/(p)$, where p is assumed to be monic and squarefree, but not necessarily irreducible. Use instead the types `UnivariatePolynomialMod` when p is not squarefree, and `SimpleAlgebraicExtension` when p is known to be irreducible.

Exports

```
UnivariatePolynomialQuotientSqfr(R, Rx)
```

Usage

UnivariatePolynomialQuotient(R, Rx):Category

Parameter	Type	Description
R	CommutativeRing	The coefficient ring of the polynomials
Rx	UnivariatePolynomialAlgebra R	A polynomial type over R

Description

UnivariatePolynomialQuotient(R, Rx) is the category of extensions of R of the form $Rx/(p)$ for some $p \in Rx$, where p is assumed to be monic, but with no other restrictions. Use instead the categories `UnivariatePolynomialQuotientSqfr` when p is also known to be squarefree, and `SimpleAlgebraicExtensionCategory` when p is also known to be irreducible.

Exports

Algebra R		
CommutativeRing		
coefficient:	$(\%, Z) \rightarrow R$	Extraction of a coefficient
compose:	$\% \rightarrow \% \rightarrow \%$	Modular composition
definingPolynomial:	Rx	Defining polynomial
generator:	$\% \rightarrow \text{Generator Cross}(R, Z)$	iterate through all the terms
knownIrreducible?:	Boolean	Irreducible modulus?
lift:	$\% \rightarrow Rx$	Conversion to a polynomial
map:	$(R \rightarrow R) \rightarrow \% \rightarrow \%$	Lift a mapping
map!:	$(R \rightarrow R) \rightarrow \% \rightarrow \%$	Lift a mapping
monom:	$\%$	Generator of the algebra
reduce:	$Rx \rightarrow \%$	Reduction of a polynomial
value:	$(P:\text{POLY } \%) \rightarrow (P, Z, Z) \rightarrow \%$	Evaluation at a rational number

where

$Z == \text{Integer}$
 $\text{POLY} == \text{UnivariatePolynomialAlgebra}$

if R has `CharacteristicZero` then

`CharacteristicZero`

if R has `FiniteCharacteristic` then

`FiniteCharacteristic`

if R has `FiniteSet` then

`FiniteSet`

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 $q_1(p), \dots, q_k(p)$ 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

knownIrreducible?

Signature

knownIrreducible?: **Boolean**

Returns

Returns *true* if the modulus is known to be irreducible, *false* otherwise. Note that the modulus could be irreducible, even if knownIrreducible? is *false*.

Usage

lift q

Signature

lift: % \rightarrow 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

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:UnivariatePolynomialAlgebra %) → (P, Integer, Integer) → %

Parameter	Type	Description
P	UnivariatePolynomialAlgebra %	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

UnivariatePolynomialQuotientSqfr(R, Rx):Category

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

Description

UnivariatePolynomialQuotientSqfr(R, Rx) is the category of extensions of R of the form $Rx/(p)$ for some $p \in Rx$, where p is assumed to be monic and squarefree, but not necessarily irreducible. Use instead the categories `UnivariatePolynomialQuotient` when p is not squarefree, and `SimpleAlgebraicExtensionCategory` when p is known to be irreducible.

Exports

```

UnivariatePolynomialQuotient(R, Rx)
newtonSeries: Rxx          Newton series of the generator
norm:         % → R        Norm
              (P:POLY %) → P → Rx
trace:        % → R        Trace
              (P:POLY %) → P → Rx

```

where

```

POLY == UnivariatePolynomialAlgebra
Rxx  == DenseUnivariateTaylorSeries R

```

if R has RationalRootRing then

```
RationalRootRing
```

if R has CharacteristicZero and R has Field then

```
DifferentialExtension R
```

Usage

newtonSeries

Signature

newtonSeries: DenseUnivariateTaylorSeries R

Returns

Returns the series

$$\sum_{n \geq 0} Tr(\alpha^n) x^n$$

where α is the image in the quotient of the term x from Rx , and Tr is the trace from the quotient into R .

See also

monom, trace

Usage

```

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

```

Signatures

```

norm,trace:  % → R
norm,trace:  (P:UnivariatePolynomialAlgebra %) → P → Rx

```

Parameter	Type	Description
q	<code>%</code>	An element of the algebraic extension
P	<code>UnivariatePolynomialAlgebra %</code>	A polynomial type
p	<code>P</code>	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.

SmallPrimes

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 than half-word-size, which allows for accumulation before reduction. Only a specific set \mathcal{P} of half-word-size primes is available.

Exports

```
PrimeCollection
```

HalfWordSizePrimes

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
```


WordSizePrimes

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
```

PrimeCollection

Usage

PrimeCollection: Category

Description

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

Exports

allPrimes:	$() \rightarrow \text{Generator } Z$	Generate all the primes
fourierPrime:	$Z \rightarrow (Z, Z)$	Fourier prime
maxPrime:	$\rightarrow Z$	Largest prime
nextPrime:	$Z \rightarrow Z$	First prime above a given number
previousPrime:	$Z \rightarrow Z$	First prime below a given number
primeInCollection?:	$Z \rightarrow \text{Boolean}$	Check a prime
primRoot:	$Z \rightarrow Z$	Modular primitive root
randomPrime:	$() \rightarrow Z$	Random prime

where

$Z == \text{MachineInteger}$

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` \rightarrow (`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 \rightarrow MachineInteger

Parameter	Type	Description
<i>n</i>	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` \rightarrow `MachineInteger`

Parameter	Type	Description
<i>n</i>	<code>MachineInteger</code>	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

primeInCollection? *n*

Signature

primeInCollection?: MachineInteger → Boolean

Parameter	Type	Description
<i>n</i>	MachineInteger	An integer

Returns

Returns *true* if *n* is a prime in the collection, *false* otherwise (*n* could still be prime in that case).

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

PrimeField2 implements the finite field with two elements.

Exports

```
SmallPrimeFieldCategory
```

PrimeFieldCategory

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 %) \rightarrow P \rightarrow List Cross(% , Integer) In-field roots

rootsSqfr: (P:POLY %) \rightarrow P \rightarrow List % In-field roots

where

POLY == UnivariatePolynomialAlgebra0

Usage

```
rootsSqfr P
rootsSqfr(P)(p)
```

Signature

```
rootsSqfr: (P:UnivariatePolynomialAlgebra0 %) → P → Generator %
```

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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.

PrimeFieldCategory0

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

FiniteField

SerializableType

lift: $\% \rightarrow \text{Integer}$ Conversion to an integer

Usage

lift x

Signature

lift: % \rightarrow Integer

Parameter	Type	Description
<i>x</i>	%	an element of the field

Returns

Return x seen as an integer.

PrimitiveRoots

Usage

```
import from PrimitiveRoots
```

Description

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

Exports

```
factors:       $Z \rightarrow \text{List } Z$   List of prime factors  
primitiveRoot:  $Z \rightarrow Z$            Modular primitive root
```

where

```
 $Z == \text{MachineInteger}$ 
```


Usage

factors n

Signature

factors: `MachineInteger` \rightarrow `List MachineInteger`

Parameter	Type	Description
n	<code>MachineInteger</code>	An integer

Returns

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

Usage

primitiveRoot p

Signature

primitiveRoot: `MachineInteger` \rightarrow `MachineInteger`

Parameter	Type	Description
p	<code>MachineInteger</code>	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: $(T, \text{Integer}) \rightarrow T$ p^{th} -powering
pExponentiation!: $(T, \text{Integer}) \rightarrow T$ In-place p^{th} -powering

Usage

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

Signature

pExponentiation: $(R, \text{Integer}) \rightarrow R$

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.

SmallPrimeField

Usage

import from SmallPrimeField p

Parameter	Type	Description
<i>p</i>	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

SmallPrimeField0

Usage

import from SmallPrimeField0 p

Parameter	Type	Description
<i>p</i>	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

SmallPrimeFieldCategory

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

SmallPrimeFieldCategory0

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 \rightarrow % conversion to a field element

discreteLogTable: Cross(A, A, Boolean) discrete log table if available

machine: % \rightarrow MachineInteger conversion to a machine integer

where

A == PrimitiveArray MachineInteger

Usage

n::%
machine m

Signatures

coerce: MachineInteger \rightarrow %
machine: % \rightarrow MachineInteger

Parameter	Type	Description
<i>m</i>	%	an element of the field
<i>n</i>	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 == PrimitivesArray 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`).

ZechPrimeField

Usage

import from ZechPrimeField p

Parameter	Type	Description
<i>p</i>	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

Backsolve

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 \rightarrow Z,ARR Z,Z,Z,R) \rightarrow V R Backsolve a triangular system
backsolve: (M,Z \rightarrow Z,ARR Z,Z,M,R) \rightarrow (M, V R) Backsolve a triangular system

if R has GcdDomain then

backsolve: (M,Z \rightarrow Z,ARR Z,Z,Z) \rightarrow V R Backsolve a triangular system
backsolve: (M,Z \rightarrow Z,ARR Z,Z,M) \rightarrow (M, V R) Backsolve a triangular system
backsolve2: (M,Z \rightarrow Z,ARR Z,Z,Z,R) \rightarrow V R Backsolve a triangular system
backsolve2: (M,Z \rightarrow Z,ARR Z,Z,M,R) \rightarrow (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
<i>a</i>	M	A matrix representing a Row Echelon Form (REF)
<i>p</i>	$Z \rightarrow Z$	A permutation of the rows of <i>a</i>
<i>st</i>	PrimitiveArray Z	The stairs of the REF
<i>r</i>	Z	The number of leading columns (before the c^{th} column)
<i>c</i>	Z	The column to be backsolved
<i>b</i>	M	A matrix whose columns have to be backsolved
<i>d</i>	R	A maximal denominator needed for a dependence relation

where

$Z == \text{MachineInteger}$

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 \rightarrow Z,PrimitiveArray Z,Z,Z,R) \rightarrow Vector R

backsolve2: (M,Z \rightarrow Z,PrimitiveArray Z,Z,M,R) \rightarrow (M,Vector R)

Parameter	Type	Description
<i>a</i>	M	A matrix representing a Row Echelon Form (REF)
<i>p</i>	Z \rightarrow Z	A permutation of the rows of <i>a</i>
<i>st</i>	PrimitiveArray Z	The stairs of the REF
<i>r</i>	Z	The number of leading columns (before the c^{th} column)
<i>c</i>	Z	The column to be backsolved
<i>b</i>	M	A matrix whose columns have to be backsolved
<i>d</i>	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.

DenseMatrix

Usage

import from DenseMatrix R

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

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

Exports

MatrixCategory R

DivisionFreeGaussElimination

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.

FractionFreeGaussElimination

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.

HermiteGaussElimination

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	<code>CommutativeRing</code>	The coefficient domain
M	<code>MatrixCategory R</code>	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 $([a_0, \dots, a_n], m)$ 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 $v, f(v), \dots, f^n(v)$ (resp. $v, Av, \dots, A^n v$).

Remarks

The relation is as small as possible, meaning that $v, f(v), \dots, f^{n-1}(v)$ (resp. $v, Av, \dots, A^{n-1}v$) 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: $M \rightarrow R$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns the determinant of *a*.

See also

factorOfDeterminant

Usage

factorOfDeterminant a

Signature

factorOfDeterminant: $M \rightarrow (\text{Boolean}, R)$

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) \rightarrow Vector R

Parameter	Type	Description
<i>gen</i>	Generator Vector R	A generator of vectors
<i>n</i>	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

Signatureinverse: $M \rightarrow (M, \text{Vector } R)$

Parameter	Type	Description
a	M	A matrix

ReturnsReturns $(b, [d_1, \dots, d_n])$ 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, \dots, 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 $\text{transpose}(a)$. 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: $M \rightarrow (\text{Boolean}, \text{Array MachineInteger}, \text{Array MachineInteger})$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns $(\text{max?}, [r_1, \dots, r_r], [c_1, \dots, c_r])$ 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: $M \rightarrow M$

Parameter	Type	Description
<i>a</i>	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: $M \rightarrow (\text{Array MachineInteger}, \text{Array MachineInteger})$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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 $(m, [d_1, \dots, d_n])$ 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: $M \rightarrow \text{MachineInteger}$

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns the rank of *a*.

See also

rankLowerBound, span

Usage

rankLowerBound a

Signature

rankLowerBound: $M \rightarrow (\text{Boolean}, \text{MachineInteger})$

Parameter	Type	Description
a	M	A matrix

Returns

Returns $(\text{rank?}, r)$ 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 \rightarrow Array MachineInteger

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns $[c_1, \dots, c_r]$ where r is the rank of a and the span of a is generated by its columns c_1, \dots, 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 $(w, m, [d_1, \dots, d_n])$ 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
<i>a</i>	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

LinearAlgebraRing: 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 $(b, [d_1, \dots, d_n])$ 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 $(max?, [r_1, \dots, r_r], [c_1, \dots, c_r])$ 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
<i>M</i>	MatrixCategory %	A matrix type
<i>a</i>	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
<i>gen</i>	Generator Vector R	A generator of vectors
<i>n</i>	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 $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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 $(m, [d_1, \dots, d_n])$ 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
<i>M</i>	MatrixCategory %	A matrix type
<i>a</i>	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	<code>MatrixCategory %</code>	A matrix type
a, b	<code>M</code>	Matrices

Returns

Returns $(w, m, [d_1, \dots, d_n])$ 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 $[c_1, \dots, c_r]$ where r is the rank of a and the span of a is generated by its columns c_1, \dots, 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 computing a row echelon form (REF) of a matrix and the first dependence relation among vectors.

Exports

extendedRowEchelon:	$M \rightarrow (M, Z \rightarrow Z, Z, \text{ARR } Z, Z, M)$	REF of a matrix
extendedRowEchelon!:	$M \rightarrow (Z \rightarrow Z, Z, \text{ARR } Z, Z, M)$	REF of a matrix
extendedRowEchelonForm:	$M \rightarrow (M, M)$	REF of a matrix
maxInvertibleSubmatrix:	$M \rightarrow (\text{Array } Z, \text{Array } Z)$	Maximal minor
maxInvertibleSubmatrix!:	$M \rightarrow (\text{Array } Z, \text{Array } Z)$	Maximal minor
pivot:	$(M, Z \rightarrow Z, Z, Z) \rightarrow Z$	Select a pivot
rowEchelon:	$M \rightarrow (M, Z \rightarrow Z, Z, \text{ARR } Z, Z)$	REF of a matrix
rowEchelon!:	$M \rightarrow (Z \rightarrow Z, Z, \text{ARR } Z, Z)$	REF of a matrix
rowEchelonForm:	$(M, M) \rightarrow (Z \rightarrow Z, Z, \text{ARR } Z, Z)$	REF of a matrix
span:	$M \rightarrow M$	REF of a matrix
span!:	$M \rightarrow \text{Array } Z$	Span of a matrix
span!:	$M \rightarrow \text{Array } Z$	Span of a matrix

if R has IntegralDomain then

denominators:	$(M, Z \rightarrow Z, Z, \text{ARR } Z) \rightarrow \text{ARR } R$	Maximal denominators
dependence:	$(\text{Generator } V, Z) \rightarrow (M, Z \rightarrow Z, Z, R)$	First linear dependence
deter:	$(M, Z \rightarrow Z, Z, Z) \rightarrow R$	Determinant
determinant:	$M \rightarrow R$	Determinant
determinant!:	$M \rightarrow R$	Determinant
rank!:	$M \rightarrow Z$	Rank of a matrix
rank:	$M \rightarrow Z$	Rank of a matrix

where

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

Usage

denominators(*a*,*p*,*r*,*st*)

Signature

denominators: (M,Z \rightarrow Z,Z, PrimitiveArray Z) \rightarrow PrimitiveArray R

where

Z == MachineInteger

Parameter	Type	Description
<i>a</i>	M	A matrix in REF form
<i>p</i>	MachineInteger \rightarrow MachineInteger	A permutation of the rows of <i>a</i>
<i>r</i>	MachineInteger	The number of stairs of the REF
<i>st</i>	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) \rightarrow (M,Z \rightarrow Z,Z,R)

where

Z == MachineInteger

Parameter	Type	Description
<i>gen</i>	Generator Vector R	A generator of vectors
<i>n</i>	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
<i>a</i>	M	A matrix in REF form
<i>p</i>	MachineInteger → MachineInteger	A permutation of the rows of <i>a</i>
<i>r</i>	MachineInteger	The number of stairs of the REF
<i>d</i>	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: $M \rightarrow R$

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: $M \rightarrow (M, Z \rightarrow Z, Z, \text{PrimitiveArray } Z, Z, M)$
 extendedRowEchelon!: $M \rightarrow (Z \rightarrow Z, Z, \text{PrimitiveArray } Z, Z, M)$

where

$Z == \text{MachineInteger}$

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 $j_1 < j_2 < \dots < j_r$ (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: $M \rightarrow (M, M)$

Parameter	Type	Description
<i>a</i>	<i>M</i>	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 $j_1 < j_2 < \dots < j_r$ (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
<i>a</i>	M	A matrix

Returns

Returns $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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
<i>a</i>	M	A matrix
<i>p</i>	MachineInteger → MachineInteger	A permutation
<i>c</i>	MachineInteger	A column index
<i>r</i>	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: $M \rightarrow \text{MachineInteger}$

Parameter	Type	Description
-----------	------	-------------

<i>a</i>	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 $j_1 < j_2 < \dots < j_r$ (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
<i>a</i>	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 \rightarrow Array MachineInteger

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

Returns

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

$*$: $(\%, V) \rightarrow V$

multiplication by a vector

$[]$: $\text{Tuple } V \rightarrow \%$

create a matrix

$\text{Generator } V \rightarrow \%$

apply: $(\%, I, I) \rightarrow R$

extract an entry

$(\%, I, I, I, I) \rightarrow \%$

extract a submatrix

$(\%, \text{Array } I, \text{Array } I) \rightarrow \%$

colCombine!: $(\%, R, I, R, I) \rightarrow \%$

In-place linear combination of columns

$(\%, R, I, R, I, I, I) \rightarrow \%$

$(\%, (R, R) \rightarrow R, I, I) \rightarrow \%$

$(\%, (R, R) \rightarrow R, I, I, I, I) \rightarrow \%$

colSwap!: $(\%, I, I) \text{ to } \%$

Swap columns in-place

$(\%, I, I, I, I) \text{ to } \%$

column: $(\%, I) \rightarrow V$

extraction of a column

columns: $\% \rightarrow \text{Generator } V$

iteration over the columns

companion: $V \rightarrow \%$

creates a companion matrix

$(V, R) \rightarrow \%$

diagonal: $V \rightarrow \%$

creates a diagonal matrix

$(R, I) \rightarrow \%$

diagonal?: $\% \rightarrow \text{Boolean}$

test for a diagonal matrix

dimensions: $\% \rightarrow (I, I)$

get row and column dimensions

map: $(R \rightarrow R) \rightarrow V \rightarrow \%$

lift a mapping

map: $(R \rightarrow R) \rightarrow \% \rightarrow \%$

lift a mapping

map!: $(R \rightarrow R) \rightarrow \% \rightarrow \%$

lift a mapping

numberOfColumns: $\% \rightarrow I$

number of columns of the matrix

numberOfRows: $\% \rightarrow I$

number of rows of the matrix

one: $I \rightarrow \%$

identity matrix

one?: $\% \rightarrow \text{Boolean}$

test for an identity matrix

row: $(\%, I) \rightarrow V$

extraction of a row

<code>rowCombine!:</code>	$(\%, R, I, R, I) \rightarrow \%$ $(\%, R, I, R, I, I, I) \rightarrow \%$ $(\%, (R, R) \rightarrow R, I, I) \rightarrow \%$ $(\%, (R, R) \rightarrow R, I, I, I, I) \rightarrow \%$	In-place linear combination of rows
<code>rowSwap!:</code>	$(\%, I, I) \text{ to } \%$ $(\%, I, I, I, I) \text{ to } \%$	Swap rows in-place
<code>rows:</code>	$\% \rightarrow \text{Generator } V$	iteration over the rows
<code>scalar?:</code>	$\% \rightarrow \text{Boolean}$	test for a scalar matrix
<code>set!:</code>	$(\%, I, I, R) \rightarrow R$	set an entry in the matrix
<code>setMatrix!:</code>	$(\%, I, I, \%) \rightarrow \%$	modify a submatrix of a matrix
<code>square?:</code>	$\% \rightarrow \text{Boolean}$	test for a square matrix
<code>tensor:</code>	$(\%, \%) \rightarrow \%$	tensor product
<code>transpose:</code>	$V \rightarrow \%$	transpose a vector
<code>transpose:</code>	$\% \rightarrow \%$	transpose a matrix
<code>transpose!:</code>	$\% \rightarrow \%$	transpose a matrix in-place
<code>zero:</code>	$(I, I) \rightarrow \%$	create a zero matrix
<code>zero!:</code>	$\% \rightarrow ()$	make all the entries zero
<code>zero?:</code>	$\% \rightarrow \text{Boolean}$	test if all entries are zero

if R has Ring then

`random:` $() \rightarrow \%$ create a random matrix
 $(I, I) \rightarrow \%$

if R has DifferentialRing then

`wronskian:` $V \rightarrow \%$ Wronskian matrix

where

`I` == MachineInteger

`V` == Vector R

if R has SerializableType then

SerializableType

Usage

A * v

Signature

*: (% , Vector R) → Vector R

Parameter	Type	Description
<i>A</i>	%	a matrix
<i>v</i>	Vector R	a vector

Returns

Returns the vector *Av*.

Usage

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

Signatures

[]: Tuple Vector R \rightarrow %
[]: Generator Vector R \rightarrow %

Parameter	Type	Description
v_1, \dots, 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

$A(n, m)$
 $\text{apply}(A, n, m)$
 $A(n, m, r, c)$
 $\text{apply}(A, n, m, r, c)$
 $A(a, b)$
 $\text{apply}(A, a, b)$

Signatures

$\text{apply}: (\%, \text{MachineInteger}, \text{MachineInteger}) \rightarrow R$
 $\text{apply}: (\%, \text{MachineInteger}, \text{MachineInteger}, \text{MachineInteger}, \text{MachineInteger}) \rightarrow \%$
 $\text{apply}: (\%, \text{Array MachineInteger}, \text{Array MachineInteger}) \rightarrow \%$

Parameter	Type	Description
A	$\%$	A matrix
n, m, r, c	<code>MachineInteger</code>	indices
a, b	<code>Array MachineInteger</code>	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. For more general submatrices, $A([a_1, \dots, a_r], [b_1, \dots, b_s])$ returns the submatrix of A consisting of the intersection of the rows a_1, \dots, a_r and columns b_1, \dots, b_r of A .

Usage

```

colCombine!(A, c1, j1, c2, j2)
colCombine!(A, c1, j1, c2, j2, i1, i2)
colCombine!(A, f, j1, j2)
colCombine!(A, f, j1, j2, i1, i2)
rowCombine!(A, c1, i1, c2, i2)
rowCombine!(A, c1, i1, c2, i2, j1, j2)
rowCombine!(A, f, i1, i2)
rowCombine!(A, f, i1, i2, j1, j2)

```

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	$(R,R) \rightarrow R$	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^{th} column (resp. i_1^{th} row) of A is replaced by the result of applying f pointwise to its j_1^{th} and j_2^{th} columns (resp. i_1^{th} and i_2^{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, j1, j2)
colSwap!(A, j1, j2, i1, i2)
rowSwap!(A, i1, i2)
rowSwap!(A, i1, i2, j1, j2)
```

Signatures

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

where

```
I == MachineInteger
```

Parameter	Type	Description
A	<code>%</code>	A matrix
$j1, j2$	<code>MachineInteger</code>	column indices
$i1, i2$	<code>MachineInteger</code>	row indices

Description

The j_1^{th} and j_2^{th} columns (resp. i_1^{th} and i_2^{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

ReturnsReturns 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: % \rightarrow Generator Vector R

Parameter	Type	Description
<i>A</i>	%	A matrix

Description

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

Usage

```
companion [r1,...,rn]
companion([r1,...,rn],a)
```

Signature

```
companion: (Vector R, R) -> %
```

Parameter	Type	Description
r_1, \dots, 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 [r1, ..., rn]`
`diagonal(r, n)`
`diagonal? A`
`scalar? A`
`square? A`

Signatures

`diagonal:` `Vector R → %`
`diagonal:` `(R, MachineInteger) → %`
`diagonal?, scalar?, square?:` `% → Boolean`

Parameter	Type	Description
r, r_1, \dots, r_n	<code>R</code>	Entries
n	<code>MachineInteger</code>	A size
A	<code>%</code>	A matrix

Description

`diagonal([r1, ..., rn])` returns a diagonal matrix whose diagonal elements are r_1, \dots, r_n , while `diagonal(r, n)` returns an $n \times n$ diagonal matrix with r on its diagonal, and `square?(A)` (resp. `diagonal A` and `scalar? A`) return *true* if A is a square (resp. diagonal and diagonal with the same entry on its diagonal) matrix, *false* otherwise.

See also

`companion, one?`

Usage

dimensions A

Signature

dimensions: % \rightarrow (MachineInteger,MachineInteger)

Parameter	Type	Description
<i>A</i>	%	A matrix

Returns

The number of rows and columns in *A*

See also

numberOfColumns,numberOfRows

Usage

```
map f
map! f
map(f)([v1, ..., vn])
map(f)(A)
map!(f)(A)
```

Signatures

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

Parameter	Type	Description
f	$R \rightarrow R$	a map
v_i	R	Entries of a vector
A	$\%$	A matrix

Description

$\text{map}(f)([v_1, \dots, v_n])$ returns the square matrix

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

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

Usage

numberOfColumns A
numberOfRows A

Signature

numberOfColumns,numberOfRows: % \rightarrow 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
<i>n,m</i>	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
<i>A</i>	%	A matrix
<i>n, m</i>	MachineInteger	Indices
<i>c</i>	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
<i>A</i> , <i>B</i>	<i>%</i>	Matrices
<i>n</i> , <i>m</i>	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
<i>v</i>	Vector R	A vector
<i>A</i>	%	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 [v₁, ..., v_n]

Signature

wronskian: Vector R → %

Parameter	Type	Description
v _i	R	Entries of a vector

Description

wronskian([v₁, ..., v_n]) returns the square matrix

$$\begin{pmatrix} v_1 & \dots & v_n \\ v'_1 & \dots & v'_n \\ \vdots & & \vdots \\ v_1^{(n-1)} & \dots & v_n^{(n-1)} \end{pmatrix}$$

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 MatrixCategory2(R, MR, S, MS)
```

Parameter	Type	Description
R, S	ExpressionType ArithmeticType	Coefficient domains
MR	MatrixCategory R	a matrix type over R
MS	MatrixCategory S	a matrix type over S

Description

MatrixCategory2(R,MR,S,MS) provides tools for lifting maps $R \rightarrow S$ to maps $MR \rightarrow MS$.

Exports

```
map: (R → S) → MR → MS  Lift a mapping
```

Usage

map f
map(f)(m)

Signature

map: (R → S) → MR → MS

Parameter	Type	Description
f	$R \rightarrow S$	A map
m	MR	A matrix with entries in R

Description

map(f)(m) returns a matrix whose $(i, j)^{\text{th}}$ entry is $f(m_{ij})$, while map(f) returns the mapping $m \rightarrow f(m)$.

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

$(v, A) := \text{makeColIntegral } B$
 $(a, v) := \text{makeColIntegral}(B, i)$

Signatures

$\text{makeColIntegral}: \text{MQ} \rightarrow (\text{Vector } R, \text{MR})$
 $\text{makeColIntegral}: (\text{MQ}, \text{MachineInteger}) \rightarrow (R, \text{Vector } R)$

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

Returns

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

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

$\text{makeColIntegral}(B, i)$ 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

$(v, A) := \text{makeRowIntegral } B$
 $(a, v) := \text{makeRowIntegral}(B, i)$

Signatures

$\text{makeRowIntegral}: \text{MQ} \rightarrow (\text{Vector } R, \text{MR})$
 $\text{makeRowIntegral}: (\text{MQ}, \text{MachineInteger}) \rightarrow (R, \text{Vector } R)$

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

Returns

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

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

$\text{makeRowIntegral}(B, i)$ 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 $v.j$.

ModulopGaussElimination

Usage

```
import from ModulopGaussElimination
```

Exports

deter:	$(M, Z, V, Z, Z, Z) \rightarrow Z$	Determinant
determinant!:	$(M, Z, Z) \rightarrow Z$	Determinant
extendedRowEchelon!:	$(M, Z, Z, Z) \rightarrow (V, Z, V, M)$	REF of a matrix
firstDependence!:	$(\text{Generator } V, Z, Z, M, M) \rightarrow Z$	First dependence relation
inverse!:	$(M, Z, Z, M, V) \rightarrow ()$	Inverse
kernel!:	$(M, Z, Z, M) \rightarrow Z$	Kernel
maxInvertibleSubmatrix!:	$(M, Z, Z) \rightarrow (V, V)$	Maximal minor
particularSolution!:	$(M, Z, Z, M, Z, Z, M, V) \rightarrow ()$	A solution
rank!:	$(M, Z, Z, Z) \rightarrow Z$	Rank
rowEchelon!:	$(M, Z, Z, Z) \rightarrow (V, Z, V)$	REF of a matrix
solve!:	$(M, Z, Z, M, Z, Z, M, V, M) \rightarrow Z$	All solutions
span!:	$(M, Z, Z) \rightarrow V$	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
<i>a</i>	A A MachineInteger	A square matrix in REF form
<i>n</i>	MachineInteger	The size of <i>a</i>
σ	A MachineInteger	A permutation of the rows of <i>a</i>
<i>r</i>	MachineInteger	The number of stairs of the REF
<i>d</i>	MachineInteger	The sign of σ
<i>p</i>	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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A square matrix
<i>n</i>	MachineInteger	The size of <i>a</i>
<i>p</i>	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) \rightarrow (A Z, Z, A Z, Z, A A Z)

where

Z == MachineInteger

A == PrimitiveArray

Parameter	Type	Description
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	MachineInteger	A prime

Description

We say that a matrix *a* is in REF if there are *r* (the rank) and $j_1 < j_2 < \dots < j_r$ (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
<i>gen</i>	Generator A MachineInteger	A generator of vectors
<i>n</i>	MachineInteger	The dimension of the vectors generated
<i>p</i>	MachineInteger	a prime
<i>a</i>	A A MachineInteger	A work matrix
<i>d</i>	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
<i>a</i> , <i>w</i>	PrimitiveArray PrimitiveArray MachineInteger	Matrices
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	MachineInteger	a prime

Returns

Returns $([r_1, \dots, r_r], [c_1, \dots, c_r])$ 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
<i>a,b,w</i>	PrimitiveArray PrimitiveArray MachineInteger	Matrices
<i>ra</i>	MachineInteger	Number of rows of <i>a</i>
<i>ca</i>	MachineInteger	Number of columns of <i>a</i>
<i>cb</i>	MachineInteger	Number of columns of <i>b</i>
<i>p</i>	MachineInteger	a prime
<i>d</i>	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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	MachineInteger	A prime

Description

We say that a matrix *a* is in REF if there are *r* (the rank) and $j_1 < j_2 < \dots < j_r$ (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
<i>a,b,w,k</i>	PrimitiveArray PrimitiveArray MachineInteger	Matrices
<i>ra</i>	MachineInteger	Number of rows of <i>a</i>
<i>ca</i>	MachineInteger	Number of columns of <i>a</i>
<i>cb</i>	MachineInteger	Number of columns of <i>b</i>
<i>p</i>	MachineInteger	a prime
<i>d</i>	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
<i>a</i>	PrimitiveArray PrimitiveArray MachineInteger	A matrix
<i>r</i>	MachineInteger	Number of rows of <i>a</i>
<i>c</i>	MachineInteger	Number of columns of <i>a</i>
<i>p</i>	MachineInteger	a prime

Returns

Returns $[c_1, \dots, c_r]$ 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, \dots, c_r .

Remarks

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

OrdinaryGaussElimination

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.

OverdeterminedLinearSystemSolver

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 \rightarrow M) \rightarrow M Solve an overdetermined system

Usage

kernel!(m, f)

Signature

kernel!: (M, M → M) → M

Parameter	Type	Description
<i>m</i>	M	A matrix
<i>f</i>	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: $M \rightarrow \text{Partial Cross (Boolean, MachineInteger)}$ Probable rank

Usage

rankLowerBound a

Signature

rankLowerBound: $M \rightarrow \text{Partial Cross}(\text{Boolean}, \text{MachineInteger})$

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.

TwoStepFractionFreeGaussElimination

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	UnivariatePolynomialAlgebra 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: $M \rightarrow \text{Integer}$ Degree bound for the determinant
 determinant: $M \rightarrow RX$ Determinant
 $(M, RX) \rightarrow RX$
 $(M, RX, \text{Integer}) \rightarrow RX$

Usage

degreeBound *a*

Signature

degreeBound: *M* → Integer

Parameter	Type	Description
<i>a</i>	<i>M</i>	A matrix

Returns

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

Usage

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

Signatures

determinant: $M \rightarrow RX$
determinant: $(M, RX) \rightarrow RX$
determinant: $(M, RX, Integer) \rightarrow RX$

Parameter	Type	Description
<i>a</i>	<i>M</i>	A matrix
<i>d</i>	<i>RX</i>	A known factor of $ a $ (optional)
<i>n</i>	<i>Integer</i>	A known degree bound on $ a $ (optional)

Returns

All calls to `determinant` return 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	UnivariatePolynomialAlgebra 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:	$M \rightarrow FX$	Determinant
hermite:	$M \rightarrow M$	Hermite form
kernel:	$M \rightarrow M$	Kernel
maxInvertibleSubmatrix:	$M \rightarrow (AZ, AZ)$	Maximal minor
popov:	$M \rightarrow M$	weak Popov form
rank:	$M \rightarrow \text{MachineInteger}$	Rank
span:	$M \rightarrow AZ$	Span

where

```
AZ == Array MachineInteger
```

Usage

hermite a

Signature

hermite: M → M

Parameter	Type	Description
<i>a</i>	M	A matrix

Returns

Returns the Hermite form of *a*.

Vector

Usage

import from Vector R

Parameter	Type	Description
R	ExpressionType AdditiveType	The coefficient domain

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 \rightarrow % zero vector

zero!: % \rightarrow () make all the entries zero

zero?: % \rightarrow Boolean test if all entries are zero

if R has ArithmeticType then

LinearCombinationType R

dot: (% , %) \rightarrow R dot product

tensor: (% , %) \rightarrow % tensor product

if R has Ring then

random: () \rightarrow % random vector

MachineInteger \rightarrow %

Usage`dot(u,v)`**Signature**`dot: (%,%) \rightarrow R`

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

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

Usage

random()
random n

Signatures

random: $() \rightarrow \%$
random: `MachineInteger` $\rightarrow \%$

Parameter	Type	Description
n	<code>MachineInteger</code>	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
<i>u, v</i>	%	Vectors with coefficients from R.

Returns

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

Usage

```
zero n
zero! v
zero? v
```

Signatures

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

Parameter	Type	Description
n	<code>MachineInteger</code>	The dimension of the new vector
v	<code>%</code>	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.

VectorOverFraction

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

```
UnivariatePolynomialAlgebra 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

```
UnivariateTaylorSeriesType R
```


FactorizationRing

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 == UnivariatePolynomialAlgebra0

Usage

factor(P)(p)

Signature

factor: (P: UnivariatePolynomialAlgebra0 %) → P → (%, Product P)

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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 == UnivariatePolynomialAlgebra0

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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.

FFTRing

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 == UnivariatePolynomialAlgebra0

Usage

fft(P)(p,q)

Signature

fft: (P: UnivariatePolynomialAlgebra0 %) → (P, P) → Partial P

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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: UnivariatePolynomialAlgebra0 %) → (P, P, P) → Boolean

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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 GenericModularPolynomialGcdPackage (R, U)

Parameter	Type	Description
R	EuclideanDomain SourceOfPrimes ModularComputation	
U	UnivariatePolynomialAlgebra (R)	

Description

GenericModularPolynomialGcdPackage (R, U) provides a generic modular gcd algorithm. See the paper *On the genericity of the Modular Gcd Algorithm* by Kaltofen and Monagan in the proc. of ISSAC 1999.

Exports

modularGcd: $(U, U) \rightarrow \text{Partial } (U)$ gcd (failure if not enough primes)

HeuristicGcd

Usage

```
import from HeuristicGcd(Z, P)
```

Parameter	Type	Description
Z	IntegerCategory	An integer-like ring
P	UnivariatePolynomialAlgebra0 Z	Polynomials over Z

Description

HeuristicGcd provides an implementation of the HEUGCD algorithm for univariate polynomials over the integers.

Exports

balancedRemainder:	$(Z, Z) \rightarrow Z$	Symmetric Remainder
heuristicGcd:	$(P, P) \rightarrow (\text{Partial } P, P, P)$	the Heuristic gcd algorithm
radixInterpolate:	$(Z, Z) \rightarrow P$	Radix interpolation

Usage

balancedRemainder(*n*, *m*)

Signature

balancedRemainder: $(\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z}$

Parameter	Type	Description
<i>n</i>	\mathbb{Z}	An integer
<i>m</i>	\mathbb{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: $(P, P) \rightarrow (\text{Partial } P, P, P)$

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 $(\text{failed}, p_1, p_2)$ is returned, although this has never been reported.

Usage

radixInterpolate(n, m)

Signature

radixInterpolate: $(\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{P}$

Parameter	Type	Description
n	\mathbb{Z}	A point
m	\mathbb{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	UnivariatePolynomialAlgebra 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: $(P, P) \rightarrow (\text{Partial } P, P, P)$ The Modular Gcd algorithm

Usage

modularGcd(p_1, p_2)

Signatures

modularGcd: (P, P) \rightarrow (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.

ModulopUnivariateGcd

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, [l1, ..., lp-1], [e1, ..., ep-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
$[l_0, \dots, l_{p-1}]$	PrimitiveArray MachineInteger	log table
$[e_0, \dots, e_{p-1}]$	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

```
import from PrimeFieldUnivariateFactorizer(F, P)
```

Parameter	Type	Description
F	PrimeFieldCategory0	Coefficient ring of the polynomials
P	UnivariatePolynomialAlgebra F	A polynomial ring

Description

PrimeFieldUnivariateFactorizer provides implementations of various univariate factorization algorithms over a prime field.

Exports

berlekamp:	$P \rightarrow \text{List } P$	Berlekamp's algorithm
cantorZassenhaus:	$P \rightarrow \text{List } P$	Cantor-Zassenhaus algorithm
factor:	$P \rightarrow (F, \text{PROD})$	Factor (default algorithm)
factor:	$(P \rightarrow \text{List } P) \rightarrow (P \rightarrow (F, \text{PROD}))$	Factor (given algorithm)
roots:	$(P, \text{Boolean}) \rightarrow \text{Generator FR } F$	Roots of a polynomial

where

```
FR    == FractionalRoot
PROD  == Product P
```

Usage

berlekamp p

Signature

berlekamp: $P \rightarrow \text{List } P$

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: $P \rightarrow \text{List } P$

Parameter	Type	Description
<i>p</i>	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.
$sqrfree?$	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 .

RationalRootRing

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 == UnivariatePolynomialAlgebra0

Usage

integerRoots(P)(p)
rationalRoots(P)(p)

Signature

integerRoots,rationalRoots: (P: POL %) → P → Generator RationalRoot
where
POL == UnivariatePolynomialAlgebra0

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	A polynomial type
p	P	A polynomial

Returns

Return $[(r_1, e_1), \dots, (r_n, e_n)]$ where the r_i 's are the integer or rational roots of p and have multiplicity e_i .

Resultant

Usage

```
import from Resultant(R,P)
```

Parameter	Type	Description
R	IntegralDomain	Coefficient ring for the polynomials
P	UnivariatePolynomialAlgebra0 R	A polynomial ring

Exports

lastSPRS:	$(P,P) \rightarrow P$	last non-zero remainder in the SPRS
extendedLastSPRS:	$(P,P) \rightarrow (P,P,P)$	extended last non-zero remainder in the SPRS
resultant:	$(P,P) \rightarrow R$	polynomial resultant
SPRS:	$(P,P) \rightarrow \text{List } P$	Subresultant Polynomial Remainder Sequence
subResultantGcd:	$(P,P) \rightarrow P$	GCD

Description

Resultant(R,P) implements polynomial resultant computations.

Usage

lastSPRS(a,b)

Signature

lastSPRS: (P,P) \rightarrow P

Parameter	Type	Description
<i>a, b</i>	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: $(P,P) \rightarrow (P,P,P)$

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: $(P,P) \rightarrow R$

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.

UsageSPRS(*a*,*b*)**Signature**

SPRS: (P,P) → List P

Parameter	Type	Description
<i>a</i> , <i>b</i>	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: $(P,P) \rightarrow P$

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

```
UnivariatePolynomialRing R
```

SparseUnivariatePolynomial

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

```
UnivariatePolynomialAlgebra R
```


Usage

```
import from UnivariateFactorialPolynomial(R, Rx)
```

Parameter	Type	Description
R	Ring	The coefficient domain
Rx	UnivariatePolynomialAlgebra 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 $(x^n)_{n \geq 0}$, where $x^n = x(x-1)\dots(x-n+1)$. Rx is used for representing the factorial polynomials, so you can choose between sparse and dense representations.

Exports

```
UnivariateFreeRing 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 $(x^n)_{n \geq 0}$, while $\text{expand}(q)$ performs the reverse conversion and $\text{trailExpand}(q)$ returns (n, h) such that $q = x^n h$.

Usage

integerRoots p
 rationalRoots p

Signature

integerRoots,rationalRoots: % \rightarrow List FractionalRoot Integer

Parameter	Type	Description
p	%	A factorial polynomial

Returns

Return $[(r_1, e_1), \dots, (r_n, e_n)]$ where the r_i 's are the integer or rational roots of p and have multiplicity e_i .

Usage

UnivariateFreeLinearArithmeticType R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariateFreeLinearArithmeticType 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 `RRing 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	<code>%</code>	A polynomial or series
t	<code>ExpressionTree</code>	An expression tree
x	<code>Symbol</code>	A name for the variables
$port$	<code>TextWriter</code>	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^3+4*x-2
to the standard stream stdout.

```

Usage

for c in coefficients m repeat { ... }

Signature

coefficients: % \rightarrow 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	<code>%</code>	A polynomial or series
m	<code>Integer</code>	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

UnivariateFreeRing R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariateFreeRing 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
IndexedFreeRRing(R, Integer)
UnivariateFreeLinearArithmeticType 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 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	$\text{Vector } R$	A coefficient vector

Description

$[v_1, \dots, v_n]::\%$ returns the polynomial $\sum_{i=0}^{n-1} v_{i+1} P_i$, while $\text{vectorize!}(v)(\sum_{j=0}^d a_j P_j)$ fills v with $[a_0, \dots, a_d, 0, \dots]$ and returns v .

Remarks

If $d > \#v - 1$, then the high coefficients of p are simply ignored.

Usage

companion p

Signature

companion: % \rightarrow 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

monomial!(p, c, n)

Signature

monomial!: (% , R, Integer) \rightarrow %

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	$() \rightarrow R$	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:  % → %
```

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) \rightarrow %

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 i th coefficient of s equals the $l + i$ th coefficient of pq . Here $l \leq h$ should hold.

Usage

import from UnivariateFreeRing2(*R*,*R**x*,*S*,*S**y*)

Parameter	Type	Description
<i>R</i> , <i>S</i>	ExpressionType ArithmeticType	Coefficient domains
<i>R</i> <i>x</i>	UnivariateFreeRing <i>R</i>	A monogenic algebra over <i>R</i>
<i>S</i> <i>y</i>	UnivariateFreeRing <i>S</i>	A monogenic algebra over <i>R</i>

Description

UnivariateFreeRing2(*R*,*R**x*,*S*,*S**y*) provides tools for lifting maps $R \rightarrow S$ to maps $Rx \rightarrow Sy$.

Exports

map: $(R \rightarrow S) \rightarrow Rx \rightarrow Sy$ Lift a mapping

Usage

```
map f
map(f)(p)
```

Signature

```
map:  (R → S) → Rx → Sy
```

Parameter	Type	Description
f	$R \rightarrow S$	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 $p \rightarrow f(p)$.

UnivariateFreeRingOverFraction

Usage

import from UnivariateFreeRingOverFraction(R , PR , Q , PQ)

Parameter	Type	Description
R	IntegralDomain	an integral domain
PR	UnivariateFreeRing R	a univariate free finite algebra type over R
Q	FractionCategory R	a fraction domain of R
PQ	UnivariateFreeRing R	a univariate free finite algebra type over Q

Description

UnivariateFreeRingOverFraction(R , PR , Q , PQ) provides useful conversions between polynomials with integral and rational coefficients.

Exports

LinearCombinationFraction(R , PR , Q , PQ)

UnivariateGcdRing

Usage

UnivariateGcdRing: Category

Description

UnivariateGcdRing is the category of rings which export a gcd algorithm for univariate polynomials over themselves.

Exports

GcdDomain

gcdUP: (P:UnivariatePolynomialAlgebra0) \rightarrow (P, P) \rightarrow P Gcd

gcdUP!: (P:UnivariatePolynomialAlgebra0) \rightarrow (P, P) \rightarrow P Gcd

gcdquoUP: (P:UnivariatePolynomialAlgebra0) \rightarrow (P, P) \rightarrow (P,P,P) Gcd

Usage

gcdUP(P)(p, q)
 gcdUP!(P)(p, q)

Signature

gcdUP: (P: UnivariatePolynomialAlgebra0 %) → (P, P) → P

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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: UnivariatePolynomialAlgebra0 %) → (P,P) → (P,P,P)

Parameter	Type	Description
P	UnivariatePolynomialAlgebra0 %	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	UnivariatePolynomialAlgebra0 Z	A polynomial type over Z

Description

UnivariateIntegralFactorizer(Z , P) implements a factorizer for polynomials with integer coefficients.

Exports

factor:	$P \rightarrow (Z, \text{Product } P)$	Factor
integerRoots:	$P \rightarrow \text{Generator FractionalRoot Integer}$	Integer roots
rationalRoots:	$P \rightarrow \text{Generator FractionalRoot Integer}$	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: $P \rightarrow \text{Generator FractionalRoot Integer}$

Parameter	Type	Description
p	P	A polynomial with integer coefficients

Returns

integerRoots(p) (resp. rationalRoots(p)) return $[(r_1, e_1), \dots, (r_n, e_n)]$ 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 ArithmeticType	Coefficients of the monomials
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`) \rightarrow `ExpressionTree`

Parameter	Type	Description
p	<code>%</code>	A monomial
t	<code>ExpressionTree</code>	An expression tree

Returns

Returns p as an expression tree using t as root variable name.

Usage

coefficient p

Signature

coefficient: % \rightarrow 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: % \rightarrow 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 → R) → % → %
```

Parameter	Type	Description
f	$R \rightarrow R$	A map
p	$\%$	A monomial

Description

`map(f)(p)` returns $f(a)x^n$ where $p = ax^n$, while `map(f)` returns the mapping $p \rightarrow f(p)$. 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: % \rightarrow %

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	<code>%</code>	A monomial
n	<code>Integer</code>	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 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

UnivariatePolynomialRing R

IndexedFreeAlgebra(R, Integer)

apply: $(\%, R) \rightarrow R$ Evaluate a polynomial

apply: $(\%, \%) \rightarrow \%$ Evaluate a polynomial

equal?: $(\%, \%, \%, \text{Integer}) \rightarrow \text{Boolean}$ Truncated equality

Horner: $(\%, R) \rightarrow (\%, R)$ Horner division by $x - a$

if R has CharacteristicZero then

ordinaryPoint: $\% \rightarrow \text{Integer}$ Point where a polynomial is nonzero

if R has CommutativeRing then

DifferentialRing

lift: $(\text{Derivation } R, \%) \rightarrow \text{Derivation } \%$ Extend a derivation

monicDivide: $(\%, \%) \rightarrow (\%, \%)$ Polynomial division

monicDivide!: $(\%, \%) \rightarrow (\%, \%)$ Polynomial division

monicDivideBy: $\% \rightarrow \% \rightarrow (\%, \%)$ Polynomial division

monicDivideBy!: $\% \rightarrow \% \rightarrow (\%, \%)$ Polynomial division

monicQuotient: $(\%, \%) \rightarrow \%$ Quotient

monicQuotient!: $(\%, \%) \rightarrow \%$ Quotient

monicQuotientBy: $\% \rightarrow \% \rightarrow \%$ Quotient

monicQuotientBy!: $\% \rightarrow \% \rightarrow \%$ Quotient

monicRemainder: $(\%, \%) \rightarrow \%$ Remainder

monicRemainder!: $(\%, \%) \rightarrow \%$ Remainder

monicRemainderBy: $\% \rightarrow \% \rightarrow \%$ Remainder

monicRemainderBy!: $\% \rightarrow \% \rightarrow \%$ Remainder

if R has CommutativeRing and R has RittRing then

integrate: $\% \rightarrow \%$ Integration

$(\%, \text{Integer}) \rightarrow \%$

if R has FactorizationRing then

factor: $\% \rightarrow (R, \text{Product } \%)$ Factorisation into irreducibles

fractionalRoots: $\% \rightarrow \text{Generator FractionalRoot } R$ Roots in the fraction field

```

roots:  % → Generator FractionalRoot R  Roots in the coefficient ring

if R has Field then
  EuclideanDomain
  rationalReconstruction:  % → % → Partial Cross(%, %)  Rational reconstruction
  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
<i>p</i>	%	A polynomial
<i>q</i>	%	A polynomial
<i>a</i>	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: % \rightarrow 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: $(\%, R) \rightarrow (\%, R)$

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: % \rightarrow %
integrate: (% , Integer) \rightarrow %

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 $\int \dots \int s(x)dx^n$.

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 $a \rightarrow \text{monicXXX}(a, b)$, while monicXXXBy!(b) returns the map $a \rightarrow \text{monicXXX}!(a, b)$.

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: % \rightarrow 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: $(\%, \%) \rightarrow (\%, \%)$

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: $(\%, \%) \rightarrow \%$

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

rationalReconstruction m
 rationalReconstruction(m)(u)

Signature

rationalReconstruction: $\% \rightarrow \% \rightarrow \text{Partial Cross}(\%, \%)$

Parameter	Type	Description
m	$\%$	A modulus of positive degree
u	$\%$	A polynomial

Returns

rationalReconstruction(m)(u) returns either (a, b) such that $a/b = u \pmod{m}$, $\deg(a) \leq (\deg(m) - 1)/2$ and $\deg(b) \leq (\deg(m) - 1)/2$, or *failed* if no such a, b exist.

Remarks

The resulting a and b are guaranteed to be unique.

See also

rationalReconstruction

Usage

resultant(p, q)

Signature

resultant: (% , %) \rightarrow R

Parameter	Type	Description
p	%	A polynomial
q	%	A polynomial

Returns

Returns the resultant of p and q.

Usage

sparseMultiple(p, n)

Signature

sparseMultiple: $(\%, \text{Integer}) \rightarrow \%$

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) \rightarrow Generator R

Parameter	Type	Description
p	%	A polynomial
a	R	A scalar

Returns

Returns a generator generating the sequence $p(a), p(a + 1), p(a + 2), \dots$

Remarks

values uses arrays of differences and can be more efficient than repeated Horner evaluation.

Usage

factor p

Signaturefactor: % \rightarrow (R, Product %)

Parameter	Type	Description
p	%	A polynomial

ReturnsReturns $(c, p_1^{e_1} \cdots p_n^{e_n})$ 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: % \rightarrow 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 $[(r_1, e_1), \dots, (r_n, e_n)]$ 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:  %  $\rightarrow$  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: (%,%) \rightarrow List Cross(% , Integer)

Parameter	Type	Description
a, b	%	Nonzero polynomials

Returns

Return $[(p_1, e_1), \dots, (p_n, e_n)]$ 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 $u = \prod_{i=1}^n \prod_{j=0}^{e_i} p_i(x - j)$.

UnivariatePolynomialKaratsuba

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

UnivariatePolynomialRing R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariatePolynomialRing 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

UnivariateFreeRing R
add!: (% ,R,Z,% , Z, Z) \rightarrow % In-place partial product and sum
compose: (% , %) \rightarrow % Compose polynomials
translate: (% , R) \rightarrow % 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 threshold

Description

add!(p, c, m, q, n, N) computes all the terms of degree at least n and at most N of

$$p + cx^mq = \sum_{i=0}^{d+m} (a_i + cb_{i-m})x^i,$$

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 $p(x - r)$.

UnivariatePolynomialSquareFree

Usage

import from UnivariatePolynomialSquareFree(R, P)

Parameter	Type	Description
R	GcdDomain	Coefficient ring of the polynomials
P	UnivariatePolynomialAlgebra0 R	A polynomial ring

Description

UnivariatePolynomialSquareFree provides implementations of various squarefree factorization algorithms.

Exports

musser: $P \rightarrow (R, \text{Product } P)$ Musser's algorithm
yun: $P \rightarrow (R, \text{Product } P)$ 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

UnivariateTaylorSeriesType

Usage

UnivariateTaylorSeriesType R: Category

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

UnivariateTaylorSeriesType R is the category of univariate Taylor series with coefficients in R.

Exports

UnivariateFreeLinearArithmeticType R

degree: % \rightarrow Partial Integer upper bound on the degree
dot: Vector R \rightarrow Vector % \rightarrow % linear combination
finite?: % \rightarrow Boolean check whether the support is finite
series: Sequence R \rightarrow % creation of a series

where

UPC == UnivariatePolynomialAlgebra

if R has CommutativeRing then

differentiate: % \rightarrow % Differentiation
 (%, Integer) \rightarrow %
reciprocal: % \rightarrow Partial % Inverse

if R has CommutativeRing and R has RittRing then

integrate: % \rightarrow % Integration
 (%, Integer) \rightarrow %

if R has FloatType then

reciprocal: % \rightarrow Partial % Inverse

Usage

degree s
finite? s

Signatures

degree: % \rightarrow Partial Integer
finite?: % \rightarrow Boolean

Parameter	Type	Description
<i>s</i>	%	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: $\% \rightarrow \%$
 differentiate,integrate: $(\%, \text{Integer}) \rightarrow \%$

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 $\int \dots \int s(x)dx^n$.

Usage

series s

Signature

series: Sequence R \rightarrow %

Parameter	Type	Description
s	Sequence R	a coefficient sequence

Returns

Returns s viewed as a series.

Usage

import from UnivariateTaylorSeriesType2Poly(R, RXX, RX)

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain
RXX	UnivariateTaylorSeriesType R	A series type over R
RX	UnivariatePolynomialAlgebra R	A polynomial type over R

Description

UnivariateTaylorSeriesType2Poly(R, RXX, RX) provides conversion tools between polynomials and series.

Exports

expand: $R \rightarrow RX \rightarrow RXX$ series expansion at a point

truncate: $(RXX, \text{Integer}) \rightarrow RX$ truncation of a series

if R has Field then

expandFraction: $R \rightarrow \text{Fraction } RX \rightarrow (\text{Integer}, RXX)$ series expansion at a point

if R has FloatType then

expandFraction: $R \rightarrow \text{Fraction } RX \rightarrow (\text{Integer}, RXX)$ series expansion at a point

if R has CommutativeRing then

monicNewtonSeries: $RX \rightarrow RXX$ Newton series

tryExpandFraction: $R \rightarrow (RX, RX) \rightarrow (\text{Integer}, \text{Partial } RXX)$ try series expansion

if R has IntegralDomain then

polynomials: $(V \text{ } RXX, \text{Integer}) \rightarrow (V \text{ } RX, M \text{ } R)$ interpolation

$(V \text{ } RXX, \text{Integer}, \text{Integer}) \rightarrow (V \text{ } RX, M \text{ } R)$

where

V == Vector

M == DenseMatrix

Usage

expand a
expand(a)(p)

Signature

expand: R → RX → RXX

Parameter	Type	Description
<i>a</i>	R	the expansion point
<i>p</i>	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 $n \leq 0$ 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, tryExpandFraction
```

Usage

polynomials($[s_1, \dots, s_n]$, N)
 polynomials($[s_1, \dots, s_n]$, N, M)

Signatures

polynomials: (Vector RXX, Integer) \rightarrow (Vector RX, DenseMatrix R)
 polynomials: (Vector RXX, Integer, Integer) \rightarrow (Vector RX, DenseMatrix R)

Parameter	Type	Description
s_i	RXX	series
N	Integer	a degree bound
M	Integer	an upper bound

Description

polynomials($[s_1, \dots, s_n]$, N , M) returns $([p_1, \dots, p_s], A)$ such that the series $[s_1, \dots, s_n]A$ all have coefficients 0 from x^{N+1} to x^M , and $[p_1, \dots, p_s]$ are the truncations to order N of $[s_1, \dots, s_n]A$. If the upper bound M is not given, then the series returned have coefficients 0 from x^{N+1} up to an order that is determined heuristically.

Usage

monicNewtonSeries p

Signature

monicNewtonSeries: $RX \rightarrow RXX$

Parameter	Type	Description
p	RX	A monic polynomial

Returns

Returns the series

$$\sum_{n \geq 0} (y_1 + \dots + y_d)^n x^n$$

where y_1, \dots, y_d are all the roots of p .

Usage

```
tryExpandFraction a
tryExpandFraction(a)(p,q)
```

Signature

```
tryExpandFraction:  R → (RX, RX) → (Integer, Partial RXX)
```

Parameter	Type	Description
a	R	the expansion point
p, q	RX	numerator and denominator of a rational function

Returns

If $q(a)$ is a unit in R , then `tryExpandFraction(a)(p,q)` returns (n, s) where $n \leq 0$ and the series expansion of p/q around a is $(x-a)^n s(x-a)$. In addition, $s(a) \neq 0$ whenever $n < 0$. Otherwise, it returns (n, failed) where n is the order of p/q at $x = a$.

See also

```
expand, expandFraction
```

Usage

truncate(s, m)

Signature

truncate: (RXX,Integer) → RX

Parameter	Type	Description
<i>s</i>	%	a series
<i>m</i>	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 ArithmeticType	The coefficient domain
Rx	UnivariateTaylorSeriesType R	Series over R
RxY	UnivariatePolynomialAlgebra Rx	Polynomials over Rx
RX	UnivariatePolynomialAlgebra R	Polynomials over R
RXY	UnivariatePolynomialAlgebra 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:  $RxY \rightarrow RxY$       Differentiation
  root:          $(RXY, R) \rightarrow Rx$     Root of a polynomial
                $(RxY, R) \rightarrow Rx$ 

if  $R$  has FloatType then
  root:  $(RXY, RXY, R) \rightarrow Rx$     Root of a polynomial
        $(RxY, RxY, R) \rightarrow Rx$ 
```

Usage

differentiate p

Signature

differentiate: RxY \rightarrow 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 BivariateUtilitiesPackage (U,V)
```

Parameter	Type	Description
U	UnivariatePolynomialAlgebra Integer	
V	UnivariatePolynomialAlgebra U	

Description

BivariateUtilitiesPackage (U,V) provides basic operations for bivariate polynomials over the integer numbers. In particular, it provides support for modular methods. In the above description, the smallest variable of a polynomial from V refers to the variable of U and its degree refers to the its degree as a univariate polynomial w.r.t. the variable of V.

Exports

maxNorm:	$U \rightarrow Z$	maximum absolute value of a coefficient
maxNorm:	$V \rightarrow Z$	maximum absolute value of a coefficient
degree_U:	$V \rightarrow Z$	degree w.r.t. the smallest variable
resultantCoefficientBound:	$(V, V) \rightarrow Z$	upper bound for the resultant maxNorm
resultantDegreeBound:	$(V, V) \rightarrow Z$	degree bound for the resultant
primeBad?:	$(V, V, I) \rightarrow B$	true iff the degree of one of the two polynomials drops modulo the number

where

```
I == MachineInteger
Z == Integer
B == Boolean
```

DirectProduct

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)

DirectProductCategory

Usage

DirectProductCategory (dim, T): Category

Parameter	Type	Description
<i>dim</i>	MachineInteger	The length of a direct product
<i>T</i>	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 (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

DistributedMultivariatePolynomial0

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 can be multiplied only by a constant from R . 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)

ExponentCategory

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 additive ordered monoid (with cancellation) by associating to every product of variables its sequence of degrees.

Exports

GeneralExponentCategory

exponent:	$V \rightarrow \%$	The exponent of a variable
exponent:	$(V, Z) \rightarrow \%$	The exponent of a power of a variable
exponent:	Generator Cross $(V, Z) \rightarrow \%$	The exponent of a power product
exponent:	$(\text{List } V, \text{List } Z) \rightarrow \%$	The exponent of a power product
terms:	$\% \rightarrow \text{Generator Cross } (V, Z)$	Inverse map of exponent
mainVariable:	$\% \rightarrow \text{Partial } V$	The biggest variable, if any
variables:	$\% \rightarrow \text{List } V$	The list of variables of a monomial
degree:	$(\%, V) \rightarrow Z$	The degree w.r.t. a variable
totalDegree:	$\% \rightarrow Z$	The sum of the degrees of an exponent
totalDegree:	$(\%, \text{List } V) \rightarrow Z$	The sum of the degrees w.r.t. a list
gcd:	$(\%, \%) \rightarrow \%$	Monomial gcd
lcm:	$(\%, \%) \rightarrow \%$	Monomial lcm
syzygy:	$(\%, \%) \rightarrow (\%, \%)$	Cofactors w.r.t. lcm

where

$Z == \text{Integer}$

Usage

FiniteAbelianMonoidRing0 (R,V,E): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	VariableType	The variable type
E	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 $c_1 x + \mathbf{term}(c_2, e)y$ 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

FiniteVariableType

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

<code>variable:</code>	<code>MachineInteger → %</code>	Retuns the n -th variable of the type, if any
<code>index:</code>	<code>% → MachineInteger</code>	Retuns the associated machine integer
<code>#:</code>	<code>MachineInteger</code>	The number of elements of the type
<code>max:</code>	<code>%</code>	The greatest element of the type
<code>min:</code>	<code>%</code>	The smallest element of the type
<code>minToMax:</code>	<code>List %</code>	The elements of the type sorted in increasing order
<code>maxToMin:</code>	<code>List %</code>	The elements of the type sorted in decreasing order

Usage

variable *i*

Signature

variable: MachineInteger \rightarrow %

Parameter	Type	Description
<i>i</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 $a \geq b$ implies $a + c \geq b + c$. 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 $a \geq b$ and *failed* otherwise. For multivariate monomials $f(a, b)$ is c if $a = bc$ and *failed* if 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
<i>n</i>	Integer	An integer
<i>x</i>	%	An element of the type

Returns

Returns the product *nx*.

IntegerExponentVectorCategory

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 non-negative integers.

Exports

CopyableType		
ExponentCategory V		
HashType		
free!:	$\% \rightarrow ()$	Asserts that the input will no longer be used
exponent:	$\text{Tuple Integer} \rightarrow \%$	Creation from a tuple
exponent:	$\text{PrimitiveArray Integer} \rightarrow \%$	Creation from a primitive array
		Degrees start at slot 1
		Slot 0 must be the total degree
parray:	$\% \rightarrow \text{PrimitiveArray Integer}$	Inverse mapping of <code>exponent</code>

IntegerPolynomial

Usage

import from IntegerPolynomial

Description

IntegerPolynomial provides a basic domain for multivariate polynomials with **Integer** coefficients and variables from **OrderedSymbol**. The representation is sparse and recursive by means of the **SparseUnivariatePolynomial** univariate polynomial domain constructor.

Exports

RecursiveMultivariatePolynomialCategory0(Z,OS)

coerce: String \rightarrow % Conversion.

univariate: % \rightarrow SUP(%) Convert to univariate w.r.t. the greatest variable

multivariate: (SUP(%), OS) \rightarrow % Convert to multivariate with given variable

where

OS == OrderedSymbol

Z == Integer

SUP == SparseUnivariatePolynomial

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 primitive 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 primitive 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 non-negative 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

MachineIntegerLexicographicalExponent

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 primitive 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` \rightarrow % Conversion to an element of the type.

OrderedVariableList

Usage

import from OrderedVariableList t

Parameter	Type	Description
<i>t</i>	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 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

OrderedVariableTuple

Usage

import from OrderedVariableTuple t

Parameter	Type	Description
<i>t</i>	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

PolynomialRing

Usage

PolynomialRing (R,V): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	TotallyOrderedType ExpressionType	The variable domain

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)

eval: $(\%, V, R) \rightarrow \%$ eval(x, v, r) evaluates x at $v = r$

eval: $(\%, V, \%) \rightarrow \%$ eval(x, v, y) evaluates x at $v = y$

where

GEN == Generator

PZ == Cross($\%$, Integer)

if R has CommutativeRing then

differentiate: $(\%, V) \rightarrow \%$ Differentiation w.r.t. a variable

differentiate: $(\%, V, Integer) \rightarrow \%$ Iterated differentiation w.r.t. a variable

if R has CommutativeRing then

CommutativeRing

if R has IntegralDomain then

IntegralDomain

if R has GcdDomain then

GcdDomain

Usage

PolynomialRing0 (R,V): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	TotallyOrderedType ExpressionType	The variable domain

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 $\{v_1, \dots, v_l\}$ this polynomial ring is just $R[v_1, \dots, v_l]$. 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 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 set of algebraically independent 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

PolynomialTypeRing(R, V)
FreeAlgebra R

Usage

PolynomialTypeRing (R,V): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	ExpressionType	The variable domain

Description

PolynomialTypeRing (R,V) is the category of standard filtered rings generated over R by power products of the form $v_1^{e_1} \cdots v_n^{e_n}$ for $v_i \in V$ and $e_i \in \mathbb{N}$. No commutativity is assumed either between the variables in V, or between R and V.

Exports

```
StandardFilteredRing(R, V)
coefficient:      (% , V, Z) → %           Coefficient w.r.t. a power  $v^n$ .
coefficient:      (% , L V, L Z) → %       Coefficient w.r.t. a power product
degree:           (% , V) → Z              Degree w.r.t. a variable
degrees:          % → GEN VZ              Degrees w.r.t. all variables
leadingCoefficient: (% , V) → %           Leading coefficient w.r.t. a variable
multivariate:     (P:IFRR R) → (P, V) → % Conversion from univariate view
                  (P:IFRR P) → (P, V) → %
reductum:         (% , V) → %             Reductum w.r.t. a variable
univariate:       (P:IFRR P) → (% , V) → P Conversion to univariate view
```

If V has TotallyOrderedType then

```
initial:          % → %                   Leading coefficient w.r.t. main variable
univariate:       (P:IFRR P) → % → P     Conversion to univariate view
```

where

```
GEN == Generator
IFRR == IndexedFreeRRing
L == List
Z == Integer
VZ == Cross (V, Integer)
```

Usage

```
import from RecursiveMultivariatePolynomial0 (UP,R,V)
```

Parameter	Type	Description
UP	$(T: \text{Join}(\text{ArithmeticType}, \text{ExpressionType})) \rightarrow \text{POL } T$	Univariate functor
R	Ring	The coefficient ring
V	VariableType	The variables

where

```
POL == UnivariatePolynomialAlgebra
```

Description

RecursiveMultivariatePolynomial0 (UP,R,V) provides a basic domain for multivariate polynomials with coefficients in R and variables in V . The representation is recursive by means of UP .

Exports

```
RecursiveMultivariatePolynomialCategory0(R,V)
```

```
univariate:  % → UP %          Convert to univariate w.r.t. the greatest variable
```

```
multivariate: (UP(%), V) → %    Convert to multivariate with given variable
```

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 recursive vision of a multivariate polynomial (as a univariate polynomial w.r.t. its main variable).

Exports

`PolynomialRing(R,V)`

<code>variables:</code>	<code>% → SortedSetV</code>	The sorted set of variables of a polynomial
<code>mvar:</code>	<code>% → V</code>	Greatest variable of a polynomial
<code>mdeg:</code>	<code>% → Integer</code>	Degree w.r.t. greatest variable
<code>rank:</code>	<code>% → %</code>	Leading monomial as univariate w.r.t. greatest variable
<code>init:</code>	<code>% → %</code>	Leading coefficient as univariate w.r.t. greatest variable
<code>tail:</code>	<code>% → %</code>	Reductum as univariate w.r.t. greatest variable
<code>head:</code>	<code>% → %</code>	Leading term as univariate w.r.t. greatest variable

Usage

import from ResultantOfBivariatePolynomialsOverSmallPrimeField (Kp, Up, Vp)

Parameter	Type	Description
Kp	SmallPrimeFieldCategory	
Up	UnivariatePolynomialAlgebra Kp	
Vp	UnivariatePolynomialAlgebra Up	

Description

ResultantOfBivariatePolynomialsOverSmallPrimeField (Kp, Up, Vp) provides resultant computations for two bivariate polynomials over a small prime field.

Exports

evaluationResultant:	$(Vp, Vp, Z) \rightarrow Up$	resultant computed by an evaluation and interpolation scheme. Arg 3 is a degree bound.
evaluationReduction:	$(Vp, Kp) \rightarrow Up$	evaluation at a point
interpolation:	$(A\ Kp, A\ Kp) \rightarrow Up$	Lagrange interpolant where the arrays give points and values resp.
integerImage:	$Up \rightarrow DUP\ I$	conversion.

where

DUP	==	DenseUnivariatePolynomial
I	==	MachineInteger
Z	==	Integer
A	==	Array

SparseIntegerMultivariatePolynomial

Usage

import from SparseIntegerMultivariatePolynomial (V)

Parameter	Type	Description
V	VariableType	The variables

Description

SparseIntegerMultivariatePolynomial (V) provides a basic domain for multivariate polynomials with coefficients over the `Integer` ring and variables in V . The representation is sparse and recursive by means of *SUP*.

Exports

RecursiveMultivariatePolynomialCategory0(Z,V)

univariate: % \rightarrow SUP(%) Convert to univariate w.r.t. the greatest variable

multivariate: (SUP(%), V) \rightarrow % Convert to multivariate with given variable

where

Z == Integer

SUP == SparseUnivariatePolynomial

SparseMultivariatePolynomial

Usage

import from SparseMultivariatePolynomial (R,V)

Parameter	Type	Description
R	Ring	The coefficient ring
V	VariableType	The variables

Description

SparseMultivariatePolynomial (R,V) provides a basic domain for multivariate polynomials with coefficients in R and variables in V . The representation is sparse and recursive by means of *SUP*.

Exports

RecursiveMultivariatePolynomialCategory0(R,V)

univariate: % \rightarrow SUP(%) Convert to univariate w.r.t. the greatest variable

multivariate: (SUP(%), V) \rightarrow % Convert to multivariate with given variable

where

SUP == SparseUnivariatePolynomial

Usage

StandardFilteredRing (R,V): Category

Parameter	Type	Description
R	ArithmeticType ExpressionType	The coefficient domain
V	ExpressionType	The variable domain

Description

StandardFilteredRing (R,V) is the category of R-rings with a standard filtration for which R is the ring of elements of degree 0, and V generates the elements of degree 1. It is the set of finite linear combinations of the form $\sum r_i w_i$ where the r_i are in R and the w_i are words of V.

Exports

CopyableType

FreeRRing R

coerce:	$V \rightarrow \%$	Conversion of a variable to a polynomial
ground?:	$\% \rightarrow \text{Boolean}$	Membership test to the coefficient ring
totalDegree:	$\% \rightarrow \mathbb{Z}$	Greatest total degree among all the monomials
term:	$(R, V, \mathbb{Z}) \rightarrow \%$	term (r, v, n) returns $r v^n$ provided $n \geq 0$
term:	$(R, \text{GEN } VZ) \rightarrow \%$	Term from a power product and a coeff.
term:	$(R, \text{L } V, \text{L } \mathbb{Z}) \rightarrow \%$	Term from a power product and a coeff.
times:	$(\%, R, V, \mathbb{Z}) \rightarrow \%$	times (x, r, v, n) is $x (r v^n)$
times!:	$(\%, R, V, \mathbb{Z}) \rightarrow \%$	times (x, r, v, n) is $x (r v^n)$ and may modify x
univariate?:	$\% \rightarrow \text{Boolean}$	Check whether elt is univariate
variable:	$\% \rightarrow V$	Conversion of a polynomial to a variable
variable?:	$\% \rightarrow \text{Boolean}$	Membership test to the variable domain
variableProduct:	$\% \rightarrow \text{GEN } VZ$	variableProduct x is g s.t. term (1, g) is x
variables:	$\% \rightarrow \text{GEN } V$	The variables occuring in a polynomial

where

GEN == Generator

L == List

Z == Integer

VZ == Cross (V, Integer)

if R has Parsable and V has Parsable then

Parsable

if V has TotallyOrderedType then

mainVariable: $\% \rightarrow V$ Greatest variable occuring

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 \rightarrow Partial % Associated variable, if any
symbol: % \rightarrow 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

infix: (TEXT, %) → TEXT

Conversion to one-dim infix output

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 L^AT_EX

texParen?: (MachineInteger, %) → Boolean

Check whether to parenthesize

where

TEXT == TextWriter

OP == ExpressionTreeOperator

Usage*format*(p, t)**Signature**

aldor,axiom,C,fortran,infix,lisp,maple,tex: (TextWriter, %) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>t</i>	%	An expression tree

Description

Writes to *p* the expression corresponding to the tree *t* in the requested format.

Usage

apply(op, [t₁, ..., t_n])
op [t₁, ..., t_n]

Signature

apply: (ExpressionTreeOperator, List %) → %

Parameter	Type	Description
<i>op</i>	ExpressionTreeOperator	An operator
<i>t_i</i>	%	Expression trees

Returns

Returns the tree whose root is *op*, with arguments *t*₁, ..., *t*_{*n*}.

Usage

arguments *t*

Signature

arguments: % \rightarrow List %

Parameter	Type	Description
<i>t</i>	%	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 \rightarrow %

Parameter	Type	Description
<i>a</i>	ExpressionTreeLeaf	A leaf

Returns

extree a returns *a* as an expression tree.

Usage

is?(t, op)

Signature

is?: (% , ExpressionTreeOperator) → Boolean

Parameter	Type	Description
<i>t</i>	%	An expression tree
<i>op</i>	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
<i>t</i>	%	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: % \rightarrow %

Parameter	Type	Description
<i>t</i>	%	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?: % \rightarrow Boolean

Parameter	Type	Description
<i>t</i>	%	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: % \rightarrow ExpressionTreeOperator

Parameter	Type	Description
<i>t</i>	%	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
<i>prec</i>	MachineInteger	An operator precedence.
<i>t</i>	%	An expression tree

Returns

Returns *true* if *t* should be parenthetized when appearing as argument of an operator of precedence *prec*, *false* otherwise.

ExpressionTreeAnd

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**.

ExpressionTreeAssign

Usage

import from ExpressionTreeAssign

Description

ExpressionTreeAssign is 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

ExpressionTreeCase

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**.

ExpressionTreeComplex

Usage

import from ExpressionTreeComplex

Description

ExpressionTreeComplex is the complex operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeEqual

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**.

ExpressionTreeExpt

Usage

import from ExpressionTreeExpt

Description

ExpressionTreeExpt is the exponentiation operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeFactorial

Usage

import from ExpressionTreeFactorial

Description

ExpressionTreeFactorial is the generalized factorial operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeGreaterEqual

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**.

ExpressionTreeGreaterThan

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**.

ExpressionTreeIf

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
infix:	(TEXT, %) → TEXT	Conversion to one-dim infix output
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 L ^A T _E X
texParen?:	% → Boolean	Check whether to parenthesize

where

```
TEXT == TextWriter
```

Usage

format(p, a)

Signature

aldor,axiom,C,fortran,infix,lisp,maple,tex: (TextWriter, %) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	Boolean DoubleFloat Float Integer MachineInteger SingleFloat String Symbol	A constant

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: % \rightarrow %

Parameter	Type	Description
<i>a</i>	%	A leaf

Returns

Returns the leaf $-a$ if *a* is a numerical leaf, *a* otherwise.

See also

negative?

Usage

negative? a

Signature

negative?: % \rightarrow Boolean

Parameter	Type	Description
<i>a</i>	%	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
<i>a</i>	%	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
<i>a</i>	%	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: % \rightarrow **String**
string?: % \rightarrow **Boolean**

Parameter	Type	Description
<i>a</i>	%	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
<i>a</i>	%	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?: % \rightarrow Boolean

Parameter	Type	Description
<i>a</i>	%	A leaf

Returns

Returns *true* if the leaf *a* should be parenthetized, *false* otherwise.

ExpressionTreeLessEqual

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**.

ExpressionTreeLessThan

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**.

ExpressionTreeLispList

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**.

ExpressionTreeList

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**.

ExpressionTreeMatrix

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**.

ExpressionTreeMinus

Usage

import from ExpressionTreeMinus

Description

ExpressionTreeMinus is the unary/binary minus operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeNotEqual

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**.

ExpressionTreeOperator

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
infix:	(TEXT, List TREE) → TEXT	Conversion to one-dim infix output
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 L ^A T _E X
texParen?:	MachineInteger → Boolean	Check whether to parenthesize
uniqueId:	MachineInteger	A unique key per operator

where

TEXT == TextWriter
TREE == ExpressionTree

Usage

format(*p*, [*t*₁, ..., *t*_{*n*}])

Signature

aldor,axiom,C,fortran,infix,lisp,maple,tex: (TextWriter, List ExpressionTree) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>t</i> _{<i>i</i>}	ExpressionTree	The arguments of the operator

Description

Writes to *p* the expression corresponding to this operator applied to the arguments (*t*₁, ..., *t*_{*n*}) in the requested format.

Usage

arity

Signaturearity: \rightarrow 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: \rightarrow Symbol

Returns

Returns the name of this operator.

Usage

texParen? prec

Signature

texParen?: MachineInteger \rightarrow Boolean

Parameter	Type	Description
<i>prec</i>	MachineInteger	An operator precedence.

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: \rightarrow MachineInteger

Returns

Returns a integer key which is associated to this operator only. This is used for testing whether two operators are equal.

ExpressionTreeOperatorTools

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, [t1, ..., tn], paren?, farg, op, left, right)`

Signatures

`infix: (TEXT, List TREE, TREE → Boolean, (TEXT, TREE) → TEXT, String, String, String) → TEXT`

Parameter	Type	Description
<i>p</i>	TEXT	The port to write to
<i>[t₁, ..., t_n]</i>	List TREE	The arguments of the operator
<i>paren?</i>	TREE → Boolean	The parenthetization function
<i>farg</i>	(TEXT, TREE) → TEXT	The function for the arguments
<i>op</i>	String	The infix symbol
<i>left</i>	String	The left parenthesis (optional)
<i>right</i>	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, [t1, ..., tn])`

Signature

`lisp: (TEXT, String, List TREE) → TEXT`

Parameter	Type	Description
<i>p</i>	TEXT	The port to write to
<i>s</i>	String	A Lisp operator name
[<i>t</i> ₁ , ..., <i>t</i> _{<i>n</i>}]	List TREE	The arguments of the operator

Description

Writes (*st*₁ ... *t*_{*n*}) to *p*, where each *t*_{*i*} is written in Lisp format.

Usage

```
prefix(p, t, paren?, farg, op, left, right)
prefix(p, [t1, ..., tn], 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
<i>p</i>	TEXT	The port to write to
<i>[t₁, ..., t_n]</i>	List TREE	The arguments of the operator
<i>paren?</i>	TREE → Boolean	The parenthetization function
<i>farg</i>	(TEXT, TREE) → TEXT	The function for the arguments
<i>op</i>	String	The prefix symbol
<i>left</i>	String	The left parenthesis (optional)
<i>right</i>	String	The right parenthesis (optional)

where

```
TEXT == TextWriter
TREE == ExpressionTree
```

Description

Writes $op(farg(t_1), \dots, farg(t_n))$ 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`

ExpressionTreePlus

Usage

import from ExpressionTreePlus

Description

ExpressionTreePlus is the addition operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreePrefix

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.

ExpressionTreeQuotient

Usage

import from ExpressionTreeQuotient

Description

ExpressionTreeQuotient is the division operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeSubscript

Usage

import from ExpressionTreeSubscript

Description

ExpressionTreeSubscript is the subscript operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeTimes

Usage

import from ExpressionTreeTimes

Description

ExpressionTreeTimes is the multiplication operator for expression trees.

Exports

ExpressionTreeOperator

ExpressionTreeVector

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**.

Evaluator

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!:	$(\text{TEXT}, \text{TREE}, \text{SYM } R) \rightarrow \overline{R}$	Interpret an arbitrary tree
evalLeaf!:	$(\text{LEAF}, \text{SYM } R) \rightarrow \overline{R}$	Interpret a leaf
evalOp!:	$(\text{TEXT}, \text{TREE}, \text{SYM } R) \rightarrow \overline{R}$	Interpret $op(args)$
evalPrefix!:	$(\text{String}, Z, \text{List } R, \text{SYM } R) \rightarrow \overline{R}$	Interpret $op(args)$, op is prefix

where

Z	==	MachineInteger
LEAF	==	ExpressionTreeLeaf
\overline{R}	==	Partial R
SYM	==	SymbolTable
TEXT	==	TextWriter
TREE	==	ExpressionTree

InfixExpressionParser

Usage

import from InfixExpressionParser

Description

InfixExpressionParser implements infix expression parsers.

Exports

ParserReader

precedences!: ($\%$, MachineInteger, MachineInteger) \rightarrow $\%$ Set operator precedences

Usage

precedences!(p, op, n)

Signature

precedences!: (% , MachineInteger, MachineInteger) → %

Parameter	Type	Description
<i>p</i>	%	A parser
<i>op</i>	MachineInteger	An operator code
<i>n</i>	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`.

LispExpressionParser

Usage

import from LispExpressionParser

Description

LispExpressionParser implements lisp expression parsers.

Exports

ParserReader

MakePartialRing

Usage

import from MakePartialRing(R, f)

Parameter	Type	Description
R	Ring	A ring
f	$R \rightarrow \text{Partial Integer}$	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: $R \rightarrow \%$ Convert an element of R to one of the partial ring

coerce: $\% \rightarrow R$ 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:	% → <code>TextWriter</code>	Input stream of the maple session
maple:	() → %	Create a maple session
run:	% → <code>Partial ExpressionTree</code>	Run a maple session

Usage

input m

Signature

input: % \rightarrow 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: % \rightarrow 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 5th 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)))
```

Parsable

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,[e1,...,en])
```

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,[e₁,...,e_n]) returns the result of evaluating $op(e_1, \dots, e_n)$ 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

<code>eof?:</code>	<code>% → Boolean</code>	Check for end of input
<code>lastError:</code>	<code>% → MachineInteger</code>	Code for last parsing error
<code>parse!:</code>	<code>% → Partial ExpressionTree</code>	Parse one expression

Usage

eof? p

Signature

eof?: % \rightarrow Boolean

Parameter	Type	Description
<i>p</i>	%	A parser

Returns

Returns *true* if the input is finished, *false* otherwise.

Usage

lastError p

Signature

lastError: % \rightarrow MachineInteger

Parameter	Type	Description
<i>p</i>	%	A parser

Returns

Returns the code for the last parsing error.

Usage

parse! p

Signature

parse!: % \rightarrow 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 \rightarrow % Create a parser

Usage

parser r

Signature

parser: TextReader → %

Parameter	Type	Description
<i>r</i>	TextReader	The input stream to parse

Returns

Returns a parser that takes its input on r.

ParsingTools

Usage

import from ParsingTools R

Parameter	Type	Description
<i>R</i>	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, \dots, e_n])`

Signature

`evalArith: (MachineInteger, List ExpressionTree) \rightarrow Partial R`

Parameter	Type	Description
op	MachineInteger	Code for an operator
e_i	ExpressionTree	Expression trees

Returns

Returns the result of evaluating $op(e_1, \dots, e_n)$ where op is a code from `include/algebrauid.as`. Provides support for the evaluation of the operators $+$, $-$, $*$ and \wedge , as well as $/$ when R has `CommutativeRing` or `FloatType`.

Usage

evalInt e

Signature

evalInt: ExpressionTree \rightarrow 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.

PartialRing

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

Scanner

Usage

import from Scanner

Description

Scanner provides a simple scanner for mathematical expressions.

Exports

scan: TextReader \rightarrow Token Scan a token

Usage

scan *p*

Signature

scan: `TextReader` \rightarrow `Token`

Parameter	Type	Description
<i>p</i>	<code>TextReader</code>	Text to scan

Returns

Returns the next token read from the reader *p*.

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

Usage

center(*p*, *s*)

Signature

center: (TextWriter, String) → TextWriter

Parameter	Type	Description
<i>p</i>	TextWriter	The port to write to
<i>s</i>	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) \rightarrow 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
<i>verbose?</i>	Boolean	Select verbose or quiet mode
<i>time?</i>	Boolean	Select whether to return times in msec
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 $n = b_1b_0$ where b_0 is 1 if a syntax error occurred, 0 otherwise, and b_1 is 1 if any other error occurred, 0 otherwise.

Token

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) \rightarrow %

Parameter	Type	Description
$[d_0, \dots, d_n]$	List Character	A list of digits
$[e_0, \dots, e_m]$	List Character	A list of digits

Returns

float($[d_0, \dots, d_n], [e_0, \dots, e_m]$) returns the float $d_n d_{n-1} \dots d_0 . e_m e_{m-1} \dots e_0$ as a token.

See also

integer

Usage

integer l

Signature

integer: List Character \rightarrow %

Parameter	Type	Description
$[d_0, \dots, d_n]$	List Character	A list of digits

Returns

integer($[d_0, \dots, d_n]$) returns the integer $d_0 + 10d_1 + \dots + 10^nd_n$ as a token.

See also

float

Usage

leaf t
leaf? t

Signatures

leaf: % → ExpressionTreeLeaf
leaf?: % → Boolean

Parameter	Type	Description
<i>t</i>	%	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

Token	name
-------	------

Usage
 name l

Signature
 name: List Character \rightarrow %

Parameter	Type	Description
$[c_0, \dots, c_n]$	List Character	A list of characters

Returns
 name($[c_0, \dots, c_n]$) returns the name $c_n c_{n-1} \dots c_0$ as a token representing a constant symbol.

See also
 prefix, string

Usage

operator t
operator? t

Signatures

operator: % → ExpressionTreeOperator
operator?: % → Boolean

Parameter	Type	Description
<i>t</i>	%	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 \rightarrow %

Parameter	Type	Description
$[c_0, \dots, c_n]$	List Character	A list of characters

Returns

prefix($[c_0, \dots, c_n]$) returns the name “ $c_n c_{n-1} \dots c_0''$ ” as a token representing a prefix function.

See also

name

Usage

special t
special? t

Signatures

special: % → MachineInteger
special?: % → Boolean

Parameter	Type	Description
<i>t</i>	%	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

string l

Signature

string: List Character \rightarrow %

Parameter	Type	Description
$[c_0, \dots, c_n]$	List Character	A list of characters

Returns

string($[c_0, \dots, c_n]$) returns the string “ $c_n c_{n-1} \dots c_0''$ ” 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
<i>a</i>	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.

AlgebraLibraryInformation

Usage

import from AlgebraLibraryInformation

Description

AlgebraLibraryInformation provides version information about the **Algebra** library.

Exports

VersionInformationType

IndexedVariable

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:    %  $\rightarrow$  S  Index of a variable
variable: S  $\rightarrow$  %  Create an indexed variable
```

Usage

index *v*

Signature

index: % \rightarrow S

Parameter	Type	Description
<i>v</i>	%	An indexed variable

Returns

Returns the index of *v*.

Usage

variable s

Signature

variable: S → %

Parameter	Type	Description
<i>s</i>	S	An index

Returns

Returns the variable x_s .

Permutation

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:	$(\%, Z) \rightarrow Z$	Image of an element
mapping:	$\% \rightarrow (Z \rightarrow Z)$	Action of a permutation
sign:	$\% \rightarrow Z$	Sign
transpose:	$(Z, Z) \rightarrow \%$	Transposition
transpose!:	$(\%, Z, Z) \rightarrow \%$	Compose with a transposition

where

$Z == \text{MachineInteger}$

Usage

apply(σ , x)
 σx

Signature

apply: ($\%$, MachineInteger) \rightarrow MachineInteger

Parameter	Type	Description
σ	$\%$	A permutation
x	MachineInteger	An index

Returns

Returns σx .

Usage

mapping σ

Signature

mapping: $\% \rightarrow (\text{MachineInteger} \rightarrow \text{MachineInteger})$

Parameter	Type	Description
σ	$\%$	A permutation

Returns

Returns the map corresponding to σ .

See also

apply

Usage

sign σ

Signature

sign: $\% \rightarrow \text{MachineInteger}$

Parameter	Type	Description
σ	$\%$	A permutation

Returns

Returns the sign of σ , *i.e.* $(-1)^\epsilon$ where ϵ is the number of transpositions in the factorization of σ .

Usage

```
transpose(x,y)
transpose!( $\sigma$ ,x,y)
```

Signatures

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

Parameter	Type	Description
σ	%	A permutation
x,y	MachineInteger	Indices

Returns

transpose(x,y) returns the transposition (xy) , while transpose!(σ ,x,y) returns the composition $(xy) \circ \sigma$.

Remarks

When using transpose!, the storage used by σ is allowed to be destroyed or reused, so do not use it unless σ 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. Since there is no guarantee of reuse, you should always use the permutation returned by transpose! rather than σ after the call.

Sequence

Usage

import from Sequence R

Parameter	Type	Description
R	ExpressionType ArithmeticType	The coefficient domain

Description

Sequence R implements infinite sequences with coefficients in R.

Exports

LinearStructureType R

UnivariateFreeLinearArithmeticType R

#:	$\% \rightarrow \mathbb{Z}$	number of computed elements
bound:	$\% \rightarrow \text{MachineInteger}$	upper bound on the support size
dot:	$\text{Vector } R \rightarrow \text{Vector } \% \rightarrow \%$	linear combination
finite?:	$\% \rightarrow \text{Boolean}$	check whether the support is finite
interlacing:	$\text{List } \% \rightarrow \%$	interlacing
sequence:	$\text{Stream } R \rightarrow \%$	create a sequence

if R has Ring then

random: $() \rightarrow \%$ random sequence

Usage

s

Signatures

#: % → MachineInteger

Parameter	Type	Description
<i>s</i>	%	a sequence

Returns

Returns the number of elements of *s* that have been computed.

Usage

bound s

finite? s

Signaturesbound: % \rightarrow MachineIntegerfinite?: % \rightarrow Boolean

Parameter	Type	Description
<i>s</i>	%	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

```
dot [r1, ..., rn]
dot([r1, ..., rn])([s1, ..., sn])
```

Signature

```
dot: Vector R → Vector % → %
```

Parameter	Type	Description
r_i	R	coefficients
s_i	%	sequences

Returns

dot(r)(s) returns the sequence $\sum_{i=1}^n r_i s_i$, while dot(r) returns the map $[s_1, \dots, s_n] \rightarrow \text{sum}_{i=1}^n r_i s_i$.

Remarks

Using dot is more efficient than building up the same linear combination via additions and multiplications, since the intermediate combinations are not storing their computed elements when using dot.

Usage

interlacing $[s_1, \dots, s_n]$

Signature

interlacing: List % \rightarrow %

Parameter	Type	Description
s_i	%	sequences

Description

Given $s_i = r_{i1}, r_{i2}, \dots$, returns the sequence $r_{11}, r_{21}, \dots, r_{n1}, r_{12}, r_{22}, \dots, r_{n2}, \dots$

Usage

random()

Signature

random: $() \rightarrow \%$

Returns

Returns a sequence with random entries.

Usage

sequence s

Signature

sequence: Stream R \rightarrow %

Parameter	Type	Description
s	Stream R	a stream

Returns

Returns s viewed as a sequence.

