axiom™

The 30 Year Horizon

Manuel Bronstein *William Burge* *Timothy Daly*
James Davenport *Michael Dewar* *Martin Dunstan*
Albrecht Fortenbacher *Patrizia Gianni* *Johannes Grabmeier*
Jocelyn Guidry *Richard Jenks* *Larry Lambe*
Michael Monagan *Scott Morrison* *William Sit*
Jonathan Steinbach *Robert Sutor* *Barry Trager*
Stephen Watt *Jim Wen* *Clifton Williamson*

Volume 15: The Sane Compiler

July 9, 2020

6d805049a0a4f2fb41b0cee798ebf040181ab1d8

```
Portions Copyright (c) 2005 Timothy Daly

The Blue Bayou image Copyright (c) 2004 Jocelyn Guidry

Portions Copyright (c) 2004 Martin Dunstan
Portions Copyright (c) 2007 Alfredo Portes
Portions Copyright (c) 2007 Arthur Ralfs
Portions Copyright (c) 2005 Timothy Daly

Portions Copyright (c) 1991-2002,
The Numerical ALgorithms Group Ltd.
All rights reserved.

This book and the Axiom software is licensed as follows:

Redistribution and use in source and binary forms, with or
without modification, are permitted provided that the following
conditions are
met:

    - Redistributions of source code must retain the above
      copyright notice, this list of conditions and the
      following disclaimer.

    - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the
      following disclaimer in the documentation and/or other
      materials provided with the distribution.

    - Neither the name of The Numerical ALgorithms Group Ltd.
      nor the names of its contributors may be used to endorse
      or promote products derived from this software without
      specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.

Michael Albaugh	Cyril Alberga	Roy Adler
Christian Aistleitner	Richard Anderson	George Andrews
Jerry Archibald	S.J. Atkins	Jeremy Avigad
Knut Bahr	Henry Baker	Martin Baker
Stephen Balzac	Yurij Baransky	David R. Barton
Thomas Baruchel	Gerald Baumgartner	Gilbert Baumslag
Michael Becker	Nelson H. F. Beebe	Jay Belanger
Siddharth Bhat	David Bindel	Fred Blair
Vladimir Bondarenko	Mark Botch	Raoul Bourquin
Alexandre Bouyer	Karen Braman	Wolfgang Brehm
Peter A. Broadbery	Martin Brock	Manuel Bronstein
Christopher Brown	Stephen Buchwald	Florian Bundschuh
Luanne Burns	William Burge	Ralph Byers
Quentin Carpent	Jacques Carette	Pierre Casteran
Robert Cavines	Pablo Cayuela	Bruce Char
Ondrej Certik	Tzu-Yi Chen	Bobby Cheng
Cheekai Chin	David V. Chudnovsky	Gregory V. Chudnovsky
Mark Clements	Roland Coeurjoly	Emil Cohen
Hirsh Cohen	Josh Cohen	James Cloos
Jia Zhao Cong	Christophe Conil	Don Coppersmith
George Corliss	Robert Corless	Gary Cornell
Frank Costa	Meino Cramer	Karl Crary
Jeremy Du Croz	David Cyganski	Nathaniel Daly
Timothy Daly Sr.	Timothy Daly Jr.	James H. Davenport
David Day	James Demmel	Didier Deshommes
Michael Dewar	Inderjit Dhillon	Jack Dongarra
Jean Della Dora	Gabriel Dos Reis	Claire DiCrescendo
Sam Dooley	Nicolas James Doye	Zlatko Drmac
Lionel Ducos	Iain Duff	Lee Duhem
Martin Dunstan	Brian Dupee	Dominique Duval
Robert Edwards	Hans-Dieter Ehrich	Heow Eide-Goodman
Carl Engelman	Lars Erickson	Mark Fahey
William Farmer	Richard Fateman	Bertfried Fauser
Stuart Feldman	John Fletcher	Brian Ford
Albrecht Fortenbacher	George Frances	Constantine Frangos
Timothy Freeman	Korrinn Fu	Marc Gaetano
Rudiger Gebauer	Van de Geijn	Kathy Gerber
Patricia Gianni	Gustavo Goertkin	Samantha Goldrich
Max Goldstein	Holger Gollan	Teresa Gomez-Diaz
Ralph Gomory	Laureano Gonzalez-Vega	Stephen Gortler
Johannes Grabmeier	Matt Grayson	Martin Griss
Klaus Ebbe Grue	James Griesmer	Vladimir Grinberg
Oswald Gschnitzer	Ming Gu	Fred Gustavson
Jocelyn Guidry	Gaetan Hache	Steve Hague
Satoshi Hamaguchi	Sven Hammarling	Mike Hansen
Richard Hanson	Richard Harke	Joseph Harry
Bill Hart	Vilya Harvey	Martin Hassner
Arthur S. Hathaway	Dan Hatton	Waldek Hebisch
Karl Hegbloom	Ralf Hemmecke	Tony Hearn
Henderson	Antoine Hersen	Nicholas J. Higham
Lou Hodes	Alan Hoffman	Hoon Hong
Roger House	Gernot Hueber	Pietro Iglio
Joan Jaffe	Alejandro Jakubi	Richard Jenks
Bo Kagstrom	William Kahan	Kyriakos Kalorkoti
Kai Kaminski	Grant Keady	Tom Kelsey
Wilfrid Kendall	Tony Kennedy	David Kincaid
Keshav Kini	Knut Korsvold	Ted Kosan

Paul Kosinski	Igor Kozachenko	Fred Krogh
Klaus Kusche	Bernhard Kutzler	Tim Lahey
Larry Lambe	Kaj Laurson	Charles Lawson
George L. Legendre	Franz Lehner	Frederic Lehobey
Michel Levaud	Howard Levy	J. Lewis
Ren-Cang Li	John Lipson	Rudiger Loos
Craig Lucas	Michael Lucks	Richard Luczak
Camm Maguire	Dave Mainey	Francois Maltey
William Martin	Ursula Martin	Osni Marques
Alasdair McAndrew	Bob McElrath	Michael McGettrick
Bob McNeill	Edi Meier	Ian Meikle
David Mentre	Jonathan Millen	Victor S. Miller
Gerard Milmeister	William Miranker	Mohammed Mobarak
H. Michael Moeller	Michael Monagan	Marc Moreno-Maza
Scott Morrison	Joel Moses	Mark Murray
William Naylor	Patrice Naudin	C. Andrew Neff
John Nelder	Godfrey Nolan	Arthur Norman
Jinzhong Niu	Michael O'Connor	Summat Oemrawsingh
Kostas Oikonomou	Humberto Ortiz-Zuazaga	Julian A. Padget
Bill Page	David Parnas	Norm Pass
Susan Pelzel	Michel Petitot	Didier Pinchon
Ayal Pinkus	Frederick H. Pitts	Frank Pfenning
Jose Alfredo Portes	E. Quintana-Orti	Gregorio Quintana-Orti
Beresford Parlett	A. Petitet	Andre Platzer
Peter Poromaas	Greg Puhak	Claude Quitte
Arthur C. Ralfs	Norman Ramsey	Anatoly Raportirenko
Guilherme Reis	Huan Ren	Albert D. Rich
Michael Richardson	Jason Riedy	Renaud Rioboo
Robert Risch	Jean Rivlin	Nicolas Robidoux
Simon Robinson	Raymond Rogers	Michael Rothstein
Martin Rubey	Jeff Rutter	R.W Ryniker II
Philip Santas	David Saunders	Alfred Scheerhorn
William Schelter	Gerhard Schneider	Martin Schoenert
Marshall Schor	Frithjof Schulze	Fritz Schwartz
Steven Segletes	V. Sima	Nick Simicich
William Sit	Elena Smirnova	Jacob Nyffeler Smith
Matthieu Sozeau	Srinivasan Seshan	Ken Stanley
Jonathan Steinbach	Fabio Stumbo	Christine Sundaresan
Klaus Sutner	Robert Sutor	Moss E. Sweedler
Eugene Surowitz	Yong Kiam Tan	Max Tegmark
T. Doug Telford	James Thatcher	Laurent Thery
Balbir Thomas	Mike Thomas	Carol Thompson
Dylan Thurston	Francoise Tisseur	Steve Toleque
Dick Toupin	Raymond Toy	Barry Trager
Hale Trotter	Themos T. Tsikas	Gregory Vanuxem
Kresimir Veselic	Christof Voemel	E.G. Wagner
Bernhard Wall	Paul Wang	Stephen Watt
Andreas Weber	Jaap Weel	Al Weis
Juergen Weiss	M. Weller	Mark Wegman
James Wen	Thorsten Werther	Michael Wester
R. Clint Whaley	James T. Wheeler	John M. Wiley
Berhard Will	Clifton J. Williamson	Stephen Wilson
Shmuel Winograd	Robert Wisbauer	Sandra Wityak
Waldemar Wiwianka	Knut Wolf	Yanyang Xiao
Liu Xiaojun	Clifford Yapp	David Yun
Qian Yun	Vadim Zhytnikov	Richard Zippel
Evelyn Zoernack	Bruno Zuercher	Dan Zwillinger

Contents

Motivation **3**

Why this effort won't succeed **7**

1.1 Intellectual Complexity 10

1.1.1 Alturki and Moore [Altu19] 10

1.1.2 An example 11

1.1.3 Specification of the example in Coq 11

1.1.4 Semantics of the example in Coq 11

1.1.5 Testing execution 12

1.1.6 Specifying correctness goals 12

1.1.7 Verifying correctness 13

Related Work **15**

1.2 Type Theory 15

1.2.1 Harper [Harp18][Harp18a] 15

1.3 Other Compilers 15

1.3.1 Hoare [Hoar03] 16

1.3.2 Pearce and Groves [Pear15] 16

The Problem **17**

1.4 A Brief History 17

1.5 Parallel Development 17

1.6 Project Goals 17

1.7 Visiting Scholar 17

1.8 Refining the Project Goals 18

1.9 Reduction to Practice 18

1.10 Subgoal: A new implementation 19

An Inside-Out Approach **21**

1.11 Conventions 21

1.11.1 The CLOS types 21

1.11.2 The SPAD types 22

1.12 AxiomClass 22

1.13 Categories 23

1.14 Domains and Packages 24

A Specification Language **25**

1.14.1 The specification class 26

Primitive Support tools **27**

Overview 29
1.15 The Category Class . . . 30
1.15.1 The hasclause class . . . 30
1.15.2 The haslist class . . . 31
1.15.3 The AxiomClass class . . . 31
1.15.4 The WithClass class . . . 33
1.15.5 The AddClass class . . . 33
1.16 Helper Functions . . . 34
1.16.1 Signatures and OperationAlist . . . 36
1.16.2 The signature class . . . 36
1.16.3 The amacro class . . . 39
1.16.4 The operation class . . . 39

The Parser 45
1.17 ParserClass . . . 45
1.18 make-Sourcecode . . . 47
1.19 sourcecode-pile . . . 48
1.19.1 The sourcecode class . . . 49
1.20 FSM-abbrev . . . 49
1.20.1 Recognize abbreviation command line . . . 49
1.20.2 The abbreviation class . . . 49
1.21 FSM-comment . . . 52
1.21.1 Parse Comments . . . 52
1.22 FSM-macros . . . 52
1.23 FSM-cdpsig . . . 53
1.23.1 The CDPSigClass class . . . 53
1.24 FSM-isCategory? . . . 54
1.25 FSM-catBody . . . 55
1.26 FSM-dpBody . . . 55
1.27 FSM-variables . . . 55
1.27.1 The FSM-VAR class . . . 56
1.28 Parser support functions . . . 57
1.29 The Compiler Function . . . 58
1.30 The Language . . . 59
1.31 Finite State Machine tools . . . 60
1.31.1 The gather class . . . 61
1.32 Sourcecode Tree Utilities . . . 67
1.32.1 Gather Macros . . . 73
1.32.2 parseSignature . . . 74

Common Lisp Types 77

EBNF 79
1.33 For show output . . . 79

The Compiler 81

Sane Interpreter 83

Axioms and Logic Rules 85

The Categories **89**
1.34 Level 1 . . . 90
1.34.1 AdditiveValuationAttribute . . . 90
1.34.2 ApproximateAttribute . . . 90
1.34.3 ArbitraryExponentAttribute . . . 91
1.34.4 ArbitraryPrecisionAttribute . . . 91
1.34.5 ArcHyperbolicFunctionCategory . . . 92
1.34.6 ArcTrigonometricFunctionCategory . . . 93
1.34.7 AttributeRegistry . . . 94
1.34.8 BasicType . . . 95
1.34.9 CanonicalAttribute . . . 103
1.34.10 CanonicalClosedAttribute . . . 103
1.34.11 CanonicalUnitNormalAttribute . . . 104
1.34.12 CentralAttribute . . . 104
1.34.13 CoercibleTo . . . 105
1.34.14 CombinatorialFunctionCategory . . . 105
1.34.15 CommutativeStarAttribute . . . 106
1.34.16 ConvertibleTo . . . 106
1.34.17 ElementaryFunctionCategory . . . 107
1.34.18 Eltable . . . 108
1.34.19 FiniteAggregateAttribute . . . 108
1.34.20 HyperbolicFunctionCategory . . . 109
1.34.21 InnerEvalable . . . 109
1.34.22 JacobiIdentityAttribute . . . 110
1.34.23 LazyRepresentationAttribute . . . 110
1.34.24 LeftUnitaryAttribute . . . 111
1.34.25 ModularAlgebraicGcdOperations . . . 111
1.34.26 MultiplicativeValuationAttribute . . . 113
1.34.27 NoZeroDivisorsAttribute . . . 113
1.34.28 NotherianAttribute . . . 114
1.34.29 NullSquareAttribute . . . 114
1.34.30 OpenMath . . . 115
1.34.31 PartialTranscendentalFunctions . . . 115
1.34.32 PartiallyOrderedSetAttribute . . . 117
1.34.33 PrimitiveFunctionCategory . . . 117
1.34.34 RadicalCategory . . . 118
1.34.35 RetractableTo . . . 118
1.34.36 RightUnitaryAttribute . . . 119
1.34.37 ShallowlyMutableAttribute . . . 120
1.34.38 SpecialFunctionCategory . . . 120
1.34.39 TrigonometricFunctionCategory . . . 121
1.34.40 Type . . . 122
1.34.41 UnitsKnownAttribute . . . 122
1.35 Level 2 . . . 123
1.35.1 Aggregate . . . 123
1.35.2 CombinatorialOpsCategory . . . 124
1.35.3 EltableAggregate . . . 125
1.35.4 Evalable . . . 126
1.35.5 FortranProgramCategory . . . 126
1.35.6 FullyRetractableTo . . . 127
1.35.7 Logic . . . 128
1.35.8 Patternable . . . 128
1.35.9 PlottablePlaneCurveCategory . . . 129
1.35.10 PlottableSpaceCurveCategory . . . 129

1.35.11 RealConstant . . . 130
1.35.12 SegmentCategory . . . 131
1.35.13 SetCategory . . . 132
1.35.14 TranscendentalFunctionCategory . . . 132
1.36 Level 3 . . . 133
1.36.1 AbelianSemiGroup . . . 133
1.36.2 BlowUpMethodCategory . . . 134
1.36.3 Comparable . . . 135
1.36.4 FileCategory . . . 135
1.36.5 FileNameCategory . . . 136
1.36.6 Finite . . . 137
1.36.7 FortranFunctionCategory . . . 137
1.36.8 FortranMatrixCategory . . . 139
1.36.9 FortranMatrixFunctionCategory . . . 140
1.36.10 FortranVectorCategory . . . 141
1.36.11 FortranVectorFunctionCategory . . . 142
1.36.12 FullyEvalableOver . . . 143
1.36.13 GradedModule . . . 144
1.36.14 HomogeneousAggregate . . . 145
1.36.15 IndexedDirectProductCategory . . . 145
1.36.16 LiouvillianFunctionCategory . . . 146
1.36.17 Monad . . . 147
1.36.18 NumericalIntegrationCategory . . . 148
1.36.19 NumericalOptimizationCategory . . . 148
1.36.20 OrderedSet . . . 149
1.36.21 OrdinaryDifferentialEquationsSolverCategory . . . 150
1.36.22 PartialDifferentialEquationsSolverCategory . . . 150
1.36.23 PatternMatchable . . . 151
1.36.24 RealRootCharacterizationCategory . . . 152
1.36.25 SExpressionCategory . . . 153
1.36.26 SegmentExpansionCategory . . . 154
1.36.27 SemiGroup . . . 155
1.36.28 SetCategoryWithDegree . . . 156
1.36.29 StepThrough . . . 156
1.36.30 ThreeSpaceCategory . . . 157
1.37 Level 4 . . . 163
1.37.1 AbelianMonoid . . . 163
1.37.2 AffineSpaceCategory . . . 164
1.37.3 BagAggregate . . . 164
1.37.4 CachableSet . . . 165
1.37.5 Collection . . . 165
1.37.6 DifferentialVariableCategory . . . 166
1.37.7 ExpressionSpace . . . 167
1.37.8 FullyPatternMatchable . . . 167
1.37.9 GradedAlgebra . . . 168
1.37.10 IndexedAggregate . . . 168
1.37.11 InfinitlyClosePointCategory . . . 169
1.37.12 MonadWithUnit . . . 169
1.37.13 Monoid . . . 170
1.37.14 OrderedAbelianSemiGroup . . . 170
1.37.15 OrderedFinite . . . 171
1.37.16 PlacesCategory . . . 171
1.37.17 ProjectiveSpaceCategory . . . 172
1.37.18 RecursiveAggregate . . . 172

1.37.19 TwoDimensionalArrayCategory . . . 173
1.38 Level 5 . . . 173
1.38.1 BinaryRecursiveAggregate . . . 174
1.38.2 CancellationAbelianMonoid . . . 174
1.38.3 DesingTreeCategory . . . 175
1.38.4 DoublyLinkedAggregate . . . 175
1.38.5 Group . . . 176
1.38.6 LinearAggregate . . . 177
1.38.7 MatrixCategory . . . 177
1.38.8 OrderedAbelianMonoid . . . 178
1.38.9 OrderedMonoid . . . 178
1.38.10 PolynomialSetCategory . . . 179
1.38.11 PriorityQueueAggregate . . . 180
1.38.12 QueueAggregate . . . 180
1.38.13 SetAggregate . . . 181
1.38.14 StackAggregate . . . 181
1.38.15 UnaryRecursiveAggregate . . . 182
1.39 Level 6 . . . 182
1.39.1 AbelianGroup . . . 182
1.39.2 BinaryTreeCategory . . . 183
1.39.3 DequeueAggregate . . . 183
1.39.4 DictionaryOperations . . . 184
1.39.5 ExtensibleLinearAggregate . . . 184
1.39.6 FiniteLinearAggregate . . . 185
1.39.7 FreeAbelianMonoidCategory . . . 186
1.39.8 OrderedCancellationAbelianMonoid . . . 186
1.39.9 PermutationCategory . . . 187
1.39.10 StreamAggregate . . . 187
1.39.11 TriangularSetCategory . . . 188
1.40 Level 7 . . . 188
1.40.1 Dictionary . . . 189
1.40.2 FiniteDivisorCategory . . . 189
1.40.3 LazyStreamAggregate . . . 190
1.40.4 LeftModule . . . 190
1.40.5 ListAggregate . . . 191
1.40.6 MultiDictionary . . . 191
1.40.7 MultisetAggregate . . . 192
1.40.8 NonAssociativeRng . . . 192
1.40.9 OneDimensionalArrayAggregate . . . 193
1.40.10 OrderedAbelianGroup . . . 194
1.40.11 OrderedAbelianMonoidSup . . . 194
1.40.12 RegularTriangularSetCategory . . . 195
1.40.13 RightModule . . . 196
1.40.14 Rng . . . 196
1.41 Level 8 . . . 197
1.41.1 BiModule . . . 197
1.41.2 BitAggregate . . . 197
1.41.3 FiniteSetAggregate . . . 198
1.41.4 KeyedDictionary . . . 198
1.41.5 NonAssociativeRing . . . 199
1.41.6 NormalizedTriangularSetCategory . . . 199
1.41.7 OrderedMultisetAggregate . . . 200
1.41.8 Ring . . . 201
1.41.9 SquareFreeRegularTriangularSetCategory . . . 201

1.41.10 StringAggregate . . . 202
1.41.11 VectorCategory . . . 202
1.42 Level 9 . . . 203
1.42.1 CharacteristicNonZero . . . 203
1.42.2 CharacteristicZero . . . 203
1.42.3 CommutativeRing . . . 204
1.42.4 DifferentialRing . . . 204
1.42.5 EntireRing . . . 205
1.42.6 LeftAlgebra . . . 205
1.42.7 LinearlyExplicitRingOver . . . 206
1.42.8 Module . . . 206
1.42.9 OrderedRing . . . 207
1.42.10 PartialDifferentialRing . . . 207
1.42.11 PointCategory . . . 208
1.42.12 SquareFreeNormalizedTriangularSetCategory . . . 208
1.42.13 StringCategory . . . 209
1.42.14 TableAggregate . . . 209
1.43 Level 10 . . . 210
1.43.1 Algebra . . . 210
1.43.2 AssociationListAggregate . . . 211
1.43.3 DifferentialExtension . . . 211
1.43.4 DivisorCategory . . . 212
1.43.5 FreeModuleCat . . . 212
1.43.6 FullyLinearlyExplicitRingOver . . . 213
1.43.7 LeftOreRing . . . 213
1.43.8 LieAlgebra . . . 214
1.43.9 NonAssociativeAlgebra . . . 214
1.43.10 RectangularMatrixCategory . . . 215
1.43.11 VectorSpace . . . 215
1.44 Level 11 . . . 216
1.44.1 DirectProductCategory . . . 216
1.44.2 DivisionRing . . . 217
1.44.3 FiniteRankAlgebra . . . 217
1.44.4 FiniteRankNonAssociativeAlgebra . . . 218
1.44.5 FreeLieAlgebra . . . 218
1.44.6 IntegralDomain . . . 219
1.44.7 MonogenicLinearOperator . . . 219
1.44.8 OctonionCategory . . . 220
1.44.9 SquareMatrixCategory . . . 221
1.44.10 UnivariateSkewPolynomialCategory . . . 221
1.44.11 XAlgebra . . . 222
1.45 Level 12 . . . 223
1.45.1 AbelianMonoidRing . . . 223
1.45.2 FortranMachineTypeCategory . . . 223
1.45.3 FramedAlgebra . . . 224
1.45.4 FramedNonAssociativeAlgebra . . . 224
1.45.5 GcdDomain . . . 225
1.45.6 LinearOrdinaryDifferentialOperatorCategory . . . 227
1.45.7 OrderedIntegralDomain . . . 228
1.45.8 QuaternionCategory . . . 228
1.45.9 XFreeAlgebra . . . 229
1.46 Level 13 . . . 229
1.46.1 FiniteAbelianMonoidRing . . . 230
1.46.2 IntervalCategory . . . 230

1.46.3 PowerSeriesCategory . . . 231
1.46.4 PrincipalIdealDomain . . . 231
1.46.5 UniqueFactorizationDomain . . . 232
1.46.6 XPolynomialsCat . . . 232
1.47 Level 14 . . . 233
1.47.1 EuclideanDomain . . . 233
1.47.2 MultivariateTaylorSeriesCategory . . . 234
1.47.3 PolynomialFactorizationExplicit . . . 234
1.47.4 UnivariatePowerSeriesCategory . . . 235
1.48 Level 15 . . . 235
1.48.1 Field . . . 236
1.48.2 IntegerNumberSystem . . . 236
1.48.3 PAdicIntegerCategory . . . 237
1.48.4 PolynomialCategory . . . 238
1.48.5 UnivariateTaylorSeriesCategory . . . 238
1.49 Level 16 . . . 239
1.49.1 AlgebraicallyClosedField . . . 239
1.49.2 DifferentialPolynomialCategory . . . 240
1.49.3 FieldOfPrimeCharacteristic . . . 241
1.49.4 FunctionSpace . . . 241
1.49.5 LocalPowerSeriesCategory . . . 242
1.49.6 PseudoAlgebraicClosureOfPerfectFieldCategory . . . 242
1.49.7 QuotientFieldCategory . . . 243
1.49.8 RealClosedField . . . 244
1.49.9 RealNumberSystem . . . 245
1.49.10 RecursivePolynomialCategory . . . 245
1.49.11 UnivariateLaurentSeriesCategory . . . 246
1.49.12 UnivariatePolynomialCategory . . . 246
1.49.13 UnivariatePuiseuxSeriesCategory . . . 247
1.50 Level 17 . . . 247
1.50.1 AlgebraicallyClosedFunctionSpace . . . 248
1.50.2 ExtensionField . . . 248
1.50.3 FiniteFieldCategory . . . 249
1.50.4 FloatingPointSystem . . . 249
1.50.5 UnivariateLaurentSeriesConstructorCategory . . . 250
1.50.6 UnivariatePuiseuxSeriesConstructorCategory . . . 251
1.51 Level 18 . . . 251
1.51.1 FiniteAlgebraicExtensionField . . . 251
1.51.2 MonogenicAlgebra . . . 252
1.51.3 PseudoAlgebraicClosureOfFiniteFieldCategory . . . 253
1.51.4 PseudoAlgebraicClosureOfRationalNumberCategory . . . 254
1.52 Level 19 . . . 254
1.52.1 ComplexCategory . . . 255
1.52.2 FunctionFieldCategory . . . 255
1.52.3 PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory . . . 256

The NonNegativeInteger Domain 257
1.52.4 Database Files . . . 257

The Domains **265**
1.53 A . . . 265
1.53.1 AffinePlane . . . 265
1.53.2 AffinePlaneOverPseudoAlgebraicClosureOfFiniteField . . . 265
1.53.3 AffineSpace . . . 266
1.53.4 AlgebraGivenByStructuralConstants . . . 266
1.53.5 AlgebraicFunctionField . . . 267
1.53.6 AlgebraicNumber . . . 267
1.53.7 AnonymousFunction . . . 268
1.53.8 AntiSymm . . . 268
1.53.9 Any . . . 269
1.53.10 ArrayStack . . . 269
1.53.11 Asp1 . . . 270
1.53.12 Asp10 . . . 270
1.53.13 Asp12 . . . 271
1.53.14 Asp19 . . . 272
1.53.15 Asp20 . . . 274
1.53.16 Asp24 . . . 275
1.53.17 Asp27 . . . 275
1.53.18 Asp28 . . . 276
1.53.19 Asp29 . . . 279
1.53.20 Asp30 . . . 279
1.53.21 Asp31 . . . 280
1.53.22 Asp33 . . . 281
1.53.23 Asp34 . . . 282
1.53.24 Asp35 . . . 283
1.53.25 Asp4 . . . 283
1.53.26 Asp41 . . . 284
1.53.27 Asp42 . . . 285
1.53.28 Asp49 . . . 286
1.53.29 Asp50 . . . 287
1.53.30 Asp55 . . . 288
1.53.31 Asp6 . . . 289
1.53.32 Asp7 . . . 290
1.53.33 Asp73 . . . 291
1.53.34 Asp74 . . . 291
1.53.35 Asp77 . . . 292
1.53.36 Asp78 . . . 293
1.53.37 Asp8 . . . 293
1.53.38 Asp80 . . . 294
1.53.39 Asp9 . . . 295
1.53.40 AssociatedJordanAlgebra . . . 296
1.53.41 AssociatedLieAlgebra . . . 296
1.53.42 AssociationList . . . 297
1.53.43 AttributeButtons . . . 298
1.53.44 Automorphism . . . 299
1.54 B . . . 299
1.54.1 BalancedBinaryTree . . . 299
1.54.2 BalancedPAdicInteger . . . 300
1.54.3 BalancedPAdicRational . . . 300
1.54.4 BasicFunctions . . . 301
1.54.5 BasicOperator . . . 301
1.54.6 BasicStochasticDifferential . . . 302
1.54.7 BinaryExpansion . . . 302

1.54.8 BinaryFile . . . 303
1.54.9 BinarySearchTree . . . 303
1.54.10 BinaryTournament . . . 304
1.54.11 BinaryTree . . . 304
1.54.12 Bits . . . 305
1.54.13 BlowUpWithHamburgerNoether . . . 305
1.54.14 BlowUpWithQuadTrans . . . 305
1.54.15 Boolean . . . 306
1.55 C . . . 306
1.55.1 CardinalNumber . . . 306
1.55.2 CartesianTensor . . . 307
1.55.3 Cell . . . 308
1.55.4 Character . . . 308
1.55.5 CharacterClass . . . 309
1.55.6 CliffordAlgebra . . . 309
1.55.7 Color . . . 310
1.55.8 Commutator . . . 310
1.55.9 Complex . . . 311
1.55.10 ComplexDoubleFloatMatrix . . . 311
1.55.11 ComplexDoubleFloatVector . . . 312
1.55.12 ContinuedFraction . . . 312
1.56 D . . . 313
1.56.1 Database . . . 313
1.56.2 DataList . . . 313
1.56.3 DecimalExpansion . . . 314
1.56.4 DenavitHartenbergMatrix . . . 314
1.56.5 Dequeue . . . 315
1.56.6 DeRhamComplex . . . 316
1.56.7 DesingTree . . . 316
1.56.8 DifferentialSparseMultivariatePolynomial . . . 317
1.56.9 DirectProduct . . . 317
1.56.10 DirectProductMatrixModule . . . 318
1.56.11 DirectProductModule . . . 318
1.56.12 DirichletRing . . . 319
1.56.13 DistributedMultivariatePolynomial . . . 319
1.56.14 Divisor . . . 320
1.56.15 DoubleFloat . . . 320
1.56.16 DoubleFloatMatrix . . . 321
1.56.17 DoubleFloatVector . . . 322
1.56.18 DrawOption . . . 322
1.56.19 d01ajfAnnaType . . . 323
1.56.20 d01akfAnnaType . . . 323
1.56.21 d01alfAnnaType . . . 324
1.56.22 d01amfAnnaType . . . 324
1.56.23 d01anfAnnaType . . . 325
1.56.24 d01apfAnnaType . . . 325
1.56.25 d01aqfAnnaType . . . 326
1.56.26 d01asfAnnaType . . . 327
1.56.27 d01fcfAnnaType . . . 327
1.56.28 d01gbfAnnaType . . . 328
1.56.29 d01TransformFunctionType . . . 328
1.56.30 d02bbfAnnaType . . . 329
1.56.31 d02bhfAnnaType . . . 329
1.56.32 d02cjfAnnaType . . . 330

1.56.33 d02ejfAnnaType . . . 331
1.56.34 d03eefAnnaType . . . 331
1.56.35 d03fafAnnaType . . . 332
1.57 E . . . 332
1.57.1 ElementaryFunctionsUnivariateLaurentSeries . . . 332
1.57.2 ElementaryFunctionsUnivariatePuiseuxSeries . . . 333
1.57.3 Equation . . . 333
1.57.4 EqTable . . . 334
1.57.5 EuclideanModularRing . . . 334
1.57.6 Exit . . . 335
1.57.7 ExponentialExpansion . . . 335
1.57.8 Expression . . . 336
1.57.9 ExponentialOfUnivariatePuiseuxSeries . . . 337
1.57.10 ExtAlgBasis . . . 337
1.57.11 e04dgfAnnaType . . . 338
1.57.12 e04fdfAnnaType . . . 338
1.57.13 e04gcfAnnaType . . . 339
1.57.14 e04jafAnnaType . . . 340
1.57.15 e04mbfAnnaType . . . 340
1.57.16 e04nafAnnaType . . . 341
1.57.17 e04ucfAnnaType . . . 341
1.58 F . . . 342
1.58.1 Factored . . . 342
1.58.2 File . . . 343
1.58.3 FileName . . . 343
1.58.4 FiniteDivisor . . . 344
1.58.5 FiniteField . . . 344
1.58.6 FiniteFieldCyclicGroup . . . 345
1.58.7 FiniteFieldCyclicGroupExtension . . . 345
1.58.8 FiniteFieldCyclicGroupExtensionByPolynomial . . . 346
1.58.9 FiniteFieldExtension . . . 346
1.58.10 FiniteFieldExtensionByPolynomial . . . 347
1.58.11 FiniteFieldNormalBasis . . . 347
1.58.12 FiniteFieldNormalBasisExtension . . . 348
1.58.13 FiniteFieldNormalBasisExtensionByPolynomial . . . 349
1.58.14 FlexibleArray . . . 349
1.58.15 Float . . . 350
1.58.16 FortranCode . . . 351
1.58.17 FortranExpression . . . 352
1.58.18 FortranProgram . . . 352
1.58.19 FortranScalarType . . . 353
1.58.20 FortranTemplate . . . 353
1.58.21 FortranType . . . 354
1.58.22 FourierComponent . . . 354
1.58.23 FourierSeries . . . 355
1.58.24 Fraction . . . 355
1.58.25 FractionalIdeal . . . 356
1.58.26 FramedModule . . . 356
1.58.27 FreeAbelianGroup . . . 357
1.58.28 FreeAbelianMonoid . . . 357
1.58.29 FreeGroup . . . 358
1.58.30 FreeModule . . . 358
1.58.31 FreeModule1 . . . 359
1.58.32 FreeMonoid . . . 359

1.58.33 FreeNilpotentLie . . . 360
1.58.34 FullPartialFractionExpansion . . . 360
1.58.35 FunctionCalled . . . 361
1.59 G . . . 361
1.59.1 GeneralDistributedMultivariatePolynomial . . . 361
1.59.2 GeneralModulePolynomial . . . 362
1.59.3 GenericNonAssociativeAlgebra . . . 362
1.59.4 GeneralPolynomialSet . . . 363
1.59.5 GeneralSparseTable . . . 363
1.59.6 GeneralTriangularSet . . . 364
1.59.7 GeneralUnivariatePowerSeries . . . 364
1.59.8 GraphImage . . . 365
1.59.9 GuessOption . . . 365
1.59.10 GuessOptionFunctions0 . . . 366
1.60 H . . . 366
1.60.1 HashTable . . . 366
1.60.2 Heap . . . 367
1.60.3 HexadecimalExpansion . . . 367
1.60.4 HTMLFormat . . . 368
1.60.5 HomogeneousDirectProduct . . . 368
1.60.6 HomogeneousDistributedMultivariatePolynomial . . . 369
1.60.7 HyperellipticFiniteDivisor . . . 369
1.61 I . . . 370
1.61.1 InfClsPt . . . 370
1.61.2 IndexCard . . . 370
1.61.3 IndexedBits . . . 371
1.61.4 IndexedDirectProductAbelianGroup . . . 371
1.61.5 IndexedDirectProductAbelianMonoid . . . 372
1.61.6 IndexedDirectProductObject . . . 372
1.61.7 IndexedDirectProductOrderedAbelianMonoid . . . 373
1.61.8 IndexedDirectProductOrderedAbelianMonoidSup . . . 373
1.61.9 IndexedExponents . . . 374
1.61.10 IndexedFlexibleArray . . . 374
1.61.11 IndexedList . . . 375
1.61.12 IndexedMatrix . . . 376
1.61.13 IndexedOneDimensionalArray . . . 376
1.61.14 IndexedString . . . 377
1.61.15 IndexedTwoDimensionalArray . . . 377
1.61.16 IndexedVector . . . 378
1.61.17 InfiniteTuple . . . 378
1.61.18 InfinitlyClosePoint . . . 379
1.61.19 InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField . . . 379
1.61.20 InnerAlgebraicNumber . . . 380
1.61.21 InnerFiniteField . . . 380
1.61.22 InnerFreeAbelianMonoid . . . 381
1.61.23 InnerIndexedTwoDimensionalArray . . . 381
1.61.24 InnerPAdicInteger . . . 382
1.61.25 InnerPrimeField . . . 382
1.61.26 InnerSparseUnivariatePowerSeries . . . 383
1.61.27 InnerTable . . . 383
1.61.28 InnerTaylorSeries . . . 384
1.61.29 InputForm . . . 384
1.61.30 Integer . . . 385
1.61.31 IntegerMod . . . 385

1.61.32 IntegrationFunctionsTable . . . 386
1.61.33 IntegrationResult . . . 386
1.61.34 Interval . . . 387
1.62 K . . . 387
1.62.1 Kernel . . . 387
1.62.2 KeyedAccessFile . . . 388
1.63 L . . . 388
1.63.1 LaurentPolynomial . . . 388
1.63.2 Library . . . 389
1.63.3 LieExponentials . . . 389
1.63.4 LiePolynomial . . . 390
1.63.5 LieSquareMatrix . . . 390
1.63.6 LinearOrdinaryDifferentialOperator . . . 391
1.63.7 LinearOrdinaryDifferentialOperator1 . . . 392
1.63.8 LinearOrdinaryDifferentialOperator2 . . . 392
1.63.9 List . . . 393
1.63.10 ListMonoidOps . . . 393
1.63.11 ListMultiDictionary . . . 394
1.63.12 LocalAlgebra . . . 394
1.63.13 Localize . . . 395
1.63.14 LyndonWord . . . 395
1.64 M . . . 396
1.64.1 MachineComplex . . . 396
1.64.2 MachineFloat . . . 396
1.64.3 MachineInteger . . . 397
1.64.4 Magma . . . 397
1.64.5 MakeCachableSet . . . 398
1.64.6 MathMLFormat . . . 398
1.64.7 Matrix . . . 399
1.64.8 ModMonic . . . 399
1.64.9 ModularField . . . 400
1.64.10 ModularRing . . . 400
1.64.11 ModuleMonomial . . . 401
1.64.12 ModuleOperator . . . 401
1.64.13 MoebiusTransform . . . 402
1.64.14 MonoidRing . . . 402
1.64.15 Multiset . . . 403
1.64.16 MultivariatePolynomial . . . 403
1.64.17 MyExpression . . . 404
1.64.18 MyUnivariatePolynomial . . . 404
1.65 N . . . 405
1.65.1 NeitherSparseOrDensePowerSeries . . . 405
1.65.2 NewSparseMultivariatePolynomial . . . 405
1.65.3 NewSparseUnivariatePolynomial . . . 406
1.65.4 None . . . 406
1.65.5 NonNegativeInteger . . . 407
1.65.6 NottinghamGroup . . . 407
1.65.7 NumericalIntegrationProblem . . . 408
1.65.8 NumericalODEProblem . . . 409
1.65.9 NumericalOptimizationProblem . . . 409
1.65.10 NumericalPDEProblem . . . 410
1.66 O . . . 411
1.66.1 Octonion . . . 411
1.66.2 ODEIntensityFunctionsTable . . . 411

1.66.3 OneDimensionalArray . . . 412
1.66.4 OnePointCompletion . . . 412
1.66.5 OpenMathConnection . . . 413
1.66.6 OpenMathDevice . . . 413
1.66.7 OpenMathEncoding . . . 414
1.66.8 OpenMathError . . . 414
1.66.9 OpenMathErrorKind . . . 415
1.66.10 Operator . . . 415
1.66.11 OppositeMonogenicLinearOperator . . . 416
1.66.12 OrderedCompletion . . . 416
1.66.13 OrderedDirectProduct . . . 417
1.66.14 OrderedFreeMonoid . . . 417
1.66.15 OrderedVariableList . . . 418
1.66.16 OrderlyDifferentialPolynomial . . . 418
1.66.17 OrderlyDifferentialVariable . . . 419
1.66.18 OrdinaryDifferentialRing . . . 420
1.66.19 OrdinaryWeightedPolynomials . . . 420
1.66.20 OrdSetInts . . . 421
1.66.21 OutputForm . . . 421
1.67 P . . . 422
1.67.1 PAdicInteger . . . 422
1.67.2 PAdicRational . . . 422
1.67.3 PAdicRationalConstructor . . . 423
1.67.4 Palette . . . 423
1.67.5 ParametricPlaneCurve . . . 423
1.67.6 ParametricSpaceCurve . . . 424
1.67.7 ParametricSurface . . . 424
1.67.8 PartialFraction . . . 425
1.67.9 Partition . . . 426
1.67.10 Pattern . . . 426
1.67.11 PatternMatchListResult . . . 427
1.67.12 PatternMatchResult . . . 427
1.67.13 PendantTree . . . 428
1.67.14 Permutation . . . 428
1.67.15 PermutationGroup . . . 429
1.67.16 Pi . . . 429
1.67.17 PlaneAlgebraicCurvePlot . . . 430
1.67.18 Places . . . 430
1.67.19 PlacesOverPseudoAlgebraicClosureOfFiniteField . . . 431
1.67.20 Plcs . . . 431
1.67.21 Plot . . . 432
1.67.22 Plot3D . . . 432
1.67.23 PoincareBirkhoffWittLyndonBasis . . . 433
1.67.24 Point . . . 433
1.67.25 Polynomial . . . 434
1.67.26 PolynomialIdeals . . . 434
1.67.27 PolynomialRing . . . 435
1.67.28 PositiveInteger . . . 435
1.67.29 PrimeField . . . 436
1.67.30 PrimitiveArray . . . 436
1.67.31 Product . . . 437
1.67.32 ProjectivePlane . . . 437
1.67.33 ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField . . . 438
1.67.34 ProjectiveSpace . . . 438

1.67.35 PseudoAlgebraicClosureOfAlgExtOfRationalNumber 439
1.67.36 PseudoAlgebraicClosureOfFiniteField 439
1.67.37 PseudoAlgebraicClosureOfRationalNumber 440
1.68 Q 441
1.68.1 QuadraticForm 441
1.68.2 QuasiAlgebraicSet 441
1.68.3 Quaternion 442
1.68.4 QueryEquation 443
1.68.5 Queue 443
1.69 R 444
1.69.1 RadicalFunctionField 444
1.69.2 RadixExpansion 444
1.69.3 RealClosure 445
1.69.4 RectangularMatrix 445
1.69.5 Reference 446
1.69.6 RegularChain 446
1.69.7 RegularTriangularSet 447
1.69.8 ResidueRing 447
1.69.9 Result 448
1.69.10 RewriteRule 448
1.69.11 RightOpenIntervalRootCharacterization 449
1.69.12 RomanNumeral 449
1.69.13 RoutinesTable 450
1.69.14 RuleCalled 450
1.69.15 Ruleset 451
1.70 S 451
1.70.1 ScriptFormulaFormat 451
1.70.2 Segment 452
1.70.3 SegmentBinding 452
1.70.4 Set 453
1.70.5 SetOfMIntegersInOneToN 453
1.70.6 SequentialDifferentialPolynomial 454
1.70.7 SequentialDifferentialVariable 454
1.70.8 SExpression 455
1.70.9 SExpressionOf 456
1.70.10 SimpleAlgebraicExtension 456
1.70.11 SimpleCell 457
1.70.12 SimpleFortranProgram 457
1.70.13 SingleInteger 458
1.70.14 SingletonAsOrderedSet 458
1.70.15 SparseEchelonMatrix 459
1.70.16 SparseMultivariatePolynomial 459
1.70.17 SparseMultivariateTaylorSeries 460
1.70.18 SparseTable 460
1.70.19 SparseUnivariateLaurentSeries 461
1.70.20 SparseUnivariatePolynomial 461
1.70.21 SparseUnivariatePolynomialExpressions 462
1.70.22 SparseUnivariatePuiseuxSeries 462
1.70.23 SparseUnivariateSkewPolynomial 463
1.70.24 SparseUnivariateTaylorSeries 463
1.70.25 SplitHomogeneousDirectProduct 464
1.70.26 SplittingNode 464
1.70.27 SplittingTree 465
1.70.28 SquareFreeRegularTriangularSet 466

1.70.29 SquareMatrix . . . 466
1.70.30 Stack . . . 467
1.70.31 StochasticDifferential . . . 467
1.70.32 Stream . . . 468
1.70.33 String . . . 468
1.70.34 StringTable . . . 469
1.70.35 SubSpace . . . 469
1.70.36 SubSpaceComponentProperty . . . 470
1.70.37 SuchThat . . . 470
1.70.38 Switch . . . 471
1.70.39 Symbol . . . 471
1.70.40 SymbolTable . . . 472
1.70.41 SymmetricPolynomial . . . 472
1.71 T . . . 473
1.71.1 Table . . . 473
1.71.2 Tableau . . . 473
1.71.3 TaylorSerieso . . . 474
1.71.4 TexFormat . . . 474
1.71.5 TextFile . . . 475
1.71.6 TheSymbolTable . . . 475
1.71.7 ThreeDimensionalMatrix . . . 476
1.71.8 ThreeDimensionalViewport . . . 476
1.71.9 ThreeSpace . . . 477
1.71.10 Tree . . . 477
1.71.11 TubePlot . . . 478
1.71.12 Tuple . . . 478
1.71.13 TwoDimensionalArray . . . 479
1.71.14 TwoDimensionalViewport . . . 479
1.72 U . . . 480
1.72.1 UnivariateFormalPowerSeries . . . 480
1.72.2 UnivariateLaurentSeries . . . 480
1.72.3 UnivariateLaurentSeriesConstructor . . . 481
1.72.4 UnivariatePolynomial . . . 481
1.72.5 UnivariatePuiseuxSeries . . . 482
1.72.6 UnivariatePuiseuxSeriesConstructor . . . 482
1.72.7 UnivariatePuiseuxSeriesWithExponentialSingularity . . . 483
1.72.8 UnivariateSkewPolynomial . . . 483
1.72.9 UnivariateTaylorSeries . . . 484
1.72.10 UnivariateTaylorSeriesCZero . . . 485
1.72.11 UniversalSegment . . . 485
1.72.12 U8Matrix . . . 485
1.72.13 U16Matrix . . . 486
1.72.14 U32Matrix . . . 486
1.72.15 U8Vector . . . 487
1.72.16 U16Vector . . . 487
1.72.17 U32Vector . . . 488
1.73 V . . . 489
1.73.1 Variable . . . 489
1.73.2 Vector . . . 489
1.73.3 Void . . . 490
1.74 W . . . 490
1.74.1 WeightedPolynomials . . . 490
1.74.2 WuWenTsunTriangularSet . . . 491
1.75 X . . . 491

1.75.1 XDistributedPolynomial . . . 491
1.75.2 XPBWPolynomial . . . 492
1.75.3 XPolynomial . . . 492
1.75.4 XPolynomialRing . . . 493
1.75.5 XRecursivePolynomial . . . 493

The Packages **495**
1.76 A . . . 495
1.76.1 AffineAlgebraicSetComputeWithGroebnerBasis . . . 495
1.76.2 AffineAlgebraicSetComputeWithResultant . . . 495
1.76.3 AlgebraicFunction . . . 496
1.76.4 AlgebraicHermiteIntegration . . . 496
1.76.5 AlgebraicIntegrate . . . 497
1.76.6 AlgebraicIntegration . . . 497
1.76.7 AlgebraicManipulations . . . 498
1.76.8 AlgebraicMultFact . . . 498
1.76.9 AlgebraPackage . . . 499
1.76.10 AlgFactor . . . 499
1.76.11 AnnaNumericalIntegrationPackage . . . 500
1.76.12 AnnaNumericalOptimizationPackage . . . 500
1.76.13 AnnaOrdinaryDifferentialEquationPackage . . . 501
1.76.14 AnnaPartialDifferentialEquationPackage . . . 501
1.76.15 AnyFunctions1 . . . 502
1.76.16 ApplicationProgramInterface . . . 502
1.76.17 ApplyRules . . . 503
1.76.18 ApplyUnivariateSkewPolynomial . . . 503
1.76.19 AssociatedEquations . . . 503
1.76.20 AttachPredicates . . . 504
1.76.21 AxiomServer . . . 504
1.77 B . . . 505
1.77.1 BalancedFactorisation . . . 505
1.77.2 BasicOperatorFunctions1 . . . 505
1.77.3 Bezier . . . 506
1.77.4 BezoutMatrix . . . 506
1.77.5 BlowUpPackage . . . 507
1.77.6 BoundIntegerRoots . . . 507
1.77.7 BrillhartTests . . . 508
1.78 C . . . 508
1.78.1 CartesianTensorFunctions2 . . . 508
1.78.2 ChangeOfVariable . . . 509
1.78.3 CharacteristicPolynomialInMonogenicalAlgebra . . . 509
1.78.4 CharacteristicPolynomialPackage . . . 510
1.78.5 ChineseRemainderToolsForIntegralBases . . . 510
1.78.6 CoerceVectorMatrixPackage . . . 511
1.78.7 CombinatorialFunction . . . 511
1.78.8 CommonDenominator . . . 512
1.78.9 CommonOperators . . . 512
1.78.10 CommuteUnivariatePolynomialCategory . . . 513
1.78.11 ComplexFactorization . . . 513
1.78.12 ComplexFunctions2 . . . 514
1.78.13 ComplexIntegerSolveLinearPolynomialEquation . . . 514
1.78.14 ComplexPattern . . . 515
1.78.15 ComplexPatternMatch . . . 515
1.78.16 ComplexRootFindingPackage . . . 515

1.78.17 ComplexRootPackage . . . 516
1.78.18 ComplexTrigonometricManipulations . . . 517
1.78.19 ConstantLODE . . . 517
1.78.20 CoordinateSystems . . . 518
1.78.21 CRApackage . . . 518
1.78.22 CycleIndicators . . . 519
1.78.23 CyclicStreamTools . . . 519
1.78.24 CyclotomicPolynomialPackage . . . 520
1.78.25 CylindricalAlgebraicDecompositionPackage . . . 520
1.78.26 CylindricalAlgebraicDecompositionUtilities . . . 520
1.79 D . . . 521
1.79.1 DefiniteIntegrationTools . . . 521
1.79.2 DegreeReductionPackage . . . 521
1.79.3 DesingTreePackage . . . 522
1.79.4 DiophantineSolutionPackage . . . 522
1.79.5 DirectProductFunctions2 . . . 523
1.79.6 DiscreteLogarithmPackage . . . 524
1.79.7 DisplayPackage . . . 524
1.79.8 DistinctDegreeFactorize . . . 525
1.79.9 DoubleFloatSpecialFunctions . . . 525
1.79.10 DoubleResultantPackage . . . 526
1.79.11 DrawComplex . . . 526
1.79.12 DrawNumericHack . . . 526
1.79.13 DrawOptionFunctions0 . . . 527
1.79.14 DrawOptionFunctions1 . . . 528
1.79.15 d01AgentsPackage . . . 528
1.79.16 d01WeightsPackage . . . 529
1.79.17 d02AgentsPackage . . . 529
1.79.18 d03AgentsPackage . . . 530
1.80 E . . . 530
1.80.1 EigenPackage . . . 530
1.80.2 ElementaryFunction . . . 531
1.80.3 ElementaryFunctionDefiniteIntegration . . . 531
1.80.4 ElementaryFunctionLODESolver . . . 532
1.80.5 ElementaryFunctionODESolver . . . 532
1.80.6 ElementaryFunctionSign . . . 533
1.80.7 ElementaryFunctionStructurePackage . . . 533
1.80.8 ElementaryIntegration . . . 534
1.80.9 ElementaryRischDE . . . 534
1.80.10 ElementaryRischDESystem . . . 534
1.80.11 EllipticFunctionsUnivariateTaylorSeries . . . 535
1.80.12 EquationFunctions2 . . . 535
1.80.13 ErrorFunctions . . . 536
1.80.14 EuclideanGroebnerBasisPackage . . . 537
1.80.15 EvaluateCycleIndicators . . . 538
1.80.16 ExpertSystemContinuityPackage . . . 538
1.80.17 ExpertSystemContinuityPackage1 . . . 538
1.80.18 ExpertSystemToolsPackage . . . 539
1.80.19 ExpertSystemToolsPackage1 . . . 539
1.80.20 ExpertSystemToolsPackage2 . . . 540
1.80.21 ExpressionFunctions2 . . . 540
1.80.22 ExpressionSolve . . . 541
1.80.23 ExpressionSpaceFunctions1 . . . 541
1.80.24 ExpressionSpaceFunctions2 . . . 542

1.80.25 ExpressionSpaceODESolver . . . 542
1.80.26 ExpressionToOpenMath . . . 543
1.80.27 ExpressionToUnivariatePowerSeries . . . 543
1.80.28 ExpressionTubePlot . . . 544
1.80.29 Export3D . . . 544
1.80.30 e04AgentsPackage . . . 545
1.81 F . . . 545
1.81.1 FactoredFunctions . . . 545
1.81.2 FactoredFunctions2 . . . 546
1.81.3 FactoredFunctionUtilities . . . 546
1.81.4 FactoringUtilities . . . 547
1.81.5 FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber . . . 547
1.81.6 FactorisationOverPseudoAlgebraicClosureOfRationalNumber . . . 548
1.81.7 FGLMIfCanPackage . . . 548
1.81.8 FindOrderFinite . . . 549
1.81.9 FiniteAbelianMonoidRingFunctions2 . . . 549
1.81.10 FiniteDivisorFunctions2 . . . 550
1.81.11 FiniteFieldFactorization . . . 550
1.81.12 FiniteFieldFactorizationWithSizeParseBySideEffect . . . 551
1.81.13 FiniteFieldFunctions . . . 551
1.81.14 FiniteFieldHomomorphisms . . . 552
1.81.15 FiniteFieldPolynomialPackage . . . 552
1.81.16 FiniteFieldPolynomialPackage2 . . . 553
1.81.17 FiniteFieldSolveLinearPolynomialEquation . . . 553
1.81.18 FiniteFieldSquareFreeDecomposition . . . 554
1.81.19 FiniteLinearAggregateFunctions2 . . . 554
1.81.20 FiniteLinearAggregateSort . . . 555
1.81.21 FiniteSetAggregateFunctions2 . . . 555
1.81.22 FloatingComplexPackage . . . 556
1.81.23 FloatingRealPackage . . . 556
1.81.24 FloatSpecialFunctions . . . 557
1.81.25 FortranCodePackage1 . . . 557
1.81.26 FortranOutputStackPackage . . . 558
1.81.27 FortranPackage . . . 558
1.81.28 FractionalIdealFunctions2 . . . 559
1.81.29 FractionFreeFastGaussian . . . 559
1.81.30 FractionFreeFastGaussianFractions . . . 560
1.81.31 FractionFunctions2 . . . 560
1.81.32 FramedNonAssociativeAlgebraFunctions2 . . . 561
1.81.33 FunctionalSpecialFunction . . . 561
1.81.34 FunctionFieldCategoryFunctions2 . . . 562
1.81.35 FunctionFieldIntegralBasis . . . 562
1.81.36 FunctionSpaceAssertions . . . 563
1.81.37 FunctionSpaceAttachPredicates . . . 563
1.81.38 FunctionSpaceComplexIntegration . . . 564
1.81.39 FunctionSpaceFunctions2 . . . 564
1.81.40 FunctionSpaceIntegration . . . 565
1.81.41 FunctionSpacePrimitiveElement . . . 565
1.81.42 FunctionSpaceReduce . . . 566
1.81.43 FunctionSpaceSum . . . 566
1.81.44 FunctionSpaceToExponentialExpansion . . . 567
1.81.45 FunctionSpaceToUnivariatePowerSeries . . . 567
1.81.46 FunctionSpaceUnivariatePolynomialFactor . . . 568
1.82 G . . . 568

1.82.1 GaloisGroupFactorizationUtilities . . . 568
1.82.2 GaloisGroupFactorizer . . . 569
1.82.3 GaloisGroupPolynomialUtilities . . . 569
1.82.4 GaloisGroupUtilities . . . 570
1.82.5 GaussianFactorizationPackage . . . 570
1.82.6 GeneralHenselPackage . . . 571
1.82.7 GeneralizedMultivariateFactorize . . . 571
1.82.8 GeneralPackageForAlgebraicFunctionField . . . 572
1.82.9 GeneralPolynomialGcdPackage . . . 572
1.82.10 GenerateUnivariatePowerSeries . . . 573
1.82.11 GenExEuclid . . . 573
1.82.12 GenUFactorize . . . 573
1.82.13 GenusZeroIntegration . . . 574
1.82.14 GnuDraw . . . 575
1.82.15 GosperSummationMethod . . . 575
1.82.16 GraphicsDefaults . . . 576
1.82.17 Graphviz . . . 576
1.82.18 GrayCode . . . 576
1.82.19 GroebnerFactorizationPackage . . . 577
1.82.20 GroebnerInternalPackage . . . 578
1.82.21 GroebnerPackage . . . 578
1.82.22 GroebnerSolve . . . 579
1.82.23 Guess . . . 579
1.82.24 GuessAlgebraicNumber . . . 580
1.82.25 GuessFinite . . . 580
1.82.26 GuessFiniteFunctions . . . 581
1.82.27 GuessInteger . . . 581
1.82.28 GuessPolynomial . . . 582
1.82.29 GuessUnivariatePolynomial . . . 582
1.83 H . . . 583
1.83.1 HallBasis . . . 583
1.83.2 HeuGcd . . . 583
1.84 I . . . 584
1.84.1 IdealDecompositionPackage . . . 584
1.84.2 IncrementingMaps . . . 584
1.84.3 InfiniteProductCharacteristicZero . . . 585
1.84.4 InfiniteProductFiniteField . . . 585
1.84.5 InfiniteProductPrimeField . . . 586
1.84.6 InfiniteTupleFunctions2 . . . 586
1.84.7 InfiniteTupleFunctions3 . . . 587
1.84.8 Infinity . . . 587
1.84.9 InnerAlgFactor . . . 588
1.84.10 InnerCommonDenominator . . . 588
1.84.11 InnerMatrixLinearAlgebraFunctions . . . 589
1.84.12 InnerMatrixQuotientFieldFunctions . . . 589
1.84.13 InnerModularGcd . . . 590
1.84.14 InnerMultFact . . . 590
1.84.15 InnerNormalBasisFieldFunctions . . . 591
1.84.16 InnerNumericEigenPackage . . . 591
1.84.17 InnerNumericFloatSolvePackage . . . 592
1.84.18 InnerPolySign . . . 592
1.84.19 InnerPolySum . . . 593
1.84.20 InnerTrigonometricManipulations . . . 593
1.84.21 InputFormFunctions1 . . . 594

1.84.22 InterfaceGroebnerPackage 594
1.84.23 IntegerBits 594
1.84.24 IntegerCombinatoricFunctions 595
1.84.25 IntegerFactorizationPackage 595
1.84.26 IntegerLinearDependence 596
1.84.27 IntegerNumberTheoryFunctions 596
1.84.28 IntegerPrimesPackage 597
1.84.29 IntegerRetractions 597
1.84.30 IntegerRoots 598
1.84.31 IntegerSolveLinearPolynomialEquation 598
1.84.32 IntegralBasisTools 599
1.84.33 IntegralBasisPolynomialTools 599
1.84.34 IntegrationResultFunctions2 600
1.84.35 IntegrationResultRFToFunction 600
1.84.36 IntegrationResultToFunction 601
1.84.37 IntegrationTools 601
1.84.38 InternalPrintPackage 602
1.84.39 InternalRationalUnivariateRepresentationPackage 602
1.84.40 InterpolateFormsPackage 603
1.84.41 IntersectionDivisorPackage 603
1.84.42 IrredPolyOverFiniteField 604
1.84.43 IrrRepSymNatPackage 604
1.84.44 InverseLaplaceTransform 605
1.85 K 605
1.85.1 KernelFunctions2 605
1.85.2 Kovacic 606
1.86 L 606
1.86.1 LaplaceTransform 606
1.86.2 LazardSetSolvingPackage 607
1.86.3 LeadingCoefDetermination 607
1.86.4 LexTriangularPackage 608
1.86.5 LinearDependence 608
1.86.6 LinearOrdinaryDifferentialOperatorFactorizer 609
1.86.7 LinearOrdinaryDifferentialOperatorsOps 609
1.86.8 LinearPolynomialEquationByFractions 610
1.86.9 LinearSystemFromPowerSeriesPackage 610
1.86.10 LinearSystemMatrixPackage 611
1.86.11 LinearSystemMatrixPackage1 611
1.86.12 LinearSystemPolynomialPackage 612
1.86.13 LinGroebnerPackage 612
1.86.14 LinesOpPack 613
1.86.15 LiouvillianFunction 613
1.86.16 ListFunctions2 614
1.86.17 ListFunctions3 614
1.86.18 ListToMap 615
1.86.19 LocalParametrizationOfSimplePointPackage 615
1.87 M 616
1.87.1 MakeBinaryCompiledFunction 616
1.87.2 MakeFloatCompiledFunction 616
1.87.3 MakeFunction 617
1.87.4 MakeRecord 617
1.87.5 MakeUnaryCompiledFunction 618
1.87.6 MappingPackageInternalHacks1 618
1.87.7 MappingPackageInternalHacks2 619

1.87.8 MappingPackageInternalHacks3 619
1.87.9 MappingPackage1 620
1.87.10 MappingPackage2 620
1.87.11 MappingPackage3 621
1.87.12 MappingPackage4 621
1.87.13 MatrixCategoryFunctions2 622
1.87.14 MatrixCommonDenominator 622
1.87.15 MatrixLinearAlgebraFunctions 623
1.87.16 MatrixManipulation 623
1.87.17 MergeThing 624
1.87.18 MeshCreationRoutinesForThreeDimensions 624
1.87.19 ModularDistinctDegreeFactorizer 624
1.87.20 ModularHermitianRowReduction 625
1.87.21 MonoidRingFunctions2 625
1.87.22 MonomialExtensionTools 626
1.87.23 MoreSystemCommands 626
1.87.24 MPolyCatPolyFactorizer 627
1.87.25 MPolyCatRationalFunctionFactorizer 627
1.87.26 MPolyCatFunctions2 628
1.87.27 MPolyCatFunctions3 628
1.87.28 MRationalFactorize 629
1.87.29 MultFiniteFactorize 629
1.87.30 MultipleMap 630
1.87.31 MultiVariableCalculusFunctions 630
1.87.32 MultivariateFactorize 631
1.87.33 MultivariateLifting 631
1.87.34 MultivariateSquareFree 632
1.88 N 632
1.88.1 NagEigenPackage 632
1.88.2 NagFittingPackage 633
1.88.3 NagLinearEquationSolvingPackage 634
1.88.4 NAGLinkSupportPackage 634
1.88.5 NagIntegrationPackage 635
1.88.6 NagInterpolationPackage 635
1.88.7 NagLapack 636
1.88.8 NagMatrixOperationsPackage 636
1.88.9 NagOptimisationPackage 637
1.88.10 NagOrdinaryDifferentialEquationsPackage 637
1.88.11 NagPartialDifferentialEquationsPackage 638
1.88.12 NagPolynomialRootsPackage 639
1.88.13 NagRootFindingPackage 639
1.88.14 NagSeriesSummationPackage 640
1.88.15 NagSpecialFunctionsPackage 640
1.88.16 NewSparseUnivariatePolynomialFunctions2 641
1.88.17 NewtonInterpolation 641
1.88.18 NewtonPolygon 642
1.88.19 NonCommutativeOperatorDivision 642
1.88.20 NoneFunctions1 643
1.88.21 NonLinearFirstOrderODESolver 643
1.88.22 NonLinearSolvePackage 644
1.88.23 NormalizationPackage 644
1.88.24 NormInMonogenicAlgebra 645
1.88.25 NormRetractPackage 645
1.88.26 NPCoef 645

1.88.27 NumberFieldIntegralBasis . . . 646
1.88.28 NumberFormats . . . 646
1.88.29 NumberTheoreticPolynomialFunctions . . . 647
1.88.30 Numeric . . . 647
1.88.31 NumericalOrdinaryDifferentialEquations . . . 648
1.88.32 NumericalQuadrature . . . 650
1.88.33 NumericComplexEigenPackage . . . 651
1.88.34 NumericContinuedFraction . . . 652
1.88.35 NumericRealEigenPackage . . . 652
1.88.36 NumericTubePlot . . . 653
1.89 O . . . 653
1.89.1 OctonionCategoryFunctions2 . . . 653
1.89.2 ODEIntegration . . . 654
1.89.3 ODETools . . . 654
1.89.4 OneDimensionalArrayFunctions2 . . . 655
1.89.5 OnePointCompletionFunctions2 . . . 655
1.89.6 OpenMathPackage . . . 656
1.89.7 OpenMathServerPackage . . . 656
1.89.8 OperationsQuery . . . 657
1.89.9 OrderedCompletionFunctions2 . . . 657
1.89.10 OrderingFunctions . . . 658
1.89.11 OrthogonalPolynomialFunctions . . . 658
1.89.12 OutputPackage . . . 659
1.90 P . . . 659
1.90.1 PackageForAlgebraicFunctionField . . . 659
1.90.2 PackageForAlgebraicFunctionFieldOverFiniteField . . . 660
1.90.3 PackageForPoly . . . 660
1.90.4 PadeApproximantPackage . . . 660
1.90.5 PadeApproximants . . . 661
1.90.6 PAdicWildFunctionFieldIntegralBasis . . . 661
1.90.7 ParadoxicalCombinatorsForStreams . . . 662
1.90.8 ParametricLinearEquations . . . 662
1.90.9 ParametricPlaneCurveFunctions2 . . . 663
1.90.10 ParametricSpaceCurveFunctions2 . . . 664
1.90.11 ParametricSurfaceFunctions2 . . . 664
1.90.12 ParametrizationPackage . . . 664
1.90.13 PartialFractionPackage . . . 665
1.90.14 PartitionsAndPermutations . . . 665
1.90.15 PatternFunctions1 . . . 666
1.90.16 PatternFunctions2 . . . 666
1.90.17 PatternMatch . . . 667
1.90.18 PatternMatchAssertions . . . 667
1.90.19 PatternMatchFunctionSpace . . . 668
1.90.20 PatternMatchIntegerNumberSystem . . . 668
1.90.21 PatternMatchIntegration . . . 669
1.90.22 PatternMatchKernel . . . 669
1.90.23 PatternMatchListAggregate . . . 670
1.90.24 PatternMatchPolynomialCategory . . . 670
1.90.25 PatternMatchPushDown . . . 671
1.90.26 PatternMatchQuotientFieldCategory . . . 671
1.90.27 PatternMatchResultFunctions2 . . . 672
1.90.28 PatternMatchSymbol . . . 672
1.90.29 PatternMatchTools . . . 672
1.90.30 Permanent . . . 673

1.90.31 PermutationGroupExamples . . . 673
1.90.32 PiCoercions . . . 674
1.90.33 PlotFunctions1 . . . 674
1.90.34 PlotTools . . . 675
1.90.35 ProjectiveAlgebraicSetPackage . . . 675
1.90.36 PointFunctions2 . . . 676
1.90.37 PointPackage . . . 676
1.90.38 PointsOfFiniteOrder . . . 677
1.90.39 PointsOfFiniteOrderRational . . . 677
1.90.40 PointsOfFiniteOrderTools . . . 678
1.90.41 PolynomialPackageForCurve . . . 678
1.90.42 PolToPol . . . 679
1.90.43 PolyGroebner . . . 679
1.90.44 PolynomialAN2Expression . . . 680
1.90.45 PolynomialCategoryLifting . . . 680
1.90.46 PolynomialCategoryQuotientFunctions . . . 681
1.90.47 PolynomialComposition . . . 681
1.90.48 PolynomialDecomposition . . . 682
1.90.49 PolynomialFactorizationByRecursion . . . 682
1.90.50 PolynomialFactorizationByRecursionUnivariate . . . 683
1.90.51 PolynomialFunctions2 . . . 683
1.90.52 PolynomialGcdPackage . . . 684
1.90.53 PolynomialInterpolation . . . 684
1.90.54 PolynomialInterpolationAlgorithms . . . 685
1.90.55 PolynomialNumberTheoryFunctions . . . 685
1.90.56 PolynomialRoots . . . 686
1.90.57 PolynomialSetUtilitiesPackage . . . 686
1.90.58 PolynomialSolveByFormulas . . . 687
1.90.59 PolynomialSquareFree . . . 687
1.90.60 PolynomialToUnivariatePolynomial . . . 688
1.90.61 PowerSeriesLimitPackage . . . 688
1.90.62 PrecomputedAssociatedEquations . . . 689
1.90.63 PrimitiveArrayFunctions2 . . . 689
1.90.64 PrimitiveElement . . . 690
1.90.65 PrimitiveRatDE . . . 690
1.90.66 PrimitiveRatRicDE . . . 691
1.90.67 PrintPackage . . . 691
1.90.68 PseudoLinearNormalForm . . . 692
1.90.69 PseudoRemainderSequence . . . 692
1.90.70 PureAlgebraicIntegration . . . 693
1.90.71 PureAlgebraicLODE . . . 693
1.90.72 PushVariables . . . 694
1.91 Q . . . 694
1.91.1 QuasiAlgebraicSet2 . . . 694
1.91.2 QuasiComponentPackage . . . 695
1.91.3 QuotientFieldCategoryFunctions2 . . . 696
1.91.4 QuaternionCategoryFunctions2 . . . 696
1.92 R . . . 697
1.92.1 RadicalEigenPackage . . . 697
1.92.2 RadicalSolvePackage . . . 697
1.92.3 RadixUtilities . . . 698
1.92.4 RandomDistributions . . . 698
1.92.5 RandomFloatDistributions . . . 699
1.92.6 RandomIntegerDistributions . . . 699

1.92.7 RandomNumberSource . . . 699
1.92.8 RationalFactorize . . . 700
1.92.9 RationalFunction . . . 701
1.92.10 RationalFunctionDefiniteIntegration . . . 701
1.92.11 RationalFunctionFactor . . . 702
1.92.12 RationalFunctionFactorizer . . . 702
1.92.13 RationalFunctionIntegration . . . 703
1.92.14 RationalFunctionLimitPackage . . . 703
1.92.15 RationalFunctionSign . . . 703
1.92.16 RationalFunctionSum . . . 704
1.92.17 RationalIntegration . . . 704
1.92.18 RationalInterpolation . . . 705
1.92.19 RationalLODE . . . 705
1.92.20 RationalRetractions . . . 706
1.92.21 RationalRicDE . . . 706
1.92.22 RationalUnivariateRepresentationPackage . . . 707
1.92.23 RealPolynomialUtilitiesPackage . . . 707
1.92.24 RealSolvePackage . . . 708
1.92.25 RealZeroPackage . . . 708
1.92.26 RealZeroPackageQ . . . 709
1.92.27 RectangularMatrixCategoryFunctions2 . . . 709
1.92.28 RecurrenceOperator . . . 710
1.92.29 ReducedDivisor . . . 710
1.92.30 ReduceLODE . . . 711
1.92.31 ReductionOfOrder . . . 711
1.92.32 RegularSetDecompositionPackage . . . 712
1.92.33 RegularTriangularSetGcdPackage . . . 713
1.92.34 RepeatedDoubling . . . 713
1.92.35 RepeatedSquaring . . . 713
1.92.36 RepresentationPackage1 . . . 714
1.92.37 RepresentationPackage2 . . . 714
1.92.38 ResolveLatticeCompletion . . . 715
1.92.39 RetractSolvePackage . . . 716
1.92.40 RootsFindingPackage . . . 716
1.93 S . . . 717
1.93.1 SAERationalFunctionAlgFactor . . . 717
1.93.2 ScriptFormulaFormat1 . . . 717
1.93.3 SegmentBindingFunctions2 . . . 718
1.93.4 SegmentFunctions2 . . . 718
1.93.5 SimpleAlgebraicExtensionAlgFactor . . . 719
1.93.6 SimplifyAlgebraicNumberConvertPackage . . . 719
1.93.7 SmithNormalForm . . . 720
1.93.8 SortedCache . . . 720
1.93.9 SortPackage . . . 721
1.93.10 SparseUnivariatePolynomialFunctions2 . . . 721
1.93.11 SpecialOutputPackage . . . 722
1.93.12 SquareFreeQuasiComponentPackage . . . 722
1.93.13 SquareFreeRegularSetDecompositionPackage . . . 723
1.93.14 SquareFreeRegularTriangularSetGcdPackage . . . 723
1.93.15 StorageEfficientMatrixOperations . . . 724
1.93.16 StreamFunctions1 . . . 724
1.93.17 StreamFunctions2 . . . 725
1.93.18 StreamFunctions3 . . . 725
1.93.19 StreamInfiniteProduct . . . 726

1.93.20 StreamTaylorSeriesOperations . . . 726
1.93.21 StreamTensor . . . 727
1.93.22 StreamTranscendentalFunctions . . . 727
1.93.23 StreamTranscendentalFunctionsNonCommutative . . . 728
1.93.24 StructuralConstantsPackage . . . 728
1.93.25 SturmHabichtPackage . . . 729
1.93.26 SubResultantPackage . . . 729
1.93.27 SupFractionFactorizer . . . 730
1.93.28 SystemODESolver . . . 730
1.93.29 SystemSolvePackage . . . 731
1.93.30 SymmetricGroupCombinatoricFunctions . . . 731
1.93.31 SymmetricFunctions . . . 732
1.94 T . . . 732
1.94.1 TableauxBumpers . . . 732
1.94.2 TabulatedComputationPackage . . . 733
1.94.3 TangentExpansions . . . 733
1.94.4 TaylorSolve . . . 734
1.94.5 TemplateUtilities . . . 734
1.94.6 TexFormat1 . . . 735
1.94.7 ToolsForSign . . . 735
1.94.8 TopLevelDrawFunctions . . . 736
1.94.9 TopLevelDrawFunctionsForAlgebraicCurves . . . 736
1.94.10 TopLevelDrawFunctionsForCompiledFunctions . . . 737
1.94.11 TopLevelDrawFunctionsForPoints . . . 737
1.94.12 TopLevelThreeSpace . . . 738
1.94.13 TranscendentalHermiteIntegration . . . 738
1.94.14 TranscendentalIntegration . . . 739
1.94.15 TranscendentalManipulations . . . 739
1.94.16 TranscendentalRischDE . . . 740
1.94.17 TranscendentalRischDESystem . . . 740
1.94.18 TransSolvePackage . . . 741
1.94.19 TransSolvePackageService . . . 741
1.94.20 TriangularMatrixOperations . . . 742
1.94.21 TrigonometricManipulations . . . 742
1.94.22 TubePlotTools . . . 743
1.94.23 TwoDimensionalPlotClipping . . . 743
1.94.24 TwoFactorize . . . 744
1.95 U . . . 744
1.95.1 UnivariateFactorize . . . 744
1.95.2 UnivariateFormalPowerSeriesFunctions . . . 745
1.95.3 UnivariateLaurentSeriesFunctions2 . . . 745
1.95.4 UnivariatePolynomialCategoryFunctions2 . . . 746
1.95.5 UnivariatePolynomialCommonDenominator . . . 746
1.95.6 UnivariatePolynomialDecompositionPackage . . . 747
1.95.7 UnivariatePolynomialDivisionPackage . . . 747
1.95.8 UnivariatePolynomialFunctions2 . . . 748
1.95.9 UnivariatePolynomialMultiplicationPackage . . . 748
1.95.10 UnivariatePolynomialSquareFree . . . 749
1.95.11 UnivariatePuiseuxSeriesFunctions2 . . . 750
1.95.12 UnivariateSkewPolynomialCategoryOps . . . 750
1.95.13 UnivariateTaylorSeriesFunctions2 . . . 751
1.95.14 UnivariateTaylorSeriesODESolver . . . 751
1.95.15 UniversalSegmentFunctions2 . . . 752
1.95.16 UserDefinedPartialOrdering . . . 752

1.95.17 UserDefinedVariableOrdering . . . 752
1.95.18 UTSodetools . . . 753
1.95.19 U32VectorPolynomialOperations . . . 753
1.96 V . . . 754
1.96.1 VectorFunctions2 . . . 754
1.96.2 ViewDefaultsPackage . . . 755
1.96.3 ViewportPackage . . . 755
1.97 W . . . 756
1.97.1 WeierstrassPreparation . . . 756
1.97.2 WildFunctionFieldIntegralBasis . . . 756
1.98 X . . . 757
1.98.1 XExponentialPackage . . . 757
1.99 Z . . . 757
1.99.1 ZeroDimensionalSolvePackage . . . 757

Support code **759**

Musings **783**
1.100 Hardware Proof Checking . . . 783
1.100.1 Mealy/Moore equivalence [Fade17] . . . 783
1.101 Specification . . . 784
1.101.1 Binary Search [Guen19] . . . 784
1.102 Parsing . . . 785
1.102.1 Top Down Operator Precedence [Prat73][**?**] . . . 785
1.103 Program proof . . . 788
1.104 Notation . . . 788
1.105 Data Description Language . . . 788
1.106 Interactive Proof Semantics . . . 788
1.106.1 Lean Semantics [Carn19b] . . . 788
1.107 Small Step Operational Semantics . . . 790
1.107.1 Sylvan Clebsch [Cleb19] . . . 790
1.108 Univalence and Homotopy Type Theory . . . 792
1.108.1 Vladimir Voevodsky [Gray18] . . . 792
1.109 Various Logics . . . 793
1.109.1 Barendregt's Lambda cube [Bare92, p78] . . . 793
1.110 Lean and Theorem Proving . . . 794
1.110.1 Jason Rute [Rute20] . . . 794
1.110.2 Robert Y. Lewis [Lewi20] . . . 795
1.110.3 Andrej Bauer [Baue07] . . . 797
1.111 Verbx Regular Expressions . . . 800
1.111.1 methods [verb20] . . . 800

Quote collections **803**

Coq Example Code **808**

Bibliography **817**

Signatures Index **834**

Figure Index **835**

Documentation Index **836**

Code Index **839**

Error Index **843**

Category Index **844**

Domain Index **855**

Package Index **872**

New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation's website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we've broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We've also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I'm looking forward to future milestones.

With that in mind I've introduced the theme of the "30 year horizon". We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How can we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The "30 year horizon" is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))

Consciousness is an emergent phenomenon, like a traffic jam. It requires a certain level of complexity to achieve a certain level of intelligence. Thus, it is not contained within a logic nor bounded by Turing restrictions.
– Tim Daly

Motivation

In the logical traditional fallacy of "appeal to authority" I present quotes from the literature.

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
– Donald Knuth [Knut92]

This literate program contains the actual source code for the new Sane version of Axiom. One of the key goals is to support understanding and long-term maintenance of this new version of Axiom. Key ideas as well as their implementation in code details are discussed.

Computational Mathematics has had two distinct branches, the computer algebra branch and the mathematical logic branch. There has been little crossover between these two fields. Ideally we would have a system where both were combined into a single system.

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.
– John McCarthy [Mcca63]

Axiom has developed in the Computer Algebra branch and, until now, has ignored the mathematical logic branch. The time has come to converge these branches.

This project has the goal of proving Axiom Sane (aka rational, coherent, judicious, and sound). To do that we enhance Axiom with the necessary logic structures (e.g. axioms, judgements, assumptions, etc.) to support formal proofs of the algorithms.

Dijkstra is one of the fathers of a large part of the movement toward formal proofs and program semantics.

The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don't master them, the software crisis will remain with us and will be considered an incurable disease.
– Edsgar Dijkstra [Dijk00]

The missive of creating a world where formal proofs of programs is the norm was given by Dijkstra.

The programmer should not ask how applicable the techniques of sound programming are, he should create a world in which they are applicable. It is his only way of delivering a high-quality design.
– Edsgar Dijkstra [Dijk00]

The challenge we face, and the problem we attack, is constructing this "world" so that Axiom programmers can develop algorithms whose implementations are high-quality, proven code.

There are, of course, "political problems" in the sense that programmers outright reject the very notion of proving programs correct.

> **One reason for preferring symbol-manipulating, calculating arguments is that their design is much better teachable than the design of verbal/pictorial arguments. Large-scale introduction of courses on such calculational methodology, however, would encounter unsurmountable political problems.**
> – Edsgar Dijkstra [Dijk00]

The anti-proof arguments abound and are not novel. Dijstra provides hope (and working examples) that these can be overcome.

> **The first pleasant – and very encouraging! – experience was the killing of the myth that formal proofs are of necessity long, tedious, laborious, error-prone, and what-have-you. On the contrary, our proofs turned out to be short and simple to check, carried out – as they are – in straightforward manipulations from a modest repertoire.**
> – Dijkstra and Scholten [Dijk90]

There is a growing movement to formalize mathematics using computer aided proofs. The Xena Project [Xena19] argues that we need to teach the next generation to develop verified algorithms. The thrust is toward *rigorous mathematics*, that is, mathematics *that is guaranteed to be correct*.

> **The classical engineering disciplines all have their standard mathematical techniques that are applied to the design of any artifact, before it is deployed, to gain confidence about its safety, suitability for some purpose, and so on. The engineers in a discipline more or less agree on what are "the rules" to be followed in vetting a design. Those rules are specified with a high degree of rigor, so that it isn't a matter of opinion whether a design is safe. Why doesn't software engineering have a corresponding agreed-upon standard, whereby programmers convince themselves that their systems are safe, secure, and correct? The concepts and tools may not quite be ready yet for broad adoption, but they have been under development for decades.**
> – Adam Chlipala [Chli17]

Axiom has many algorithms that have mathematical specifications, such as the GCD. Proving that the implementation meets the specification will make the code more reliable.

> **Proofs serve two main purposes. First, proofs provide a way to ensure the reliability of mathematical claims, just as laboratory verification provides a check in the other sciences. Second, the act of finding a proof often yields, as a byproduct, new insights and unexpected new data, just as does work in the laboratory.**
> – Jaffe and Quinn [Jaff93]

There is the fundamental question of which logic to choose. It turns out that this can be decided at the programmer's convenience if we could separate the question of "the logic" from the "machinery" to implement the logic.

As a fellow computational mathematician, Andrej Bauer is reducing a logic system to its fine-grain structure, separating the question of the kernel (aka the nucleus) from the judgements that define the logic.

A Proof Assistant Wish List:

- **small trusted kernel**
- **user-definable dependent type theories**
- **including user-definable judgemental equality**
- **no commitment to an ambient type theory**
- **support common proof-development techniques**

– Andrej Bauer [Baue19a]

We plan to develop the judgements, axioms, and other logic machinery in a fully factored form so that it can be inherited. This allows the definition of a commutative versus a noncommutative domain by simply choosing different categories to inherit. Bauer's idea of separation of the nucleus from the logic allows us to experiement with different forms of reasoning in an integrated way. This achieves one of Milner's goals.

[...] in any realistic work with the machine as a proof assistant we expect to be working in a particular problem domain, or theory as we shall call it [...] it should [...] allow us access to those tactics that we have previously defined either of general use or pertaining to the theory in question. Furthermore, almost every interesting applied theory is founded upon more primitive theories (called its ancestor theories), and while working in any theory we expect to have access to all material pertaining to its ancestors.

An important function of the proof assistant is therefore to keep our tower of theories properly organized, allowing us [...] to work in any existing theory (not only those at the top of the tower) by proving new theorems in them.
– Robin Milner [Miln84]

Axiom supports "First Class Dynamic Types". This allows code to construct and query new types at run time.

Computational Mathematics has two nearly disjoint branches. One branch is proof systems, such as Coq, Agda, and Idris. A second branch is computer algebra systems, such as Axiom, Mathematica, and Maple.

There is a vast literature in each branch. However, the bibliographic references are nearly disjoint.

There are attempts to prove various computer algebra algorithms in proof systems. The dream is to have a full computer algebra system developed in this manner. The key problem is that computer algebra systems are very large, containing many thousands of algorithms.

There are almost no attempts to prove computer algebra systems correct. One key problem is that most systems are not well-founded. They are ad hoc, "it worked for me" development. Another key problem is that most of the proof systems cannot handle fully dependent type theory.

Axiom is unique among computer algebra systems as it has a scaffold based on abstract algebra. Abstract algebra framework gives a solid theoretical base that provides obvious places to provide algebraic axioms.

Extending Axiom to provide these axioms means that they will be inherited just as signatures are inherited. Proofs of algorithms can refer to these axioms.

Axiom needs to be extended to include a specification language so that the algorithms have exact specifications. We will be examining several specification languages, starting with Z [Wood20].

Axiom also needs to be extended with a proof language so algorithms can have associated proofs.

Due to the complexity of these requirements and their need to interact, it is clear that Axiom's compiler and interpreter need to be re-architected and re-implemented.

Because all of these goals are based on type theory we have decided to use the Common Lisp Object System (CLOS). CLOS classes define new types at the Lisp level, providing a base case for type inference.

Type analysis will require both compile-time typing and gradual typing [Bake84].

Hoare [Hoar03] provided a call for a *Grand Challenge* to create a *Verifying Compiler*. This effort falls along the lines suggested but is only an instance in a small domain. On the other hand, Axiom's focus on mathematical algorithms provides the best chance for a successful instance of his challenge.

Homotopy Type Theory (HoTT)[Acze13][Lica16] applies to Axiom, defining paths between the various categories.This is likely to instill discipline on coercions. In the original Axiom systems, coercions were ad hoc constructions. But such coercions in a Homotopy Type Theory fall under the Univalence principle, defining coercions and their inverses. There is, for example, a "curry" path between $A \times B \rightarrow C$ and $A \rightarrow B \rightarrow C$. [Lica16a] This might be usefully applied using generic code to perform the conversion. Exactly how to incorporate this the Sane effort is still to be thought through.

Why this effort won't succeed

DeMillo, Lipton, and Perlis [Demi79] identify what they believe is the key weakness of program verification, that, to increase confidence in the correctness of a program

> **"... the device that verifiers use... is a long chain of formal, deductive logic." Although mathemticians "could in theory" use the same device to increase confidence in theorems "in fact they don't. What they use is a proof, a very different animal... in the end, it is a social process that determines whether mathematicians feel confident about a theorem – and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail"**
> – from Donald Mackenzie [Mack01]

Dijkstra [Dijk78] wrote a reply to their paper.

There is a confusion here, I believe. DeMillo et al. think that the effort to verify software "is not what a mathematician does". We can agree with that. Dijkstra's point of view is that verification increases the confidence in software's correctness. They both seem to get at the heart of the matter, just different hearts. To this point, we do not claim to be "doing mathematics". We claim to be doing program verification.

On the subject of verification DeMillo et al. write

> **There is a fundamental logical objection to verification, an objection on its own ground of formalistic rigor. Since the requirement for a program is informal and the program is formal, there must be a translation, and the translation itself must necessarily be informal.**
> – DeMillo, Lipton, and Perlis [Demi79]

In the context of Axiom, however, the "requirement for a program" is quite formal since we are working with algorithms to implement mathematics with known, formal properties, such as the GCD.

DeMillo et al. also write

> **The formal demonstration that a program is consistent with its specifications has value only if the specifications and the program are independently derived.**
> – DeMillo, Lipton, and Perlis [Demi79]

While we could take issue with this statement in general, it is clear that with Axiom's algorithms, their criteria for "has value" is met.

DeMillo et al. are also confused as to the mechanism of verification. They seem to assume that verification cannot use the results of prior verification. That is, they seem to assume that the "GCD theorem", shown by prior verification, needs to be fully expanded and re-verified at the most primitive level every time. That's a really odd view for a mathematician. Proof checkers don't re-prove everything every time.

Fetzer writes

> **The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.**
> – James H. Fetzer [Fetz88]

Hoare (quoted in Fetzer' paper writes)

> **I hold the opinion that the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived from their specifications through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic.**
> – C.A.R. Hoare [Fetz88]

Fetzer makes an interesting distinction between absolute verification and relative verification.

> **With respect to deductions, the term *verification* can be used in two rather different senses. One of these occurs in pure mathematics and in pure logic, in which theorems of mathematics and of logic are subject to demonstration. These theorems characterize claims that are always true as a function of the meanings assigned to the specific symbols by means of which they are expressed. Theorem-schemata and theorems in this sense are subject to verification by deriving them from no presmises at all (within systems of natural deduction) or from primitive axioms (within axiomatic formal systems). The other occurs in ordinary reasoning and in scientific contexts in general, whenever *conclusions* are shown to follow from specific sets of *premises*, where there is no presumption that these conclusions might be derived from no premises at all or that those premises should be true as a function of their meaning. Thus, within a system of natural deduction or an axiomatic formal system, the members of the class of consequences that can be derived from no premises at all or that follow from primitive axioms alone may be said to be *absolutely* verifiable. By contrast those members of the class of consequences that can only be derived relative to specific sets of premises whose truth is not absolutely verifiable, may be said to be *relatively* verifiable.**
> – James H. Fetzer [Fetz88]

Fetzer claims that programs are subject to relative rather than absolute verification, in relations to rules and axioms in the form of lawful and causal generalizations as premises – empirical claims whose truth can never be established with certainty. The very idea of program verification trades upon an equivocation.

Fetzer does make an interestng distinction between algorithms and programs. A program is an instance of instantiating an algorithm. So the proof of the algorthm and the verification of a program that implements that algorithm are two different tasks.

Ron Pressler has several blog posts about software verification, making some excellent points. He postulates a theorem:

> **There does not exist a generally useful programming language where every program is feasibly verifiable.**
> – Ron Pressler [Pres19]

On the other hand, Doron Zeilberger writes:

> **In this process, computer algebra, that great implementer of high-school algebra, will play a central role. The king (abstract math) is dead. Long live the King (Concrete Mathematics).**
> – Doron Zeilberger [Zeil02]

Maurer, in a letter to ACM in response to the DeMillo article, makes an analogy to compilers

> **Originally, this was done by hand; people wrote out programs in sequences of steps specified informally in English and then proceeded to translate these into machine languages. Then compilers came along, and started to do this job automatically. At first people were against this ... Compilers did what had previously been a fascinating human job in a machine-like, humorless manner. (They also produced overly long object code, in much the same way that a [mechanized] verifier produces overly long proofs.) Nobody is ever going to read the object code produced by a compiler, either; one simply trusts the compiler and goes about one's business. What we hope for in verifiers is that we will at least be able to trust them to show program correctness.**
> – as quoted in [Mack01]

So there is a "storm of misunderstanding" between these various viewpoints. We will try not to claim that this effort is "doing mathematics", only that it is "increasing the trust in the algorithms so they can be used in interactive proof systems".

Make no mistake, this is an enormous task.

It requires a deep understanding of how Axiom implements first-order dependent type theory. Dependent types are undecidable so there have to be heuristics and some user-extensible mechanism for new heuristics.

It requires a deep understanding of first-order dependent types. How can we construct a new dependent type "on the fly" in the interpreter that is type-correct and provable?

It requires a deep understanding of Type Theory. Which form of type theory is used? How can it be factored into categories? How can we show that all inherited paths form sound and complete types?

It requires a deep understanding of the Calculus of Construction with induction. Coq and Lean [Carn19a] implement this form of logic. They are both at the leading edge of the field and still under active development and debate.

It requires a deep understanding of Program Verification. Coq and Lean do not (yet) have libraries that directly support program semantics. Nor does the Spad language (yet) have well-defined semantics.

It requires proving algorithms that may never have been proven, may not be correct, and may not terminate under all inputs. Further, many of the algorithms have no specification.

It requires constructing a well-typed compiler and a well-typed interpreter. The compiler needs to parse both the Spad language and the new logical extensions.

It requires validation and verification of the well-typed-ness of the compiler and interpreter, both of which are implemented in CLOS on top of Common Lisp.

It requires constructing the whole tower on a trusted kernel of logic which itself might sit on a trusted nucleus that can adapt to a range of logics. This must be verified and validated "to the metal".

The question of what constitutes "real proof" is not quite as settled as most would believe. MacKensie [Mack01] published a sociology study of the question and points out that there are various cohorts within the mathematics community that do not agree that machine-based proofs are "proofs".

DeMillo et al. makes a reasonably valid point though

> **The lone fanatic might construct his own verification, but he would never have any reason to read anyone else's, nor would anyone else ever be willing to read his. No community could develop. Even the most zealous verifier could be induced to read a verification only if he thought he might be able to borrow or swipe something from it. Nothing could force him to read someone else's verification once he had grasped the point that no verification bears any necessary connection to any other verification.**
> – DeMillo, Lipton, and Perlis [Demi79]

But since "verification" is the "assembly language" of the system this is hardly a criticism. Proof systems will read the verification, just like CPUs "read the assembly language".

The key "transition paper" from these arguments seems to be Martin-Löf's "Constructive Mathematics and Computer Programming" paper [Mart79]. To quote the abstract

> **If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute (a difference that Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics. This explains why the intuitionistic theory of types, which was originally developed as a symbolism for the precise codification of constructive mathematics, may equally well be viewed as a programming language. As such it provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Moreover, the inference rules of the theory of types, which are again completely formal, appear as rules of correct program synthesis. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.**
> – Per Martin-Löf [Mart79]

Robert Harper uses it as the basis for his Computational Type Theory Course [Harp18].

1.1 Intellectual Complexity

There is a considerable gap between writing a program and proving the program correct. Languages like Coq, and even the terms withing Coq such as "inductive", and "constructor" are not normally part of the programmer's world.

There is a very slow convergence of syntax in languages like ML and Haskell but their emphasis is on type theory as a "checking" bit of machinery rather than on type theory as a "proof" bit of machinery.

1.1.1 Alturki and Moore [Altu19]

Consider a simple example of a Coq program proof.

We need to specify three things:

- **L**, a formal model of the syntax and semantics of the programming language

- **P**, a formal representation in model **L**
- **S**, a specification of the property to be verified

1.1.2 An example

See the example code in the appendix 1.111.1.

Given a program

```
sum = 0;
while (!(n <= 0)) {
  sum = sum + n;
  n = n + -1;
}
```

we need to have

- a formal model of the semantics of the language used
- a precise specification of the program structure
- a specification that this computes the sum of integers from 1 to n.

1.1.3 Specification of the example in Coq

In Coq an arithmetic expression can be defined as:

```
Inductive AExp :=
  | var  : string -> AExp
  | con  : Z -> Aexp
  | plus : Aexp -> AExp -> AExp
```

where variables use constructor **var** applied to a string, constants use constructor **con** applied to an integer (Z is the set of integers), and the **plus** constructor takes 2 **AExp** arguments and returns a **AExp** result.

Programs are defined as

```
Inductive Pgm :=
    pgm : list string -> Stmt -> Pgm.
```

The summation program above become essentially an abstract syntax tree

```
Definition sum_pgm N : Pgm :=
 pgm ["n"; "sum"]
(seq (assign "n" (con N))
(seq (assign "sum" (con 0))
     (while (not (le (var "n") (con 0)))
     (seq (assign "sum" (plus (var "sum") (var "n")))
          (assign "n" (plus (var "n") (con (-1)))))))).
```

1.1.4 Semantics of the example in Coq

Note that we are using operational semantics here.

The semantics of arithmetic expressions is captured by a binary transition relation on states, where a state is a pair **(AExp, Env)** consisting of an arithmetic expression **AExp** and the environment **Env** in which **AExp** is to be evaluated.

This binary transition relation can be defined in Coq as an inductively defined proposition (of type Prop), **step_e**, that tells us whether a given pair of states belongs to the relation.

```
Inductive step_e : (AExp * Env) -> (AExp * Env) -> Prop :=
  | step_var: forall v x env, get v env = Some x ->
      step_e (var v, env) (con x, env)
  | step_plus: forall x y env,
      step_e (plus (con x) (con y), env) (con (Z.add x y), env)
  | cong_plus_r: forall e1 e2 e2' env env',
      step_e (e2, env) (e2', env') ->
      step_e (plus e1 e2, env) (plus e1 e2', env')
  | cong_plus_l: forall e2 e1 e1' env env',
      step_e (e1, env) (e1', env') ->
      step_e (plus e1 e2, env) (plus e1' e2, env')
```

The **step_var** states that the program variable **var v** evaluates in one step to its value **con x** in the environment **env** (**x** is the integer value obtained from the function application **get v env**, the value to which **v** is mapped in the **env**, and this holds for all program variables and environments (recall that **con** is the constructor for integer constants). Another example is the **step_plus** case, which states that any arithmetic expression of the form

```
   plus (con x) (con y)
```

evaluated to constant values, evaluates in one step to the value that is the sum of the operands, regardless of the current environment (**Z.add** is the integer addition operation). Note how **step_plus**, **cong_plus_r**, and **cong_plus_1** collectively specify call-by-value semantics of **plus**.

1.1.5 Testing execution

Using the Coq definition of **sum_pgm** above, we can test that **sum_pgm 100** produces the expected result, 5050.

```
Lemma test_execution :
  clos_relf_trans_1n _ step_p
    (sum_pgm 100,[])
    (pgm nil skip, [("sum",5050);("n",0)]).
```

An execution trace is constructed by the reflexive-transitive closure of the **step_p** relation (see 1.111.1.), computed by Coq's inductively defined proposition **clos_refl_trans_1n**. Note that the program in the final state is the command **skip**, requiring that the program be fully executed.

Completing the proof requires picking the right clause of the step relation for each step of execution.

1.1.6 Specifying correctness goals

We need to formalize the statement of the property we wish to verify as a reachability property. Following the language independent approach of program verification by coinduction [Moor18], we define a program property as a reachability claim on program states of the form **(Pgm * Env)**. In particular, using the coinductive proof machinery (see 1.111.1), the main correctness property of **sum** can be defined as the following inductive proposition (recall that the quoted strings "n" and "sum" are the program variables, while **n** is a universally quantified logical name denoting a symbolic value)

```
Inductive sum_spec : Spec (Pgm * Env) :=
 | sum_claim: forall n, 0 <= n ->
   sum_spec
     (pgm ["n"; "sum"]
       (seq (assign "n" (con n))
       (seq (assign "sum" (con 0))
            (while (not (le (var "n") (con 0)))
            (seq (assign "sum" (plus (var "sum") (var "n")))
                 (assign "n" (plus (var "n") (con (-1))))))))
     ,[])
     (fun cfg' => cfg' = (pgm [] skip, [("sum",((n + 1) * n)/2);("n",0)]))
```

Spec is the general type of (language independent) program specifications (reachability claims), instantiated here with states **Spec (Pgm * Env)**. The specification states that for all non-negative values **n**, the initial pair consisting of the program **sum** and the empty environment **[]** reaches a state having the empty program **skip** as its first component and an environment in its second component that maps the program variables "n" to zero and "sum" to the value of the expression **((n + 1) * n)/2**.

Other properties, e.g. the loop invariant for **sum**, may also need to be defined in addition to the main correctness property. In Coq, these properties on program states, or instantiated **Spec** propositions in the case when the coinductive program verification approach described here is used. Coq tends to be more verbose requiring having other, smaller, properties specified and proved including correctness properties of environment manipulation operations.

1.1.7 Verifying correctness

In a language verification framework, deductive program verification entails showing that a property **S**, like the main correctness property of **sum**, follows logically from a given model **L** of the language and a model **P** of the program, along with other smaller intermediate properties that have already been show to hold in **L** and **P**.

Following the language-independent coinductive verification approach (see [Moor18], a program is shown to meet its specification by (1) using the step relation defining the semantics of the language in which the program is written as a symbolic execution engine, and (2) using simple co-inductive reasoning on circular behaviors emanating from recursive and iterative constructs in the language (such as loops). The machinery specified in Coq allowing the reachability logic proof strategy can be found in the appendix (see 1.111.1).

To prove correctness we need to specify and show the loop invariant. For that, we introduce another constructor of the type **sum_spec**

```
 | sum_loop_claim : forall env n, get "n" env = Some n -> 0 <= n ->
                    forall s, get "sum" env = Some s ->
      sum_spec
         (pgm []
           (while (not (le (var "n") (con 0)))
             (seq (assign "sum" (plus (var "sum") (var "n")))
                  (assign "n" (plus (var "n") (con (-1))))))
         ,env)
      (fun cfg' => fst cfg' = pgm [] skip /\
         snd cfg' = set "n" 0 (set "sum" (s + ((n + 1) * n)/2) env)).
```

Note that **get** $\times$ **env** evaluates to the value to which x is mapped in **env** (if defined), while **fst** and **snd** return respectively the first and second components of a state pair.

To verify that **sum** meets its specification given by **sum_spec**, which is that the initial state of **sum** either diverges or terminates with the correct value for **sum**, we instantiate the general coinductive soundness theorem with the step relation **step_p** defining the semantics, and show the following statement:

```
Lemma sum_ok : sound step_p sum_spec.
```

sound is a generic proposition on reachability claims. The proof of this instantiation is based on a generalized co-induction theorem that mechanizes the verification approach.

Related Work

This is a "wide" project that touches on many areas. Here we try to collect and mention papers that we've read. A good overview of the historical development is given in MacKenzi [Mack01].

1.2 Type Theory

1.2.1 Harper [Harp18][Harp18a]

Types are Specifications of Behavior.

A Type – is a specification of behavior

$M \in A$ – M is a program that has A behavior

Both M and A are programs.

if(Nat,Bool)(M) is a type when M $\in$ Bool

if(17,true)(M) $\in$ if(Nat,Bool)(M)

type-indexed family of types aka dependent types

$f \in n : Nat \rightarrow Seq(n)$ is the same as $(\Pi n : Nat.Seq(n))$

Critical Idea: Functionality. Families of types and families of elements must respect *equality* of indicies. The key question is, "What is equality".(eq Seq(2+2) is the "same as" Seq(4)). Also

Seq(if(17;18)(M)) is the "same as" if(Seq(17);Seq(18))(M)

if $M \doteq M' \in A$ and $A \doteq A'$ then $M \doteq M' \in A'$ that is, if M and M' satisfy the specification A and the specification A and A' are equivalent then M and M' satisfy the specification A'.

$A \Downarrow A_0$ where A_0 is always a final value.

$A \doteq A'$ means if $A \Downarrow A_0$ and $A' \Downarrow A'_0$ and $A_0 \doteq_0 A'_0$ are equal type values.

If A is a type (meaning $A \Downarrow A_0$ and $A_0 \doteq_0 A_0$) then $M \doteq M' \in A$ means $M \Downarrow M_0$ and $M' \Downarrow M'_0$ and $M_0 \doteq_0 M'_0 \in A_0$. That is, these are equal-values in a type-value.

Functionality: $a : A >> B \doteq B'$ means if $M \doteq M' \in A$ then $B[M/a] \doteq B'[M'/a]$

$a : A >> N \doteq N' \in B$ means if $M \doteq M' \in A$ then $N[M/a] \doteq N'[M'/a] \in B[M/a]$

1.3 Other Compilers

There are other ongoing efforts to create compilers along with an association to proofs.

1.3.1 Hoare [Hoar03]

Tony Hoare makes a call for verifying compilers which is, as he calls it, a "Grand Challenge".

1.3.2 Pearce and Groves [Pear15]

Pearce and Groves created a language, called Whiley, specifically designed to be easier to prove. They have a list of design decisions.

1. Functions are pure, whilst methods maybe impure
2. Compound data types have value (copy) semantics
3. Arithmetic is unbounded
4. All specifications are checkable at runtime
5. Type checking is separate from verification

They introduce keywords like `requires` and `ensures`. For example,

```
function decrement(int x) -> (int y)
// Parameter x must be greater than zero
requires x > 0
//Return must be greater or equal to zero
ensures y >= 0:
  return x - 1
```

The associated verification conditions is

$$\forall x \in int.(x > 0 \Rightarrow x - 1 \geq 0$$

Whiley supports loop invariants by decorating loops with a `where` clause.

```
Function sum(int[] items) -> (int r)
// Every item in items is greater or equal to zero
requires all ( i in 0..|items| | items[i] >= 0
// Return must be greater or equal to zero
ensures r >= 0:
  int r = 0
  int i = 0
  while i < |items| where r >= 0 && i >= 0
    r = r + items[i]
    i = i + 1
  return r
```

Whiley has an `assert` and `forall`

```
assert:
  forall(int x0, int x1):
    if:
      x0 < 0
      x1 == -x0
    then:
      x1 >= 0
```

Whiley distinguishes pure code using the keyword `function` from impure code using the keyword `method`.

The Problem

1.4 A Brief History

Axiom[Daly17] is a computer algebra system. It was originally developed at IBM Research based on a proposal by James Griesmer[Blai70a] in 1965, led to fruition by Richard Jenks[Dave84]. The project was called Scratchpad. A more complete history is available in the online documentation.

Around this time the ideas of computational mathematics (CM) were “in the air”. There was the beginning of a branch of computer algebra (CA) with developments, as descibed by Joel Moses [Mose71], like SAINT and SIN. Anthony Hearn [HEAR80] created REDUCE.

A second branch of computational mathematics was the development of automated proof systems (PS). Early work in the area included the idea, by Robinson [Robi65], “resolution”, leading to resolusion-based theorem proving. De Bruijn [Brui68a] started work on AUTOMATH.

Scratchpad grew out of the computer algebra branch of computational mathematics.

Though they are both areas of computational mathematics, issues such as equality are treated differently. Davenport [Dave02] addresses the issue of equality and its several ambiguities.

Homotopy Type Theory [Acze13] has the univalence axiom which defines a form of equality based on paths.

1.5 Parallel Development

1.6 Project Goals

Computational mathematics has had intense development efforts on both branches. But the two branches had very little overlap. [Daly18].

Axiom was released as a free and open source project. One of the obvious goals, based on the observation that Computer Algebra is a sub-area of Computational Mathematics, would be to “prove Axiom correct”.

This goal languished for many years. There were other goals that had higher priority. There were various attempts at discussion on the mailing list but no action.

1.7 Visiting Scholar

Around 2014 I started reading papers in the proof systems area. This was extremely difficult because, after so many years of independent development, the proof community had developed a language and notation that was opaque to the unskilled. The language of Judgments and Rules, written in what appears to the outsider as classical greek, required a great deal of education and training to read.

One major impediment was that all of the interesting research papers were published in paywalled journals. A 4 page paper costs 40 dollars. I needed access to hundreds of papers. This made it nearly impossible to do research without a University connection.

In 2015 I read a paper by Jeremy Avigad [Avig12a]. Since he was at Carnegie Mellon, where I had previously worked, I called him to discuss his paper. It was clear we had the same mathematical ideas but no common ground for reducing them to practice. We were from different branches of computational mathematics.

Jeremy invited me to audit a class he was developing around the proof system LEAN [**?**]. While probably a frustrating experience for Jeremy, I found the class enlightening. I finally had a glimpse of the notation and thought process behind proof systems.

Fortunately, around the beginning of 2016, I started "dropping onto campus" at Carnegie Mellon University. I sat at a hallway table in the Computer Science department several days a week. I had hoped to use the local network to access research papers but I needed to be an active member of the University.

By chance, Frank Pfenning [**?**] walked by my table on a way to a meeting. He didn't have time to talk but said I should make an appointment with this secretary.

I developed a "beg cycle" presentation about the parallel development of computational algebra and proof systems, ending with a request to have library access by any possible means. Frank arranged for me to be a "visiting scholar", sponsored by Klaus Sutner [**?**].

Now, as a University person, I could access and read leading edge research papers. It made all the difference.

Several professors at CMU allowed me to audit courses, including Karl Crary [**?**], Frank Pfenning, Andre Platzer [**?**], and others.

1.8 Refining the Project Goals

I spent a "survey year" reading papers in computational mathematics, trying to understand the areas of overlap. Eventually it was clear that the subjects were nearly disjoint. Following the bibliography of proof system papers generated one pile. Doing the same in computer algebra generated another pile. There was hardly a name that occurred in both piles.

It also became clear that "Proving Axiom Correct" was the wrong view of the problem. Axiom had no global specification and without that it made no sense to talk about "Correct".

By 2018 I refined Axiom's project goals. The new goal was to "Prove Axiom Sane". Sane's thesaurus lists synonyms like "rational", "coherent", "judicious", and "sound", all words used by the proof system researchers.

"Proving Axiom Sane", the crossover project goal of uniting computer algebra and proof systems, was the theme originally presented to Frank Pfenning. It was now refined into a talk given at the International Conference on Mathematics Software [**?**] in August of 2018.

1.9 Reduction to Practice

By late 2018 the project goal was more specific. The goal was now limited to the proving the many Greatest Common Divisor (GCD) implementations in Axiom correct. There are GCD algorithms for non-negative integers and GCD algorithms for things like polynomials. Clearly there was a specific specifications for these instances. There is also a specification of the general GCD algorithm, so the notion of correctness made sense.

But there was the question of automation. Axiom needed a language to express the specifications. It needed a language to express the semantics of the implementation of GCD. It needed axioms that expressed the properties of statements in the specification, such as the commutativity axiom. It needed to hierarchically "gather" axioms that apply from the type hierarchy (called Categories in Axiom) as well as axioms specific to the instance (called a Domain in Axiom).

Beyond the language, Axiom needed a way to create and manipulate the associated proofs. It needed a means to access existing proof systems to carry out the verification of the proof correctness. The compiler needs access to a proof checker to validate the proof at compile time.

There was much to be done.

1.10 Subgoal: A new implementation

Axiom is a large (about 1.2 million line) system implemented in Common Lisp. The code base grew "by accretion", originally from a MacLisp [**?**] base, ported to Lisp/VM [Dave85], and then ported to Common Lisp [Stee90]. Since it was born as a research project there was no real effort to make it into a "product" until it was sold to The Numerical Algorithms Group (NAG) [IBMx91].

The net result is that the original code base was not really a good foundation for full computational mathematics. But Axiom has several thousand working functions, many derived from PhD thesis work, that could not practically be duplicated from scratch.

So the first step toward a Sane Axiom system would be to restructure the compiler and interpreter "from the ground up" with the requirement that it could execute the existing algorithms.

After much pondering, the design decision would be to use the Common Lisp Object System (CLOS) [**?**] as the new basis for Axiom.

One reason for this choice is that the base is still Common Lisp so it would be possible to reuse existing pieces of code.

Another reason is the CLOS objects are considered "types" so they could be used in the proof system style of type checking. This gives a way to introduce type checking at the Lisp level, below the Axiom Category and Domain abstraction. It also gives a firm basis for reasoning about dependent types.

An Inside-Out Approach

The traditional approach to compiler development starts with a BNF [**?**] definition of the language syntax, followed by an intermediate representation, and then one of several "back end" implementations of the runtime.

Instead, a new approach evolved. It would be possible to re-cast the Axiom Category and Domain hierarchy as CLOS types. Furthermore, since the CLOS types are automatically checked for precedence at compile time, problems in inheritance are caught early.

CLOS classes can be hand-written. This means that the usual intermediate compiler language could actually be read, constructed, and modified by hand.

Constructing Axiom in CLOS "from the inside" means that the surface syntax of Axiom, called Spad (short for Scratchpad), could be created later with the clear target of a working intermediate.

Since the intermediate CLOS code needed to support the surface language it should be possible to automatically create the surface language from the CLOS. That makes it clear that the surface language was fully supported by the CLOS implementation.

It also means that the traditional parser could be developed later in the project with a very clear target language that fully supports all of the surface language features.

1.11 Conventions

Everything has two kinds of types. There are CLOS types, such as those which handle the parsed input. There are SPAD types, such as those that reference spad categories.

1.11.1 The CLOS types

The CLOS types have generic functions to handle many operations. There are, in general, two generic functions for each CLOS type.

One is the **print-object** generic method. It outputs the CLOS object as top-level SPAD syntax. The idea is that the output of the parser should mirror the input to the parser but derived from the internal structures. That provides a "round-trip" check on the parser correctness.

Each **print-object** has calls to the **format** function. The command string for each format call references a global variable. This allows any print-object to be modified at run time to change the output syntax.

Each CLOS type has a make-closname function that creates an instance with required and actual parameters.

A second generic function **intern-object** is available for all CLOS classes. It outputs an intermediate representation of the CLOS object for further processing. Like print-object generics, each **format** function command string references a global variable which can be modified at run time. This allows the intern-object to be adjusted easily.

Extensive use is made of the common lisp **labels** form to define single-use functions locally. This reduces the namespace pollution and makes local function definitions able to be inlined by the compiler. Many of these single-use functions exist just to make the logic of the function easier to understand. In particular, local predicates are used to name the thing being checked.

1.11.2 The SPAD types

The SPAD types from the Axiom distribution are included here as CLOS classes. These form a basic library of objects available at the lisp level.

The design allows for the library to be modified, extended, or replaced completely. All that is necessary about the SPAD language should be independent of the categories, domains, and packages available to the user.

Certain "fundamental" libraries need to "exist before they exist" and are essentiall built in to the system. However, they are built-in at the library level, not at the compiler level. The compiler expects them to exist but does not hard code them.

One of the challenges of building a SPAD library in CLOS is that CLOS checks the inheritance hierarchy. This has the advantage of a consistancy check but it is at times painful to get right.

Spad types, called **Union**, are logical 'sum types'.

> The values of a sum type are typically grouped into several classes, called *variants*. A value of a variant type is usually created with a quasi-functional entity called a *constructor*. Each variant has its own constructor, which takes a specified number or arguments with specified types. The set of all possible values of a sum type is the set-theoretic sum, i.e. the *disjoint union*, of the sets of all possible values of its variants. *Enumerated types* are a special case of sum types in which the constructors take no arguments, as exactly one value is defined for each constructor.
> – Wikipedia [Wiki19]

Spad types, called **Record**, are logical 'product types'.

> The values of a product type typically contain several values, called *fields*, All values of that type have the same combination of field types. The set of all possible values of a product type is the set-theoretic product, i.e. the *Cartesian product*, of the sets of all possible values of its field types.
> – Wikipedia [Wiki19]

1.12 AxiomClass

Axiom has Categories, not the same as the categories of category theory. The Categories are essentially types from type theory.

In Common Lisp the type hierarchy has a most general type 't'. In CLOS we define a new type **AxiomClass** to be the most general type of our hierarchy. AxiomClass inherits methods from 't', such as **print-object**.

AxiomClass provides the base definition for slots that every Category, Domain, and Package will have. Not all of them are in use for every case but this provides a place to add new features that work for every class.

AxiomClass provides **print-object** for the classes. The general case print-object will construct and print the source code that corresponds to the defclass definition. So, as if by magic, we can build the source code from the intermediate CLOS defclass.

This leads to the idea of building the compiler 'inside out'. Normally one would define the language syntax, write a parser, construct an intermediate abstract syntax tree (AST) that holds the result of the parse, and then construct a back end to a target machine.

However, CLOS is human readable and constructable. So we have taken the novel approach of writing the AST in CLOS and 'deriving' the source code from the AST. Later construct a parser that will accept the output of 'print-object', parse it, and reconstruct the CLOS defclass.

This means that we have a 'round trip' test of the parser. Every defclass prints its source representation. The parser must recreate the defclass from the printed source.

Refer to the AxiomClass 1.15.3 section for more detailed information.

1.13 Categories

We need to construct the Axiom Category (aka type) hierarchy in CLOS. We will use the term 'type' to signify Axiom objects, since the defclass definitions are valid Common Lisp types.

We can do this because CLOS support inheritance.

Axiom can print the list of ancestors for a given type. For example, PolynomialCategory ancestors are:

```
getAncestors 'PolynomialCategory

   (3)
   {AbelianGroup, AbelianMonoid, AbelianMonoidRing,
    AbelianSemiGroup, Algebra, BasicType, BiModule,
    CancellationAbelianMonoid, CharacteristicNonZero,
    CharacteristicZero, CoercibleTo, CommutativeRing,
    ConvertibleTo, EntireRing, Evalable, FiniteAbelianMonoidRing,
    FullyLinearlyExplicitRingOver, FullyRetractableTo, GcdDomain,
    InnerEvalable, IntegralDomain, LeftModule, LeftOreRing,
    LinearlyExplicitRingOver, Module, Monoid, OrderedSet,
    PartialDifferentialRing, PatternMatchable,
    PolynomialFactorizationExplicit, RetractableTo, RightModule,
    Ring, Rng, SemiGroup, SetCategory, UniqueFactorizationDomain}
                                                      Type: Set(Symbol)
```

PolynomialCategory is quite high up in the type hierarchy (15 levels up, actually) so most of these are already inherited. We provide a program (called **leaves**), that given the above list will compute the types that are not inherited. For example, given the above Axiom output,

```
(leaves '(|AbelianGroup| |AbelianMonoid| |AbelianMonoidRing|
    |AbelianSemiGroup| |Algebra| |BasicType| |BiModule|
    |CancellationAbelianMonoid| |CharacteristicNonZero|
    |CharacteristicZero| |CoercibleTo| |CommutativeRing|
    |ConvertibleTo| |EntireRing| |Evalable| |FiniteAbelianMonoidRing|
    |FullyLinearlyExplicitRingOver| |FullyRetractableTo| |GcdDomain|
    |InnerEvalable| |IntegralDomain| |LeftModule| |LeftOreRing|
    |LinearlyExplicitRingOver| |Module| |Monoid| |OrderedSet|
    |PartialDifferentialRing| |PatternMatchable|
    |PolynomialFactorizationExplicit| |RetractableTo| |RightModule|
    |Ring| |Rng| |SemiGroup| |SetCategory| |UniqueFactorizationDomain|))

(|ConvertibleTo| |Evalable| |FiniteAbelianMonoidRing|
 |FullyLinearlyExplicitRingOver| |OrderedSet| |PartialDifferentialRing|
 |PatternMatchable| |PolynomialFactorizationExplicit|)
```

The PolynomialCategory class only needs to inherit the types that are listed by the leaves program.

CLOS computes the precendence of types in the hierarchy. This provides assurance that the Axiom category hierarchy is, at least at a minimum, reasonably constructed.

1.14 Domains and Packages

Domains add a representation (called a "carrier" in logic).

In addition to the axioms inherited from the category hierarchy, domains add additional axioms to characterize the properties of the representation. These additional axioms contribute to the proof.

A Specification Language

If it compiles, it works *as specified***.**
– Vladislav Zavialov [**?**]

Much of the essence of building a program is in fact the debugging of the specification. – Fred Brooks [Warn16]

Real programming languages are inevitably complex, and any serious attempt to give a formal treatment of such a language and a development framework based on it is an ambitious undertaking bringing a host of problems that do not arise when considering toy programming languages or when considering specification and formal development in abstract terms. Our EML experience suggests that, at least at the present time, tackling the problems of specification and formal development in a real programming language at a fully formal level is just too difficult
– Donald Sannella and Andrzej Tarlecki [Sann99]

We must distinguish the language used for specification (the *what* from the language for implementation the *how*.

Sannella, et al. [Sann91, Sann99] develop a specification language for Standard ML (SML) [Miln90, Miln91]

Kahrs and Sannella [Kahr98] designed EML to provde a specification language for ML. They point out that they want a formal connection between the language (P) and the specifications (S). They write:

> Given that aim, it is not possible to come up with a meaningful specification language for P unless P has a formal semantics. Without a formal semantics for P we are not certain what P-programs are supposed to do, making it impossible to establish reliably any property of any P-program or to prove interesting relationships between P-programs and S-specifications. Unfortuately, this requirement rules out most present-day programming languages.

The Axiom Sane effort is trying to design a specification language for the mathematics implemented by Axiom, not a specification language for Axiom's Spad language.

The notion of "exceptions" in mathematics is unclear. Since Axiom implements mathematical functions it is not much of a concern. Side-effects are also not much of a concern, at least for the mathematical algorithms.

As such, the Sane effort is a "domain specific specification", not a "language specific specification". If there is a Spad language construct used to implement a function, all one has to show is that the language construct "covers" the use case in the particular function.

They raise a series of question about a specification [Sann99]

1. What is a specifiation?
2. What does a specification mean?
3. When does a program satisfy a specification?
4. When does a specification guarantee a property that it does not state explicitly?
5. How does one prove this?
6. How are specifications structured?
7. How does the structure of a specification relate to the structure of programs?
8. When does one specification correctly refine another specification?
9. How does one prove correctness of refinement steps?
10. When do refinement steps compose?
11. What is the role of information hiding?

The specification language is related to what we wish to prove. [Kahr98]. Is it a

1. proof that a given program satisfies a given specification?
2. proof that one specification is a refinement of another?
3. proof that all programs satisfying a given specification will satisfy a given property?

John Hughes [Hugh19] identifies several types of specification tests:

1. invariant properties
2. postconditions
3. metamorphic properties
4. inductive properties
5. model-based properties

1.14.1 The specification class

— defclass specification —

```
(defclass specification ()
  ((precondition  :initarg :precondition  :initform nil :accessor spec-precondition)
   (postcondition :initarg :postcondition :initform nil :accessor spec-postcondition)))
```

——–

Primitive Support tools

Due to type checking, these functions have to appear early in the file but we do it by using the literate reference.

Overview

Data are just dumb programs
– Dan Friedman

The classic hacker disdain for "bondage and discipline languages" is short sighted – the needs of large, long-lived multi-programmer projects are just different than the quick work you do for yourself.
– John Carmack

If you have a large enough codebase, any class of error that is syntactically legal probably exists there.
– John Carmack

Anything that isn't clear to your static analysis tool probably isn't clear to your fellow programmers either.
– John Carmack

A lot of the serious reported errors are due to modifications of code long after it was written.
– John Carmack

The first step is fully admitting that the code you write is riddled with errors. That is a bitter pill to swallow for a lot of people, but without it, most suggestions for change will be viewed with irritation or outright hostility. You have to want criticism of your code.
– John Carmack

The ***Categories*** variable contains a list of all defined Categories.

— defvar categories —

```
(defvar *Categories* nil)
```

———-

The ***Domains*** variable contains a list of all defined Domains.

— defvar domains —

```
(defvar *Domains* nil)
```

———-

The ***Packages*** variable contains a list of all defined Packages.

— defvar packages —

```
(defvar *Packages* nil)
```

———–

— defvar indent —

```
(defvar *indent* "    ")
```

———–

1.15 The Category Class

[Dosr11, p4] [Daly17]

- **marker** is a constant to know if something is a category, a domain, or a package.
- **parents** is a list of the direct parents of this category.
- **CategoryForm** holds the canonical category form of the expression whose evaluation produces the category object under consideration
- **ExportInfoList** holds a list of function signatures exported by the category
- **AttributeList** holds a list of attributes and the conditions under which they hold
- **CategoryMark** always contains the form (Category). It serves as a runtime type checking tag
- **PrincipalAncestorList** is a list of principal ancestor category forms
- **ExtendedCategoryList** is a list of directly extended category forms
- **DomainInfoList** is a list of domains explicitly used in that category
- **UsedDomainList** holds the list of all domain forms mentioned in the exported signatures

1.15.1 The hasclause class

— defclass hasclause —

```
(defclass hasclause ()
  ((whichtype :initarg :whichtype :initform nil  :accessor hasclause-whichtype)
   (condition :initarg :condition :initform nil  :accessor hasclause-condition)
   (siglist   :initarg :siglist   :initform nil  :accessor hasclause-siglist)))
```

———–

make-hasclause : **(type,condition,siglist)** → **hasclause**
— defun make-hasclause —

```
(defun make-hasclause (type condition siglist)
  (make-instance 'hasclause :whichtype type :condition condition :siglist siglist))
```

———-

— defmethod print-object —

```
(defmethod print-object ((clause hasclause) stream)
  (let ((deeper (concatenate 'string "  " *indent*)))
    (format stream "if ~a has ~a then~%~a~a~%"
      (hasclause-whichtype clause)
      (hasclause-condition clause)
      deeper
      (hasclause-siglist clause))))
```

———-

1.15.2 The haslist class

— defclass haslist —

```
(defclass haslist ()
  ((clauses :initarg :clauses :initform nil :accessor haslist-clauses)))
```

———-

make-haslist : **clauses** → **hasclause**
— defun make-haslist —

```
(defun make-haslist (clauses)
  (make-instance 'haslist :clauses clauses))
```

———-

1.15.3 The AxiomClass class

The **AxiomClass** is a place to hang Axiom-specific methods.
— defclass AxiomClass —

```
(defclass |AxiomClass| ()
  ((marker :initform 'category :accessor marker :initarg :marker)
   (name :initform "AxiomClass" :reader name :allocation :class)
   (parents :initarg :parents
            :accessor parents
            :initform nil
            :type (or null list)
            :documentation "A list of direct parents")
   (level        :initarg :level        :reader   level)
   (abbreviation :initarg :abbreviation :reader   abbreviation)
   (macros       :initarg :macros       :accessor macros   :initform nil)
   (comment      :initarg :comment      :accessor comment  :initform nil)
   (argslist     :initarg :arg          :accessor argslist :initform nil)
   (withlist     :initarg :with         :accessor withlist :initform nil)
   (haslist      :initarg :has          :accessor haslist  :initform nil)
   (addlist      :initarg :add          :accessor addlist  :initform nil)))
```

———-

The print-oject method for AxiomClass re-creates the source level language from the internal data structures. The output of print-object should exactly mirror the input syntax. This allows "round-trip" assurance that the source code and its intermediate representation match.

Format control strings are used to create output from the print-object function. Because we may want to (dynamically?) change the output of any AxiomClass object, we provide the control strings as global variables.

— defmethod print-object —

```
(defvar formatCategoryAbbrev "~&)abbrev category ~a ~a")
(defvar formatCategorySIG "~&~a(~{~a~^, ~}) : Category == SIG where~2%")
(defvar formatCategoryComment "~&~{~^++ ~a~%~}~%")

(defvar formatDomainAbbrev   "~&)abbrev domain ~a ~a")
(defvar formatDomainSIG      "~&~a(~{~a~^, ~}) : SIG == CODE where~2%")
(defvar formatDomainComment  "~&~{~^++ ~a~%~}~%")

(defvar formatPackageAbbrev  "~&)abbrev package ~a ~a")
(defvar formatPackageSIG     "~&~a(~{~a~^, ~}) : SIG == CODE where~2%")
(defvar formatPackageComment "~&~{~^++ ~a~%~}~%")

(defvar formatNoSuper        "  SIG ==> () ")
(defvar formatSuper          "  SIG ==> ~a ")
(defvar formatJoin           "  SIG ==> Join(~{~a~^,~%               ~}) ")

(defvar formatWith           "with ~%")
(defvar formatWithNil        "with nil~%")

(defvar formatSig            "    ~a~%")

(defvar formatHas            "~%~aif % has ~a then~%")

(defvar formatAdd            "  add ~%")

(defmethod print-object ((cat |AxiomClass|) stream)
  (let (has condition)
   (cond
     ((eq (marker cat) 'category)
       (format stream formatCategoryAbbrev (abbreviation cat) (name cat))
       (format stream formatCategoryComment (comment cat))
       (format stream formatCategorySIG (name cat) (argslist cat)))
     ((eq (marker cat) 'domain)
       (format stream formatDomainAbbrev (abbreviation cat) (name cat))
       (format stream formatDomainComment (comment cat))
       (format stream formatDomainSIG (name cat) (argslist cat)))
     ((eq (marker cat) 'package)
       (format stream formatPackageAbbrev (abbreviation cat) (name cat))
       (format stream formatPackageComment (comment cat))
       (format stream formatPackageSIG (name cat) (argslist cat))))
   (let ((adults (parents cat)))
     (cond
      ((null adults) (format stream formatNoSuper))
      ((= (length adults) 1) (format stream formatSuper (car adults)))
      (t (format stream formatJoin adults))))
   (if (withlist cat)
     (progn
       (format stream formatWith)
       (dolist (sig (withlist cat))
         (format stream formatSig sig)))
     (format stream formatWithNil))
   (if (setq has (car (haslist cat)))
     (progn
       (setq condition (car (haslist-clauses has)))
       (format stream formatHas *indent* (hasclause-condition condition))
       (let ((*indent* (concatenate 'string *indent* "  ")))
         (dolist (sig (hasclause-siglist condition))
           (format stream formatSig sig)))))
   (when (addlist cat)
     (format stream formatAdd)
     (dolist (sig (addlist cat))
       (format stream "~%    ~a~%" sig)))))
```

———-

1.15.4 The WithClass class

— defclass WithClass —

```
(defclass |WithClass| () ())
```

———-

1.15.5 The AddClass class

— defclass AddClass —

```
(defclass |AddClass| () ())
```

———-

The **showSig** method will show a signature of a CategoryClass object in abbreviated forms. So, for example,

```
   (showSig |PAdicIntegerCategory|) ==> PADICCT(EUCDOM,CHARZ)
```

— defgeneric showSig —

```
(defgeneric showSig (name))
```

———-

— defmethod showSig —

```
(defmethod showSig ((name |AxiomClass|))
  (let ((parents (parents name)) abbrevs)
   (dolist (him parents) (push (abbreviation (symbol-value him)) abbrevs))
   (format t "~a(~{~a~^,~})~%"  (abbreviation name) (nreverse abbrevs))))
```

———-

1.16 Helper Functions

The **defunt** macro allows specifying the type INT in an argument list. For example, these are all valid:

```
   (defunt one (p1 p2) ....)
   (defunt one ((int p1) p2) ...)
   (defunt one ((int p1) (int p2)) ...)
```

— defmacro defunt —

```
(defmacro defunt (name (&rest args) &body body)
  "defun with optional type declarations"
  `(progn
     (declaim (ftype
               (function
                ,(let (declares)
                  (dolist (arg args)
                   (push
                    (if (listp arg)
                      (if (equalp (string (first arg)) "int")
                        'fixnum
                        (first arg))
                      t)
                     declares))
                   declares)
                 t) ,name))
     (defun ,name
       ,(loop for arg in args
          collect
            (if (listp arg)
                (second arg)
                arg))
       ,@body)))
```

———-

We usually want lines without leading and trailig spaces. The common lisp idiom for this is very wordy so we make a macro.

— defmacro trim —

```
(defmacro trim (arg)
  `(string-trim '(#\space) ,arg))
```

———-

Here we create the Axiom **getAncestors** function. It walks the tree of ancestors to collect the union of the set.

This accepts the symbol for the class or the instance.
getAncestors : **symbol** → **list(ancestors)**

— **defun getAncestors** —

```
(defun getAncestors (symbol)
  (labels (
    (theAncestors (name)
      ; look up the ancestors
      (let (class)
        (cond
         ((and (symbolp name) (setq class (find-class name nil)))
           (parents (make-instance class)))
         (t
           (parents
             (make-instance
               (class-name (class-of (symbol-value name))))))))))
  (let (name todo result)
   (setq todo (theAncestors symbol))
   (loop while todo do
     (setq name (pop todo))
     (unless (member name result) (push name result))
     (setq todo (set-difference (union (theAncestors name) todo) result)))
  (sort result #'string<))))
```

———-

The **leaves** needs to compute the most immediate ancestors (the leaves) of the type tree. We walk the tree collecting ancestors and remove duplicates.
leaves : **survive** → **survive**

— **defun leaves** —

```
(defun leaves (survive)
  (let ((l survive))
    (dolist (thing l) (setq survive (set-difference survive (getAncestors thing))))
  survive))
```

———-

We introduce the concept of a **level** to order classes based on how deep they are in the inheritance hierarchy. Level 1 Categories only depend on **CategoryClass**. Level 2 Categories only depend on Level 1, etc.

We order the class inheritance list based on level.

Here we need to be able to know what level to assign to a Category. Each level defines a variable, e.g. **level1**, **level2**, etc that contains all of the Category instance symbols defined at that level.

— **defmacro ll** —

```
(defmacro ll (name)
  `(cond
     ((member ',name level1) 'level1)
     ((member ',name level2) 'level2)
     ((member ',name level3) 'level3)
     ((member ',name level4) 'level4)
     ((member ',name level5) 'level5)
     ((member ',name level6) 'level6)
     ((member ',name level7) 'level7)
     ((member ',name level8) 'level8)
     ((member ',name level9) 'level9)
     ((member ',name level10) 'level10)
     ((member ',name level11) 'level11)
     ((member ',name level12) 'level12)
     ((member ',name level13) 'level13)
     ((member ',name level14) 'level14)
     ((member ',name level15) 'level15)
     ((member ',name level16) 'level16)
     ((member ',name level17) 'level17)
     ((member ',name level18) 'level18)
     ((member ',name level19) 'level19)
  ))
```

———-

Here we create the Axiom **getDomains** function. It finds all of the domains that inherit this class. Given a domain it return an empty set.

— defun getDomains —

```
(defun getDomains ())
```

———-

1.16.1 Signatures and OperationAlist

The **OperationAlist** is a list of all of the operations that are available for the domain. Note that if there are two operations with the same name then they are combined into one operation but with different signatures.

1.16.2 The signature class

— defclass signature —

```
(defclass signature ()
  ((name      :initarg :name      :initform ""  :accessor signature-name)
   (returns   :initarg :returns   :initform nil :accessor signature-returns)
   (arguments :initarg :arguments :initform nil :accessor signature-arguments)
   (comment   :initarg :comment   :initform nil :accessor signature-comment)
   (examples  :initarg :examples  :initform nil :accessor signature-examples)
   (infix?    :initarg :infix     :initform nil :accessor signature-infix)))
```

———-

make-signature : **(funcname return arglist comment examples infix?** → **signature**

— defun make-signature —

```
(defun make-signature (funcname return arglist
                       &optional (comment nil) (examples nil) (infix? nil))
  (make-instance 'signature :name funcname
                            :returns return
                            :arguments arglist
                            :comment comment
                            :examples examples
                            :infix infix?))
```

———-

— defmethod print-object —

```
(defvar signatureFormatNull       "~&~a~a ")
(defvar signatureFormat1Arg       "~&~a~a : ~{~^~a~^,~} -> ~a")
(defvar signatureFormatArgs       "~&~a~a : (~{~^~a~^,~}) -> ~a")
(defvar signatureFormatListArg    "List(~{~a~^,~})")
(defvar signatureFormatArgsType   "~a(~{~a~^,~})")
(defvar signatureFormatUnion      "Union(~{~s~^,~})")
(defvar signatureFormatRecord     "Record(~{~a: ~a~^,~})")
(defvar signatureFormatReturnList "List(~{~a~^,~})")
(defvar signatureFormatReturnType "~a(~{~s~^,~})")
(defvar signatureFormatComment    "~&~a  ++ ~a")
(defvar signatureFormatExamples   "~&~a  ++X ~a")

(defmethod print-object ((sig signature) stream)
  (labels (
    (singletonList? (args)
      ; change singleton list into a single element
      (loop for arg in args
        when (and (consp arg) (= (length arg) 1)) do (setq arg (car arg))
        collect arg))
    (argsList? (args)
      ; Is the args type a List?
 (format t "argsList=~s~%" args)
      (let (result)
        (dolist (arg args (nreverse result))
          (if (and (consp arg) (eq (car arg) '|List|))
            (push (format nil signatureFormatListArg
                          (substitute 'K '|#1| (cdr arg)))
             result)
          (push arg result)))))
    (argsType? (args)
      ; Is the args type a List?
 (format t "argsType=~s~%" args)
      (let (result)
        (dolist (arg args (nreverse result))
          (if (and (consp arg) (symbolp (car arg)))
            (push (format nil signatureFormatArgsType (car arg)
                          (substitute 'K '|#1| (cdr arg)))
             result)
          (push arg result)))))
    (union? (returns)
      ; Is the return type a Union?
 (format t "union=~s~%" returns)
      (if (and (consp returns) (eq (car returns) '|Union|))
         (setq returns (format nil signatureFormatUnion
                         (cdr (substitute '% '$  returns))))
         returns))
    (record? (returns)
      ; Is the return type a Record?
 (format t "record=~s~%" returns)
      (let (result)
        (if (and (consp returns) (eq (car returns) '|Record|))
          (format nil signatureFormatRecord
            (dolist (field (cdr returns) (nreverse result))
              (setq field (substitute '% '$ field))
              (push (second field) result)
              (push (third field) result)))
          returns)))
    (returnList? (returns)
      ; Is the return type a List?
 (format t "returnList=~s~%" returns)
      (if (and (consp returns) (eq (car returns) '|List|))
         (setq returns (format nil signatureFormatReturnList
                         (cdr
                           (substitute 'K '|#1|
                             (substitute '% '$  returns)))))
         returns))
    (returnType? (returns)
      ; Is the return type a Type?
 (format t "returnType=~s~%" returns)
      (if (and (consp returns) (symbolp (car returns)))
         (setq returns (format nil signatureFormatReturnType (car returns)
                         (cdr
                           (substitute 'K '|#1|
                             (substitute '% '$  returns)))))
         returns))
    (singletonVar? (returns)
      ; Is the return type a single temp variable?
 (format t "singleVar=~s~%" returns)
      (if (eq returns '|#1|)
        'K
        returns))
    (infix? (sig)
 (format t "infix ~s ~s~%" (signature-infix sig) (signature-name sig))
      (if (signature-infix sig)
        (format nil "\"~a\"" (signature-name sig))
        (signature-name sig)))
    (formatWithlist (name args returns sig)
      ; for each signature in the list, make it pretty
 (format t "formatWithList name=~s args=~s returns=~s sig=~s~%" name args returns sig)
      (cond
        ((and (null args) (null returns))
          (format stream signatureFormatNull *indent* name))
        ((= (length args) 1)
          (format stream signatureFormat1Arg *indent* name args returns))
        (t
          (format stream signatureFormatArgs *indent* name args returns)))
      (dolist (comment (signature-comment sig))
        (format stream signatureFormatComment *indent* comment))
      (when (signature-examples sig)
        (dolist (example (signature-examples sig))
          (format stream signatureFormatExamples *indent* example))))
  )
  (let ((args (signature-arguments sig)) (returns (signature-returns sig)) name)
   (format t "main=~s~%" args)
    (setq name (infix? sig))
    ; handle the arg list cleanup
    (setq args (singletonList? args))
    (setq args (argsType? args))
    (setq args (mapcar #'singletonVar? args))
    ; handle return type cleanup
    (when (and (consp returns) (= (length returns) 1)) (setq returns (car returns)))
    (setq returns (union? returns))
    (setq returns (record? returns))
    (setq returns (returnType? returns))
    (setq returns (singletonVar? returns))
    ; and format the Withlist
    (formatWithlist name args returns sig))))
```

————-

1.16.3 The amacro class

— defclass amacro —

```
(defclass amacro ()
  ((name
     :initarg :name
     :initform ""
     :accessor amacro-name
     :type symbol)
   (body
     :initarg :body
     :initform nil
     :accessor amacro-body)))
```

————-

1.16.4 The operation class

— defclass operation —

```
(defclass operation ()
  ((name
     :initarg :name
     :initform ""
     :accessor operation-name
     :type string)
   (signatures
     :initarg :signatures
     :initform nil
     :accessor operation-signatures
     :type list)))
```

———-

— defvar infixOperations —

```
(defvar *infixOperations* '("*" "**" "+" "<" "<=" "=" ">" ">=" "^"
                            "quo" "rem" "~=" "."))
```

———-

— defmethod print-object —

```
(defmethod print-object ((op operation) stream)
  (let ((name (operation-name op)) (sigs (operation-signatures op)))
    (cond
      ((eq name '|Zero|) (setq name "0"))
      ((eq name '|One|)  (setq name  "1"))
      ((eq name '|elt|)  (setq name  ".")))
    (dolist (sig sigs)
      (if (member name *infixOperations* :test #'string=)
        (format stream "?~a? : ~a~%" name sig)
        (format stream "~a : ~a~%" name sig)))))
```

———-

— defgeneric operation-add —

```
(defgeneric operation-add (operation signature))
```

———-

— defmethod operation-add —

```
(defmethod operation-add ((op operation) signature)
  (setf (operation-signatures op) (cons signature (operation-signatures op))))
```

———-

make-operation : **name** → **operation**
— defun make-operation —

```
(defun make-operation (name)
  (make-instance 'operation :name name))
```

———-

From OperationAlist to Signatures

```
(* (($ (|PositiveInteger|) $) NIL) (($ (|NonNegativeInteger|) $) NIL) (($ $ $) NIL))
                                    -> ?*? : (%,%) -> %
                                    -> ?*? : (NonNegativeInteger,%) -> %
                                    -> ?*? : (PositiveInteger,%) -> %
(** (($ $ (|NonNegativeInteger|)) NIL) (($ $ (|PositiveInteger|)) NIL))
                                    -> ?**? : (%,PositiveInteger) -> %
                                    -> ?**? : (%,NonNegativeInteger) -> %
(+ (($ $ $) NIL))                   -> ?+? : (%,%) -> %
(< (((|Boolean|) $ $) NIL))         -> ?<? : (%,%) -> Boolean
(<= (((|Boolean|) $ $) NIL))        -> ?<=? : (%,%) -> Boolean
(= (((|Boolean|) $ $) NIL))         -> ?=? : (%,%) -> Boolean
(> (((|Boolean|) $ $) NIL))         -> ?>? : (%,%) -> Boolean
(>= (((|Boolean|) $ $) NIL))        -> ?>=? : (%,%) -> Boolean
(|One| (($) NIL T CONST))           -> 1 : () -> %
(|Zero| (($) NIL T CONST))          -> 0 : () -> %
(^ (($ $ (|NonNegativeInteger|)) NIL) (($ $ (|PositiveInteger|)) NIL))
                                    -> ?^? : (%,PositiveInteger) -> %
                                    -> ?^? : (%,NonNegativeInteger) -> %
(|coerce| (((|OutputForm|) $) NIL))  -> coerce : % -> OutputForm
(|gcd| (($ $ $) 11))                -> gcd : (%,%) -> %
(|hash| (((|SingleInteger|) $) NIL)) -> hash : % -> SingleInteger
(|latex| (((|String|) $) NIL))      -> latex : % -> String
(|max| (($ $ $) NIL))               -> max : (%,%) -> %
(|min| (($ $ $) NIL))               -> min : (%,%) -> %
(|one?| (((|Boolean|) $) NIL))      -> one? : % -> Boolean
(|qcoerce| (($ (|Integer|)) 8))     -> qcoerce : Integer -> %
(|quo| (($ $ $) NIL))               -> ?quo? : (%,%) -> %
(|random| (($ $) NIL))              -> random : % -> %
(|recip| (((|Union| $ "failed") $) NIL))
                                    -> recip : % -> Union(%,"failed")
(|rem| (($ $ $) NIL))               -> ?rem? : (%,%) -> %
(|sample| (($) NIL T CONST))        -> sample : () -> %
(|shift| (($ $ (|Integer|)) 7))     -> shift : (%,Integer) -> %
(|sup| (($ $ $) 6))                 -> sup : (%,%) -> %
(|zero?| (((|Boolean|) $) NIL))     -> zero? : % -> Boolean
(~= (((|Boolean|) $ $) NIL))        -> ?~=? : (%,%) -> Boolean
(|divide| (((|Record| (|:| |quotient| $) (|:| |remainder| $)) $ $) NIL))
                                    -> divide : (%,%) -> Record(quotient: %,remainder: %)
(|exquo| (((|Union| $ "failed") $ $) NIL))
                                    -> exquo : (%,%) -> Union(%,"failed")
(|subtractIfCan| (((|Union| $ "failed") $ $) 10))
                                    -> subtractIfCan : (%,%) -> Union(%,"failed")
```

operationToSignature : **alistEntry** → **signature**

— defun operationToSignature —

```
(defun operationToSignature (alistEntry)
  (let (returnType args sig result)
    (setq result (make-operation (car alistEntry)))
    (dolist (item (cdr alistEntry))
      (setq sig (substitute '% '$ (car item)))
      (setq returnType (car sig))
      (setq args (cdr sig))
      (operation-add result (make-signature returnType args nil)))
    result))
```

———-

As painful as it is, this code strives to reproduce the exact output of the **)show** command in Axiom.

Functions with the same name but multiple signatures, such as *****, are stored in a single operation instance. We have to break up the multiple signatures into single signatures. We do this with the **split** function.

Some signatures are in tabular format. Longer signatures are output after the table. There are special cases and we keep a list of these in the ***forceLongs* variable**. Sigh.

The tabular format has two columns, the second column is at character position 40. The SBCL tabular output in format is broken so we had to create the **padTo38** function to add pad characters to signatures.

— defvar forcelongs —

```
(defvar *forceLongs* '("exquo"))
```

———-

operations-show : **operationAlist** → **values**
— defun operations-show —

```
(defun operations-show (operationAlist)
  (labels (
    ; split multiple signatures
    (split (string)
       (splitchar string #\newline))
    ; Axiom wants certain lines delayed for no reason
    (forceLong (sig)
      (let (opname)
        (setq opname (subseq sig 0 (1- (position #\: sig))))
        (member opname *forceLongs* :test #'string=)))
    ; Axiom wants a tabular display
    (padTo38 (string)
      (let ((excess (- 38 (length string)))
            (pad "                                      "))
        (concatenate 'string string (subseq pad 0 excess)))))
  (let (sigs longlines shortlines (column 0))
   (dolist (op operationAlist)
     ; operationToSignature might return multiple signatures
     (setq sigs (format nil "~a" (operationToSignature op)))
     (setq sigs (split (subseq sigs 0 (1- (length sigs)))))
     ; classify the output by length
     (dolist (sig (nreverse sigs))
       (if (or (>= (length sig) 36) (forceLong sig))
         (push sig longlines)
         (push sig shortlines))))
    ; output the short lines
    (dolist (sig shortlines)
      (if (= column 0)
        (progn (format t " ~a" (padTo38 sig)) (setq column 1))
        (progn (format t "~a~%" sig) (setq column 0))))
   ; output the long lines
   (dolist (sig longlines)
     (format t "~& ~a~%" sig))
   (values))))
```

———-

From Signatures to OperationAlist

— defun signatureToOperation —

```
(defun signatureToOperation (sig)
  (declare (ignore sig))
)
```

———-

The Parser

The parser operates on the input stream step by step.

There are three possible approaches to the parser and these are left "unchosen" for the moment. There is the traditional Chomsky BNF style, the Parsing Expression Grammar (PEG) [Ford04], and the Pratt parser [Prat73].

PEG parsers reduce the ambiguity of choice, making parsing more efficient.

Pratt Parsers allow user modification of syntax. While this has not been of interest to Axiom it has been used in the proof community. Once proof technology is integrated and presented to users it will probably be more important.

1.17 ParserClass

The ***place*** is the search path for the .spad files. These paths are concatenated before the spad filenames.

— defvar place —

```
(defvar *place* "/research/20191011/src/algebra/")
; "axiom/src/algebra/")
; "/research/fricas/waldek/fricasNew/src/algebra/"
```

———-

— defclass ParserClass —

```
(defclass ParserClass ()
  ((abbrev     :initarg :abbrev     :initform nil :accessor parser-abbrev)
   (topcomment :initarg :topcomment :initform nil :accessor parser-topcomment)
   (macros     :initarg :macros     :initform nil :accessor parser-macros)
   (cdpsig     :initarg :cdpsig     :initform nil :accessor parser-cdpsig)
   (with       :initarg :with       :initform nil :accessor parser-with)
   (add        :initarg :add        :initform nil :accessor parser-add)))
```

———-

— defmethod print-object —

```
(defvar formatabbrev "~a")
```

```
(defvar formattopcomment "~a~%")
(defvar formatmacros "~a ==> ~a~%")
(defvar formatcdpsig "~a~%")

(defmethod print-object ((p ParserClass) stream)
  (format stream formatabbrev (parser-abbrev p))
  (format stream formattopcomment (parser-topcomment p))
  (format stream formatcdpsig (parser-cdpsig p)))
```

———-

The **theparse** is a global variable holding the result of the parse. It holds a **ParserClass** object.

— defvar theparse —

```
(defvar theparse nil)
```

———-

The **make-Sourcecode** [p47] reads the file and constructs a **sourcecode** [p49] object.

The bf make-Sourcecode function constructs a **sourcecode-pile** field containing a "useful" version of the original source, (e.g. the source stripped of −− comments and joining contiued lines) so we can easily parse the result.

The **Parse 1** error occurs when the **)abbrev** line is missing. The **)abbrev** line must occur before the code.

Parse : **filename** → **(theparse,pile)**
— defun Parse —

```
; begin...for debugging
(defvar t1 nil "a copy of the source code for debugging")

(defun expandMacro (name)
  (cdr (assoc name (parser-macros theparse))))

(defun P (filename)
 (setq t1 (make-sourcecode (concatenate 'string *place* filename ".spad")))
 (pretty (sourcecode-tree t1))
 (values))

(defun q (filename)
 (setq t1 (make-sourcecode
            (concatenate 'string filename ".spad")))
 (pretty (sourcecode-tree t1))
 (values))

; end...for debugging

(defvar formatParse1 "Parse 1: ~%    In ~a~%    Missing )abbrev line~%")

(defun Parse (filename)
  (unwind-protect
    (let (pile object alist iscat?)
      (setq t1 (make-Sourcecode (concatenate 'string *place* filename ".spad")))
      (setq pile (sourcecode-pile t1))
      (setq theparse (make-instance 'ParserClass))
      (multiple-value-setq (pile object) (FSM-abbrev pile))
      (unless object
        (error formatParse1 (concatenate 'string *place* filename ".spad")))
      (setf (parser-abbrev theparse) object)
      (multiple-value-setq (pile object) (FSM-comment pile))
      (setf (parser-topcomment theparse) object)
      (multiple-value-setq (pile alist) (FSM-macros pile))
      (setf (parser-macros theparse) alist)
      (multiple-value-setq (pile object) (FSM-cdpsig pile theparse))
      (setf (parser-cdpsig theparse) object)
      (multiple-value-setq (pile iscat?) (FSM-isCategory? pile theparse))
      (if iscat?
        (multiple-value-setq (pile object) (FSM-catBody pile theparse))
        (multiple-value-setq (pile object) (FSM-dpBody pile theparse)))
      (multiple-value-setq (pile object) (FSM-variables pile theparse))
  (print (list 'pile0 (first pile)))
      (values theparse pile))
  (print (list 'gotit))))
```

——————-

1.18 make-Sourcecode

The bf make-Sourcecode function constructs a **sourcecode** instance containing the original source, the source stripped of −− comments, joining contiued lines, and finally converted from pile form to tree form.

The **make-Sourcecode** function reads a spad file in as a list of strings. All minus comments are deleted, using **noComments**, from the initial −− to the end of the line.

The result is a **sourcecode** instance contains the raw contents of the file, the cleaned up version called a pile, and a tree version, containing parens indicating depth.

Then, using **oneline**, any line that has a trailing escape character (the underscore) gets merged with the following line.

The **escaped?** predicate checks for the trailing underscore. The **oneline** process, when it finds a trailing underscore, will join the lines(s) until a non-escaped line occurs.

The **blankline?** predicate checks for a blank line. These are elimiated as part of the **oneline** process.

A bit of clever optimization is possible if the **oneline** function processed the list of strings in reverse so the two calls to nreverse could be removed.

make-Sourcecode : **filename** → **sourcecode**
— defun make-Sourcecode —

```
(defvar formatMakeSourcecode "make-Sourcecode: ERROR: File ~a does not exist")

; FILENAME -> SOURCECODE
(defun make-Sourcecode (filename)
  (labels (
    ; LINE -> BOOLEAN
    (blankline? (line)
      (string= "" (trim line)))
    ; LINE -> LINE
    (boot? (line)
      (when (>= (length line) 3)
        (string-equal ")bo" (subseq line 0 3))))
    ; LINE -> LINE
    (noComments (line)
      (subseq line 0 (search "--" line)))
    ; LINE -> BOOLEAN
    (escaped? (line)
      (let ((len (length line)))
        (and (> len 0) (char= #\_ (char line (1- len))))))
    ; LINELIST -> LINELIST
    (oneline (linelist)
      (let (gather result)
        (dolist (line linelist)
          (let ((underscored? (escaped? line)))
            (cond
              ; skip blank lines
              ((blankline? line))
              ; first time
              ((and (not gather) underscored?)
                (setq gather (string-right-trim '(#\_) line)))
              ; still more
              ((and gather underscored?)
                (setq gather
                  (concatenate 'string gather " " (string-trim '(#\space #\_) line))))
              ; last one
              ((and gather (not underscored?))
                (push
                  (concatenate 'string gather " " (string-trim '(#\space #\_) line))
                 result)
                (setq gather nil))
              (t
                (push line result)))))
      result))  )
  (let (rawcode listLine pile)
    (with-open-file (file filename :direction :input :if-does-not-exist nil)
      (if file
        (let ((done (gensym)))
          (do ((line (read-line file nil done) (read-line file nil done)))
              ((eq line done) (setq pile (nreverse (oneline (nreverse listLine)))))
              (push line rawcode)
              (unless (boot? line)
                (push (noComments line) listLine)))
        (make-instance 'sourcecode :name filename
                                   :rawcode (nreverse rawcode)
                                   :pile pile
                                   :tree (pile2tree pile)))
     (error formatMakeSourcecode filename)))))
```

——————-

1.19 sourcecode-pile

The **sourcecode** type contains 4 fields:

1. name – the filename of the source file
2. rawcode – the exact source text in the file
3. pile – remove −− comments, continued lines
4. tree – piled strings to a nested tree

1.19.1 The sourcecode class

— defclass sourcecode —

```
(defclass sourcecode ()
  ((name    :initarg :name    :initform nil :accessor sourcecode-name)
   (rawcode :initarg :rawcode :initform nil :accessor sourcecode-rawcode)
   (pile    :initarg :pile    :initform nil :accessor sourcecode-pile)
   (tree    :initarg :tree    :initform nil :accessor sourcecode-tree)))
```

———-

— defmethod print-object —

```
;(defmethod print-object ((source sourcecode) stream)
;  (pretty (sourcecode-tree source)))
```

———-

Spad is indentation sensitive so the rawcode semantics depends on the indentation.

1.20 FSM-abbrev

1.20.1 Recognize abbreviation command line

An **)abbrev** line consists of 4 tokens, for example,

```
   )abbrev domain BITS Bits
```

where the domain is one of "category", "domain", or "package", which can be of any case. The BITS field must be less than 8 characters and all uppercase, and the Bits field must be a valid constructor name.

1.20.2 The abbreviation class

The **abbreviation** holds the result of parsing an)abbrev line.

The **FSM-abbrev** function parses an)abbrev line.

— defclass abbreviation —

```
(defclass abbreviation ()
  ((class  :initarg :class  :initform nil :accessor abbreviation-class)
   (abbrev :initarg :abbrev :initform nil :accessor abbreviation-abbrev)
   (string :initarg :string :initform ""  :accessor abbreviation-string)
   (name   :initarg :name   :initform nil :accessor abbreviation-name)))
```

———-

make-instance : **(class,abbrev,string)** → **abbreviation**
— defun make-abbreviation —

```
(defun make-abbreviation (class abbrev string)
  (make-instance 'abbreviation :class class :abbrev abbrev :string string
                               :name (intern string)))
```

———-

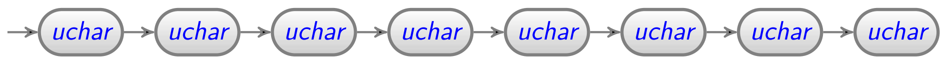
— defmethod print-object —

```
(defvar formatAbbreviation ")abbrev ~a ~a ~a~%")

(defmethod print-object ((abb abbreviation) stream)
  (format stream formatAbbreviation
    (abbreviation-class abb)
    (abbreviation-abbrev abb)
    (abbreviation-string abb)))
```

———-

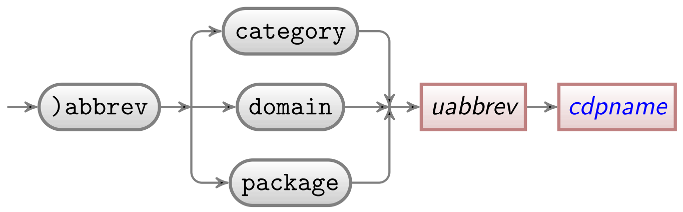
The **FSM-abbrev** function handles the abbrev line which contains 4 symbols.

An UABBREV non-terminal consists of 1 to 8 CAP characters where a CAP character is either an uppercase character or a number.



```
UABBREV ::= UCHAR
          | UCHAR UCHAR
          | UCHAR UCHAR UCHAR
          | UCHAR UCHAR UCHAR UCHAR
          | UCHAR UCHAR UCHAR UCHAR UCHAR
          | UCHAR UCHAR UCHAR UCHAR UCHAR UCHAR
          | UCHAR UCHAR UCHAR UCHAR UCHAR UCHAR UCHAR
          | UCHAR UCHAR UCHAR UCHAR UCHAR UCHAR UCHAR UCHAR
```

Figure 1.1: UABBREV syntax



```
ABBREV ::= ')abbrev' ['category' | 'domain' | 'package'] UABBREV CONSTRUCT
```

Figure 1.2: ABBREV syntax

FSM-abbrev : **linelist** → **(linelist,abbreviation)**
— defun FSM-abbrev —

```
(defun FSM-abbrev (linelist)
  (let ((split (splitchar (first linelist) #\space)) result)
    (when
      (fsm-and
        (fsm-match ")abbrev")
        (fsm-or
          (fsm-match "category")
          (fsm-match "domain")
          (fsm-match "package"))
        (fsm-abbname)
        (fsm-word))
      (values
        (rest linelist)
        (make-abbreviation (third result) (second result) (first result))))))
```

———-

FSM-symbol : **linelist** → **(linelist,symboliation)**
— defun FSM-symbol —

```
(defun FSM-symbol (linelist)
  (let (result)
    (with-input-from-string (s (first linelist))
      (setf (readtable-case *readtable*) :preserve)
      (setq result (read s))
      (setf (readtable-case *readtable*) :upcase))
    (when result
      (setf (first linelist)
        (subseq (trim (first linelist)) (length (string result)))))
    (values linelist (symbolp result) result)))
```

———-

1.21 FSM-comment

1.21.1 Parse Comments

The **FSM-comment** function strips off the comments from linelist, gathering them together. It returns 2 values, the first is a new linelist without the comments. The second is a gather object containing the comment lines.

FSM-comment : **linelist** $\rightarrow$ **linelist,list(comments)**
— defun FSM-comment —

```
(defun FSM-comment (linelist)
  (let (comment result)
    (setq comment (make-instance 'gather))
    (values
      (FSM-gather linelist (FSM-startsWith "++") comment)
      comment)))
```

———-

1.22 FSM-macros

The **FSM-macros** function scans the source code for macros. If one is found the function creates a pair (name . definition) and puts it on an association list. The result is the program text minus the macros and the association list of macros. So, for example, a source code line that reads

```
  SIG ==> BitAggregate() with
```

will become the pair

```
  (SIG . "  BitAggregate() with")
```

A minor wrinkle is that the macro (e.g. SIG) has a meaningful indentation. We need to compute the indentation, clip out that many spaces, and later add it into the first returned line. The actual pair created looks like

```
  (SIG . "  BitAggregate() with")
```

FSM-macros : **linelist** $\rightarrow$ **(linelist,macroAlist)**
— defun FSM-macros —

```
(defun FSM-macros (linelist)
  (labels (
    ; is this a macro line?
    (hasMacros? (linelist)
     (loop for line in linelist
       when (search "==>" line) return t))
    (join (stringlist)
      (format nil "~{~a~^ ~}" stringlist))
    (makeAlist (gather)
      (let (alllines line name def indent spaces)
       ; get all the lines in the macro
       (setq alllines (gather-lines gather))
       ; remember the indent
       (setq indent (indent (first alllines)))
       (setq spaces (subseq (first alllines) 0 indent))
       ; hack out the macro name
       (setq line (splitchar (trim (pop alllines)) #\space))
       (setq name (intern (first line)))
       (setq def (cons (join (cddr line)) alllines))
       ; insert indent spaces in first macro line
       (setf (first def) (concatenate 'string spaces (first def)))
       (cons name def)))
  )
  (let (macro macroAlist)
    (loop while (hasMacros? linelist) do
      (multiple-value-setq (linelist macro) (FSM-extract1Macro linelist))
      (push (makeAlist macro) macroAlist))
    (values linelist macroAlist))))
```

———-

Sometimes we just want to know what symbols are actually the name of a macro. The **FSM-macroNames** function returns a list of those names.

FSM-macroNames : **theparse** $\rightarrow$ **list(macronames)**

— defun FSM-macroNames —

```
(defun FSM-macroNames (theparse)
  (mapcar #'(lambda (x) (car x)) (parser-macros theparse)))
```

———-

1.23 FSM-cdpsig

1.23.1 The CDPSigClass class

— defclass CDPSigClass —

```
(defclass CDPSigClass ()
  ((cdpname  :initarg :classname :initform nil :accessor CDPSigClass-cdpname)
   (argslist :initarg :argslist  :initform nil :accessor CDPSigClass-argslist)
   (cdptag   :initarg :cdptag    :initform nil :accessor CDPSigClass-cdptag)))
```

———-

— defmethod print-object —

```
(defvar formatcat "~a~a : Category == ~%")
(defvar formatdp "~a~a : ~%")

(defmethod print-object ((cdp CDPSigClass) stream)
  (if (string-equal "category" (CDPSigClass-cdptag cdp))
    (format stream formatcat (CDPSigClass-cdpname cdp) (CDPSigClass-argslist cdp))
    (format stream formatdp  (CDPSigClass-cdpname cdp) (CDPSigClass-argslist cdp))))
```

———-

The **FSM-cdpsig** function expects to be called to handle the definition of a category, domain, or package. It constructs a **CDPSigClass** object to hold the name and argument list. At this point, the argument list is essentially unparsed, consisting of the top level string starting at the open parenthesis and ending at the matching closed parenthesis. Further processing will parse the contents of this list. This is necessary since the variables could be macros in the body which we have not yet seen.

FSM-cdpsig : **(linelist,theparse)** $\rightarrow$ **(linelist,CDPSigClass)**
— defun FSM-cdpsig —

```
(defun FSM-cdpsig (linelist theparse)
  (let ((split (splitchar (first linelist) #\space)) result cdpname argslist)
    (when (FSM-startsWith (first split))
      (setq result (make-instance 'CDPSigClass))
      (setf (CDPSigClass-cdptag result) (abbreviation-class (parser-abbrev theparse)))
      (setq cdpname (abbreviation-string (parser-abbrev theparse)))
      (setf (CDPSigClass-cdpname result) cdpname)
      (setq linelist (cons (subseq (pop linelist) (length cdpname)) linelist))
      (multiple-value-setq (linelist argslist) (FSM-gatherList linelist))
      (setf (CDPSigClass-argslist result) argslist)
      (values linelist result))))
```

———-

1.24 FSM-isCategory?

FSM-isCategory? : **(linelist,theparse)** $\rightarrow$ **(linelist,boolean)**
— defun FSM-isCategory? —

```
(defun FSM-isCategory? (linelist theparse)
  (let ((split (splitchar (first linelist) #\space)) result tmp)
     (FSM-match "")
     (FSM-match ":")
     (setq tmp (abbreviation-class (parser-abbrev theparse)))
     (if split
       (setq result (list (joinsplit split #\space) (rest linelist)))
       (setq result (rest linelist)))
     (if (and (string-equal tmp "category") (FSM-match tmp))
       (values result t)
       (values result nil))))
```

———-

1.25 FSM-catBody

FSM-catBody : **(linelist,theparse) → list(tokens)**
— defun FSM-catBody —

```
(defun FSM-catBody (linelist theparse)
  (let ((split (splitchar (first linelist) #\space)) result)
    (setq result (FSM-match "=="))
    (values (cons (joinsplit split #\ ) (rest linelist)) result)))
```

———-

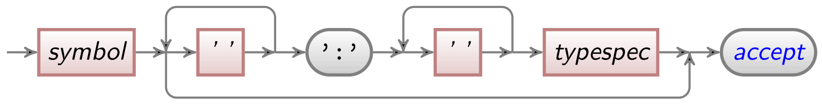
1.26 FSM-dpBody

FSM-dpBody : **(linelist,theparse) → nil**
— defun FSM-dpBody —

```
(defun FSM-dpBody (linelist theparse)
  (let ((split (splitchar (first linelist) #\space)) result)
    (values (cons (joinsplit split #\ ) (rest linelist)) result)))
```

———-

1.27 FSM-variables



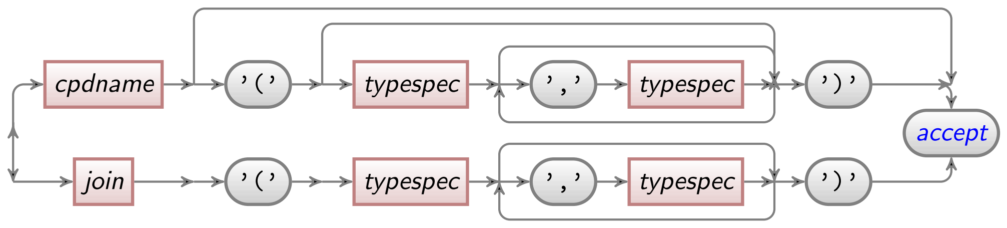
```
VAR ::= SYMBOL [ [WSPACE]* ':' [WSPACE]* TYPESPEC ]
```

Figure 1.3: VAR syntax

FSM-VAR : **linelist → (linelist,var)**
— defun FSM-VAR —

```
(defun FSM-VAR (linelist)
  (let ((split (splitchar (first linelist) #\space)) result)
    (when
      (FSM-and
        (setq varname (FSM-symbol))
        (FSM-optional
          (FSM-skipwspace)
          (FSM-colon)
          (FSM-skipwspace)
          (setq spec (FSM-typespec))))
      (values
        linelist
        (make-FSM-VAR varname spec))))
```

————-



```
TYPESPEC ::= CDPNAME [ '(' [ TYPESPEC [ ',' TYPESPEC ]* ] ')']
           | JOIN '(' TYPESPEC [ ',' TYPESPEC ] ')'
```

Figure 1.4: TYPESPEC syntax

1.27.1 The FSM-VAR class

— defclass FSM-VAR —

```
(defclass FSM-VAR ()
  ((variable   :initarg :variable  :accessor FSM-VAR-variable)
   (paramtype  :initarg :paramtype :accessor FSM-VAR-paramtype)))
```

————-

— defmethod print-object —

```
(defmethod print-object ((param FSM-VAR) stream)
  (format stream "~a:~a"
    (FSM-VAR-variable param)
    (FSM-VAR-paramtype param)))
```

————-

The **FSM-variables** function is called to "clean up" the variables of a type definition. These are found in two places. Either they are in the argument list where the variable name is followed by a colon or they are in the body where the variable name is followed by a colon.

FSM-variables : **(linelist,theparse)** $\rightarrow$ **nil**
— defun FSM-variables —

```
(defun FSM-variables (linelist theparse)
  (print (list 'fsm-variables
    (fsm-gathervars (cdpsigclass-argslist (parser-cdpsig theparse)))))
  (values linelist nil))
```

————-

1.28 Parser support functions

These two functions, **splitchar** and **joinsplit** are essentiall inverses of each other. So

```
   (joinsplit (splitchar somestring #\space) #\space)
```

should return the original string, without leading or trailing spaces.

The **splitchar** function will take a **string** and break the string apart every place it sees the **char**.

splitchar : **(string,char)** $\rightarrow$ **list(token)**
— defun splitchar —

```
(defun splitchar (string char)
    (loop for i = 0 then (1+ j)
      as j = (position char string :start i :test #'char=)
      collect (subseq string i j)
      while j))
```

———-

The **joinsplit** function will that a list of strings, called the **split** and concatenate them, inserting the **char** character between the strings.

Note that this can be used to construct a comma-separated list by

```
    (joinsplit somestring #\,)
```

joinsplit : **(list(token),char)** $\rightarrow$ **string**
— defun joinsplit —

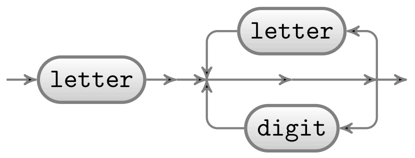
```
(defun joinsplit (split char)
  (let ((charstr (string char)) result)
    (setq result
      (apply #'concatenate 'string
        (loop for word in split
          collect word collect charstr)))
   (subseq result 0 (1- (length result)))))
```

———-

The **explode** turns a string into characters.
explode : **string** $\rightarrow$ **list char**
— defun explode —

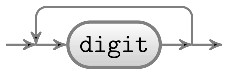
```
(defun explode (string)
  (loop for c across string collect c))
```

———-



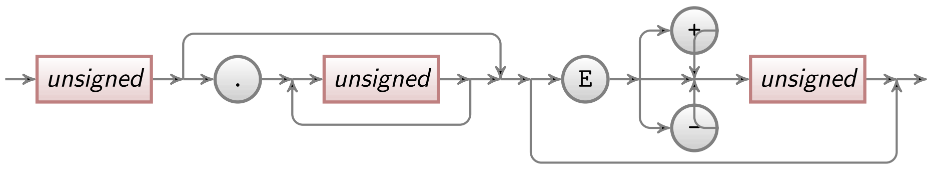
```
SYMBOL := letter [ letter | digit ]*
```

Figure 1.5: SYMBOL syntax



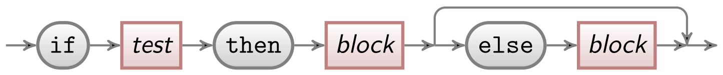
```
UNSIGNED := digit [ digit ]*
```

Figure 1.6: UNSIGNED syntax



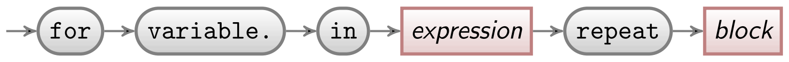
```
FLOAT ::=
```

Figure 1.7: FLOAT syntax



```
IF ::= if TEST then BLOCK [else BLOCK]
```

Figure 1.8: IF syntax



```
FOR ::= for VARIABLE in EXPRESSION repeat BLOCK
```

Figure 1.9: FOR syntax

1.29 The Compiler Function

Given a **theparse** structure, construct the corresponding defclass.

```
(defclass |BitsType| (|BitAggregateType|)
  ((parents :initform '(|BitAggregate|))
   (name :initform "Bits")
   (marker :initform 'domain)
   (abbreviation :initform 'BITS)
   (comment :initform (list
     "Bits provides logical functions for Indexed Bits."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Bits|
  (progn
    (push '|Bits| *Domains*)
    (make-instance '|BitsType|)))
```

The **compileSpad** function takes the parser structure and compiles it to executable code.
compileSpad : **theparse** → **values**

— defun compileSpad —

```
(defun compileSpad (theparse)
  (let* ((abbrev (parser-abbrev theparse))
         (classname (intern (concatenate 'string (abbreviation-string abbrev) "Type"))))
    (print
     `(progn
        (defclass ,classname ()
          ((name         :initform ,(abbreviation-string abbrev))
           (marker       :initform ,(abbreviation-class abbrev))
           (abbreviation :initform ,(abbreviation-abbrev abbrev))
           (comment      :initform ,(parser-topcomment theparse))))
        (defvar ,(abbreviation-name abbrev)
          (progn
             (push ',(abbreviation-name abbrev) *Domains*)
             (make-instance ',classname)))))
   (values)))
```

———–

1.30 The Language

```
FILENAME    ::= PATHNAME
RAWCODE     ::= List(STRING)
PILE        ::= List(STRING)
TREE        ::= Tree(STRING)
SOURCECODE  ::= FILENAME RAWCODE PILE TREE

UCHAR       ::= uppercase character

CONSTRUCT   ::= constructorname


FILE        ::= ABBREV BODY
BODY        ::= CDPSIG MACROS WITH ADD
CDPSIG      ::= CDPNAME ['(' [ CDPARGS ] ')'] '=='
```

```
MACROS      ::= MACNAME '==>' MACBODY
WITH        ::= [PARENTS] with SIGNATURES
ADD         ::=

COLONTYPE   ::= name : TYPESPEC [',' COLONGTYPE]*
RECORD      ::= 'Record(' COLONTYPE* ')'
UNIONTAG    ::= TYPESPEC | STRING
UNION       ::= 'Union(' UNTIONTAG [',' UNIONTAG]+ ')'

INFIX       ::= + | - * | ** | mod | ^ | / | exquo
PREFIX      ::= + | -
SPECIALNAME ::= "*" | "**" | "^" | "/" | PREFIX | INFIX
SIGNAME     ::= SPECIALNAME | SYMBOL
FSMSIG      ::= INDENT SIGNAME [':' SIGINTYPE '->' SIGOUTTYPE] [SIGIFSEC]
```

1.31 Finite State Machine tools

The Finite State Machine tools mimic the EBNF in the language grammar. An example of its use is to recognize the abbreviation line. The line looks like:

```
   )abbrev domain TIM Tim
```

The Finite State Machine code to recognize this and construct an abbreviation instance would be:

```
(defun FSM-abbrev (linelist)
  (let ((split (splitchar (first linelist) #\space)) result)
    (when
      (fsm-and
        (fsm-match ")abbrev")
        (fsm-or
          (fsm-match "category")
          (fsm-match "domain")
          (fsm-match "package"))
        (fsm-abbname)
        (fsm-word))
      (values
        (rest linelist)
        (make-abbreviation (third result) (second result) (first result))))))
```

Things to note about this code are that the variables **split** and **result** are expected to be in the environment of the macros. When this code successfully recognizes the input line it constructs a class object to hold the result.

The **FSM-OR** macro mimics the action of the choice vertical bar in the grammar. It accepts a list of tests and returns if one of the tests succeeds.

— defmacro FSM-OR —

```
(defmacro FSM-OR (&body FSM-tests)
  `(or ,@FSM-tests))
```

———-

The **FSM-AND** macro mimics the action of a sequence of matches in the grammar. It accepts a list of tests and returns if all of the tests succeed.

— defmacro FSM-AND —

```
(defmacro FSM-AND (&body FSM-tests)
  `(and ,@FSM-tests))
```

———-

1.31.1 The gather class

The **gather** class holds the result of collecting several associated lines. It is filled by side-effect by **FSM-gather**.

— defclass gather —

```
(defclass gather ()
  ((lines  :initarg :lines :initform nil :accessor gather-lines)))
```

———-

— defmethod print-object —

```
(defvar formatgather "~{~a~%~^~}")

(defmethod print-object ((g gather) stream)
  (format stream formatgather (gather-lines g)))
```

———-

The **FSM-gather** macro rips lines that pass the test from the list of input strings.It expects 3 arguments

1. **linelist** is a list of input strings from the raw field of the sourcecode class object.
2. **test** is a test to decide whether this line should be gathered from the linelist.
3. **into** is an instance of the **gather** class that will hold the lines that pass the test.

Given a linelist that looks like:

```
  ("++ Author: Stephen M. Watt"
    "++ Description:"
    "++ \\spadtype{Bits} provides logical functions for Indexed Bits."
    "Bits() : SIG == CODE where"
    "  SIG ==> BitAggregate() with"
    "    bits : (NonNegativeInteger, Boolean) -> %"
    "      ++ bits(n,b) creates bits with n values of b"
    "  CODE ==> IndexedBits(1) add"
    "    bits(n,b)    == new(n,b)")
```

and the call to gather comments, lines starting the "++":

```
(defun FSM-comment (linelist)
  (let (comment result)
    (setq comment (make-instance 'gather))
    (values
      (FSM-gather linelist (FSM-match "++") comment)
      comment)))
```

when invoked with:

```
   (multiple-value-setq (t3 t4) (fsm-comment t2))
```

we end up with t4 as a **gather** instance containing the lines that passed the **FSM-match** test. For example,

```
The object is a STANDARD-OBJECT of type GATHER.
0. LINES:
 ("++ Author: Stephen M. Watt"
  "++ Description:"
  "++ \\spadtype{Bits} provides logical functions for Indexed Bits.")
```

The t3 variable contains the new linelist without the matching lines:

```
   ("Bits() : SIG == CODE where"
    "  SIG ==> BitAggregate() with"
    "    bits : (NonNegativeInteger, Boolean) -> %"
    "      ++ bits(n,b) creates bits with n values of b"
    "  CODE ==> IndexedBits(1) add"
    "    bits(n,b)    == new(n,b)")
```

— defmacro FSM-gather —

```
(defmacro FSM-gather (linelist test into)
  `(let ((split (splitchar (first ,linelist) #\space)))
     (loop while (and linelist ,test) do
       (setf (gather-lines ,into)
         (append (gather-lines ,into) (list (pop ,linelist))))
       (setq split (splitchar (first ,linelist) #\space)))
     ,linelist))
```

———-

NOTE: **These macros assume two variables in the environment.**

1. **split** that is a tokenized list of the input, usually by calling the **splitchar** function. Each element of the list is a string.
2. **result** is a variable used to collect matches in a list.

The **FSM-startsWith** occurs in cases where we want to check whether the prefix is a word we expect, such as looking for the category name, for example,

```
   Bits() : Category
   FSM-match "Bits"
```

— defmacro FSM-startsWith —

```
(defmacro FSM-startsWith (word)
  `(let ((term (explode (first split))))
     (setq result t)
     (loop for c across ,word do
       (when (and result (char/= c (pop term)))
         (setq result nil)))
     (when result ,word)))
```

———-

The **FSM-match** matches a string with any case and, when successful, it adds the matched word to the result and pops it off the split.

— defmacro FSM-match —

```
(defmacro FSM-match (word)
  `(when (string-equal (first split) ,word)
     (push (first split) result)
     (pop split)))
```

———-

The **FSM-dword** matches the lowercased word, then match. When successful, it adds the matched word to the result and pops it off the split.

— defmacro FSM-dword —

```
(defmacro FSM-dword (word)
  `(when (string= (string-downcase ,word) (first split))
     (push (first split) result)
     (pop split)))
```

———-

The **FSM-uword** matches the uppercased word, then match. When successful, it adds the matched word to the result and pops it off the split.

— defmacro FSM-uword —

```
(defmacro FSM-uword (word)
  `(when (string= (string-upcase ,word) (first split))
     (push (first split) result)
     (pop split)))
```

———-

The **FSM-word** simply return the next word. It always succeeds. It adds the word to the result and pops it off the split.

— defmacro FSM-word —

```
(defmacro FSM-word ()
  '(progn
     (push (first split) result)
     (pop split)))
```

———-

The **FSM-cdpname** checks to see if the next word in the split is a known class. For example,

```
  (FSM-cdpname *categories*)
```

will check if the string at the top of the split is in the known list of category names. (Note that it interns the string first). It adds the word to the result and pops it off the split.

— defmacro FSM-cdpname —

```
(defmacro FSM-cdpname (cdplist)
  `(when (member (intern (first split)) ,cdplist)
     (push (first split) result)
     (pop split)))
```

———-

Abbreviations of categories, domains, and packages must be 8 or less characters and all uppercase.

The **FSM-abbname** performs this check. When successful, it adds the word to the result and pops it off the split.

— defmacro FSM-abbname —

```
(defmacro FSM-abbname ()
  `(let ((word (first split)))
     (when (and (<= (length word) 8)
                (loop with result
                      for c across word
                      while (setq result (or (digit-char-p c) (upper-case-p c)))
                      finally (return result)))
       (push (first split) result)
       (pop split))))
```

———-

The function is called when an open-paren is detected as the next character in the first line of the linelist. It will walk across the linelist gathering all of the text between the open-paren and the balancing closed-paren, even if there are newlines or other parens. It works character by character. The **result** variable is the parenthesized object, including the delimiting parens.

The **depth** variable is used to handle embedded parens. Only a closed paren at depth 0 is considered a match. The **more** variable causes the loop to exit when it becomes false. The **instring** variable is used to ignore parens embedded in strings. The **str** variable is the current string we are walking. The complication is that there could be newlines between balancing parens so we have to move to the next string in the linelist. The **c** character is the current character we are considering. There are several cases with associated comments. The **listlen** variable is a counter which counts how much of the string we need to remove before returning. The complication is that when we have multiple lines we have to also consider the leading spaces in indents.

The **FSM-gatherList 1** error occurs because the next character whould have been an open parenthsis. The function expects to find matching parentheses and it needs an initial parenthesis to start the balance. This error usually means that the category, domain, or package name needs an explicit argument list even it it takes no arguments.

FSM-gatherList : **linelist** $\rightarrow$ **(linelist,list(args)**
— defun FSM-gatherList —

```
(defvar formatGatherList1
  "FSM-gatherList 1: ~%    In ~a~%    Missing open parenthesis~%")

(defun FSM-gatherList (linelist)
  (let (result (depth 0) (more t) instring str c (listlen 0))
     (setq str (explode (first linelist)))
     (unless (char= (first str) #\()
       (error formatGatherList1 (first linelist)))
     (loop while more do
       (setq c (pop str))
       (cond
         ; hit a newline?
         ((null c)
           (pop linelist)
           (setq listlen (indent (first linelist)))
           (setq str (explode (trim (first linelist))))
           (push #\space str))
         ; end of string?
         ((and instring (char= c #\"))
           (setq instring nil)
           (incf listlen)
           (push c result))
         ; still in string?
         (instring
           (incf listlen)
           (push c result))
         ; start string?
         ((and (not instring) (char= c #\"))
           (setq instring t)
           (incf listlen)
           (push c result))
         ; start list?
         ((char= c #\()
           (incf depth)
           (incf listlen)
           (push #\( result))
         ; exit nested list?
         ((and (> depth 0) (char= c #\)))
           (decf depth)
           (when (= depth 0) (setq more nil))
           (incf listlen)
           (push #\) result))
         ; end list?
         ((and (= depth 0) (char= c #\)))
           (incf listlen)
           (push #\) result)
           (setq more nil))
         (t
           (incf listlen)
           (push c result))))
    (setq result (coerce (nreverse result) 'string))
    (setf (first linelist) (subseq (first linelist) listlen))
    (values linelist result)
))
```

———–

The function expects a list of arguments to a function. It breaks up the list based on the top-level commas, ignoring commas in strings and in embedded arguments that are not top-level. It returns a list of strings, one per top level argument.

FSM-gatherVars : **argslist** → **list(vars)**
— defun FSM-gatherVars —

```
;  ARGLIST -> LIST(ARGS)
(defun FSM-gatherVars (argslist)
  (let (result (depth 0) (more t) instring str c vars)
     (setq str (explode (trim argslist)))
     (pop str)
     (loop while more do
       (setq c (pop str))
       (cond
         ; hit a newline?
         ((null c)
  (print (list 'gv 0))
           (pop argslist)
           (if argslist
             (progn
               (setq str (explode (trim (first argslist))))
               (push #\space str))
             (progn
               (push (trim (coerce (nreverse result) 'string)) vars)
               (setq more nil))))
         ; found a top-level comma?
         ((and (= depth 0) (not instring) (char= c #\,))
  (print (list 'gv 1))
           (push (trim (coerce (nreverse result) 'string)) vars)
           (setq result nil))
         ; end of string?
         ((and instring (char= c #\"))
  (print (list 'gv 2))
           (setq instring nil)
           (push c result))
         ; still in string?
         (instring
  (print (list 'gv 3))
           (push c result))
         ; start string?
         ((and (not instring) (char= c #\"))
  (print (list 'gv 4))
           (setq instring t)
           (push c result))
         ; start list?
         ((char= c #\()
  (print (list 'gv 5))
           (incf depth)
           (push #\( result))
         ; exit nested list?
         ((and (> depth 0) (char= c #\)))
  (print (list 'gv 6))
           (decf depth)
           (push #\) result))
         ; end list?
         ((and (= depth 0) (not instring) (char= c #\)))
  (print (list 'gv 7 'result result))
           (when result (push (trim (coerce (nreverse result) 'string)) vars))
```

```
          (setq more nil))
        (t
  (print (list 'gv 8))
          (push c result))))
    (nreverse vars)
))
```

————-

1.32 Sourcecode Tree Utilities

We process the original source code into a "clean form" that has no −− comments, no blank lines, and no line continuations. We call this the "bf pile" form of the code. This is just a list of strings.

Next we convert the pile form into tree form using the indentation. Elements at the same level of identation are grouped togehter.

The **indent** function tells us how many spaces, aka the indentation, we have at the beginning of the line.

indent : **line** $\rightarrow$ **depth**

— defun indent —

```
(defun indent (line)
  (let ((depth 0))
    (loop for char across line do
      (if (char= char #\space)
        (incf depth)
        (return)))
    depth))
```

————-

Just for mnemonic purposes we create constructors and accessor functions. All we are really doing is creating pairs, the car of which is the indentation and the cdr is anything.

The **birth** function expects the indentation count and anything, and just conses them together.

```
   DEPTH, ANY -> BIRTHFORM
```

— defmacro birth —

```
(defmacro birth (depth name)
 `(cons ,depth ,name))
```

————-

Naturally, if we have a cons, we need to disassemble it. The **depthof** function returns the indentation.

```
   BIRTHFORM -> DEPTH
```

— defmacro depthof —

```
(defmacro depthof (node)
 `(car ,node))
```

————-

And the **nameof** returns the thing we put in the cdr.

```
   BIRTHFORM -> ANY
```

— defmacro nameof —

```
(defmacro nameof (node)
 `(cdr ,node))
```

————-

Given a list of strings, each of which has a variable number of leading blanks, we construct pairs where the car is the indentation and the cdr is anything, usually the source string. We call this rewrite the **spawn** of the tree. From this information we can compute things like the depth of nesting.

```
   PILEFORM -> SPAWNFORM
```

spawn : **tree** $\rightarrow$ **list(indentedLines)**
— defun spawn —

```
(defun spawn (tree)
  (loop for node in tree
    collect (birth (indent node) node)))
```

————-

The **pile2tree** function takes a **sourcecode pile** which looks like:

```
(")abbrev domain BITS Bits"
 "++ Author: Stephen M. Watt" "++ Description:"
 "++ \\spadtype{Bits} provides logical functions for Indexed Bits."
 "Bits() : SIG == CODE where"
 "  SIG ==> BitAggregate() with"
 "    bits : (NonNegativeInteger, Boolean) -> %"
 "      ++ bits(n,b) creates bits with n values of b"
 "  CODE ==> IndexedBits(1) add"
 "    bits(n,b)    == new(n,b)")
```

and inserts parentheses that group the code into a tree form, called a **sourcecode tree** looks like:

```
(")abbrev domain BITS Bits"
 "++ Author: Stephen M. Watt" "++ Description:"
 "++ \\spadtype{Bits} provides logical functions for Indexed Bits."
 "Bits() : SIG == CODE where"
 "  (SIG ==> BitAggregate() with"
 "    (bits : (NonNegativeInteger, Boolean) -> %"
 "      (++ bits(n,b) creates bits with n values of b"
 "      )))"
 "  (CODE ==> IndexedBits(1) add"
 "    (bits(n,b)    == new(n,b)"
 "    ))")
```

The **deep** function will walk a spawn-ed tree and return a reverse sorted list of the indentations. We use this to walk the spawn tree "inside-out", handling the most deeply nested nodes first.

pile2tree : **pileform** → **treeform**
— defun pile2tree —

```
(defun pile2tree (pile)
  (labels (
    ; INTEGER -> STRING of PARENS
    (closeparens (count)
      (let (result)
        (coerce (dotimes (i count result) (push #\) result)) 'string)))
    ; INTEGER -> STRING of SPACES
    (spaces (count)
      (let (result)
        (coerce (dotimes (i (* 2 count) result) (push #\space result))
'string)))
    ; SPAWNFORM -> LIST DEPTH
    (deep (tree)
      (let (result)
        (loop for node in tree do
          (setq result (adjoin (depthof node) result)))
        (sort result #'>)))
    ; PILE -> PILE
    (redepth (pile)
     (let (deeplist thepile)
      (setq thepile (spawn pile))
      (setq deeplist (nreverse (deep thepile)))
      (loop for line in thepile do
        (setf (car line) (position (car line) deeplist)))
      thepile))
    ; PILE -> LIST STRING
    (extract (code)
      (loop for line in code
        collect (cdr line)))
  )
  (let ((was 0) lastline closers)
    (let (stack)
      (loop for line in (redepth pile) do
        (setf (cdr line) (trim (cdr line)))
        (cond
          ((= (depthof line) was)
            (setf (cdr line) (concatenate 'string (spaces (car line)) " " (cdr line)))
            (push line stack))
          ((> (depthof line) was)
            (setf (cdr line) (concatenate 'string (spaces (car line)) "(" (cdr line)))
            (push line stack))
          ((< (depthof line) was)
            (setq closers (closeparens (car lastline)))
            (push (cons (car lastline) (concatenate 'string (spaces (car lastline)) closers)) stack)
            (setf (cdr line) (concatenate 'string (spaces (car line)) "(" (cdr line)))
            (push line stack))
          )
        (setq was (car line))
        (setq lastline line))
    (setq closers (closeparens (car lastline)))
    (push (cons (car lastline) (concatenate 'string (spaces (car lastline)) closers)) stack)
    (extract (nreverse stack))))))
```

———-

The **pretty** function takes a sourcecode-tree, or a sublist thereof, and prints it in a form we prefer.

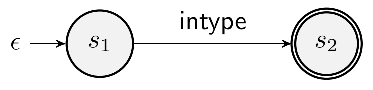
pretty : **treeform** → **values**
— defun pretty —

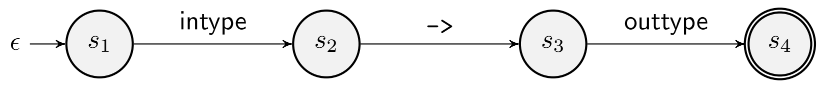
```
(defun pretty (tree)
  (cond
    ((stringp tree) (format t "~a~%" tree))
    ((consp tree)   (mapcar #'(lambda (tree) (pretty tree)) tree)))
  (values))
```

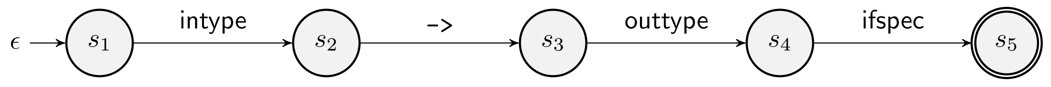
———-

Spad code is indentation sensitive. This is called 'pile format'. We unpile the spad file, recording the indentation.

```
FILE        ::= ABBREV BODY
BODY        ::= CDPSIG MACROS WITH ADD
CDPSIG      ::= CDPNAME ['(' [ CDPARGS ] ')'] '=='
MACROS      ::= MACNAME '==>' MACBODY
WITH        ::= [PARENTS] with SIGNATURES
ADD         ::=

COLONTYPE   ::= name : TYPESPEC [',' COLONGTYPE]*
RECORD      ::= 'Record(' COLONTYPE* ')'
UNIONTAG    ::= TYPESPEC | STRING
UNION       ::= 'Union(' UNTIONTAG [',' UNIONTAG]+ ')'

INFIX       ::= + | - * | ** | mod | ^ | / | exquo
PREFIX      ::= + | -
SPECIALNAME ::= "*" | "**" | "^" | "/" | PREFIX | INFIX
SIGNAME     ::= SPECIALNAME | SYMBOL

FSMSIG      ::= INDENT SIGNAME [':' SIGINTYPE '->' SIGOUTTYPE] [SIGIFSEC]
```







```
FSMSIG ::=
```

Figure 1.10: FSMSIG syntax

TODO

Figure 1.11: LEAVE syntax

```
LEAVE
```

```
  FUNCTIONSPEC ::= INTTYPE
                 | INTYPE -> OUTPUT
                 | INTYPE -> OUTPUT IFSPEC
```

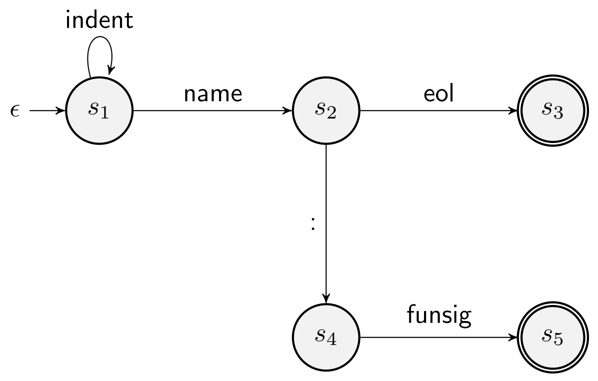


Figure 1.12: SIGNATURE

```
  SIGNATURE ::= NAME EOF
              | NAME : FUNCTIONSPEC
```

The Finite State Machine (FSM) walks across a spad signature character by character. It constructs the syntactic pieces.

The inner loop drives forward character by character, calling the **classify** function to determine the next state. The inner loop is a large case statement handling each classified state. The inner loop is **:accept** at the end of the file.

Both **inttype**, the input type specification and **outtype**, the output type specification may both be very complex type specifications. They are recognized by a separate finite state machine.

A signature may also have a conditional specification that is handled by its own finite state machine.

The **look** function just peeks at the next character. The **eat** function consumes it.

The **next** function updates the **nextstate** variable.

The **here** function outputs debugging information.

FSMSIG : **debug** → **values**
— defun FSMSIG —

```
(defun FSMSIG (&optional (debug nil))
  (let (nextstate prev char (eofsym (gensym))
        (indenting? t) (indent 0)
        inword? word words)
    (labels (
      (classify (char)
        (unless (eq char #\space) (setq indenting? nil))
        (cond
         ((eq char eofsym)    (next :accept))
         ((eq char #\space)   (next :space))
         ((eq char #\:)       (next :colon))
         ((eq char #\$)       (next :dollar))
         ((eq char #\-)       (next :dash))
         ((eq char #\>)       (if (char= #\- prev) (next :arrow) (next :greater)))
         ((lower-case-p char) (next :lowercase))
         ((upper-case-p char) (next :uppercase))
      ))
      (look (in)
        (setq prev char)
        (setq char (peek-char nil in nil eofsym)))
      (eat (in)
        (setq prev char)
        (setq char (read-char in nil eofsym)))
      (next (newstate)
        (setq nextstate newstate))
      (endword () (print 'endword)
        (push (coerce (nreverse word) 'string) words)
        (setq inword? nil)
        (setq word nil))
      (here (&optional (keyword nil))
        (when debug
          (if keyword
            (format t "~&~a ~a" char keyword)
            (format t "~&~a ~a" char nextstate))))
    )
  (let ()
    (with-open-file (spad "spad" :direction :input)
      (loop until (eq nextstate :accept) do
        (look spad)
        (case (classify char)
          (:space (here)
            (when indenting? (incf indent))
            (when inword? (endword))
            (eat spad))
         (:lowercase (here)
           (setq inword? t)
           (push char word)
           (eat spad))
         (:uppercase (here)
           (setq inword? t)
           (push char word)
           (eat spad))
         (:colon (here)
           (when inword? (endword))
           (eat spad))
         (:dollar (here)
           (push char word)
           (eat spad))
         (:dash (here)
           (push char word)
           (eat spad))
         (:arrow (here)
           (setq prev (pop word))
           (endword)
           (push prev word)
           (push char word)
           (endword)
           (eat spad))
         (:accept (here :accept)
           (endword))
         (t (here :fail)
            (endword)
            (next :accept))
       )))
    (print (list 'indent indent
                 'words (nreverse words)))
    (values)
  ))))
```

————-

1.32.1 Gather Macros

The **FSM-extract1Macro** walks the linelist until it finds a line containing the string "==>" which marks the start of a macro. All lines (including the first) are removed from the linelist and collected into a **gather** structure. The **pre** variable contains the lines up to the start of the macro. Ultimately, we return a new linelist without the macro lines and a gather structure containing the macro lines.

FSM-extract1Macro : **linelist** → **(linelist,gather)**
— defun FSM-extract1Macro —

```
(defun FSM-extract1Macro (linelist)
  (let (pre gather indent)
    (setq pre
      (loop while (not (search "==>" (first linelist)))
            collect (pop linelist)))
    (setq gather (make-instance 'gather))
    (when (search "==>" (first linelist))
      (setq indent (indent (first linelist)))
      (setf (gather-lines gather) (list (pop linelist)))
      (FSM-gather linelist (> (indent (first linelist)) indent) gather))
    (values
      (append pre linelist)
      gather)))
```

———-

1.32.2 parseSignature

parseSignature : **linelist** → **values**
— **defun parseSignature** —

```
(defun parseSignature (linelist)
  (let (funcname return argslist comment colon arrow line control plusplus)
    (labels (
      (substring (line start &optional end)
        (trim (subseq line start end)))
      (nospad (line)
        (let (pos)
          (cond
            ((setq pos (search "spad{" line))
              (concatenate 'string
                (subseq line 0 pos)
                (subseq line (+ 5 pos) (setq pos (search "}" line :start2 pos)))
                (subseq line (1+ pos))))
            ((setq pos (search "spadtype{" line))
              (concatenate 'string
                (subseq line 0 pos)
                (subseq line (+ 9 pos) (setq pos (search "}" line :start2 pos)))
                (subseq line (1+ pos))))
           (t line))))
      (mangle (str)
        (cond
          ((string= str "()") str)
          ((string= str "%") str)
          (t (intern str))))
    )
    ; the first line is the signature
    (setq line     (first linelist))
    ; find the delimiters if they exist
    (setq colon    (search ":" line))
    (setq arrow    (search "->" line))
    ; parse out the pieces
    (setq funcname (intern (substring line 0 colon)))
    (setq argslist  (mangle (substring line (1+ colon) arrow)))
    (setq return   (mangle (substring line (+ 2 arrow))))
    ; collect the comment linelist
    (dolist (line (cdr linelist))
      (setq plusplus (search "++" line))
      (push (nospad (substring line (+ 2 plusplus))) comment))
    ; construct a format control string
    (setq control "(make-signature '|~a|"))
    (cond
       ((string= return "()")
         (setq control (concatenate 'string control " '~a ")))
       ((string= return "%")
         (setq control (concatenate 'string control " '(~a) ")))
       (t
         (setq control (concatenate 'string control " '(|~a|)"))))
    (cond
       ((string= argslist "()")
         (setq control (concatenate 'string control " '~a ")))
       ((string= argslist "%")
         (setq control (concatenate 'string control " '(~a) ")))
       (t
         (setq control (concatenate 'string control " '(|~a|)"))))
    (setq control (concatenate 'string control " '~s)~%"))
    (format t control funcname return argslist (nreverse comment))
    (values)))
```

———–

Common Lisp Types

Class	Class Precedence List
array	array t
bit-vector	bit-vector vector array sequence t
character	character t
complex	complex number t
cons	cons list sequence t
float	float number t
integer	integer rational number t
list	list sequence t
null	null symbol list sequence t
number	number t
ratio	ratio rational number t
rational	rational number t
sequence	sequence t
string	string vector array sequence t
symbol	symbol t
t	t
vector	vector array sequence t

EBNF

1.33 For show output

```
PARAM ::= UPPERCASELETTER

DOMAINSPEC ::= DOMAIN
             | DOMAIN(%)
             | DOMAIN(PARAM)
             | DOMAIN(DOMAIN[,DOMAIN]*)

HASSPEC ::= 'if' '$' 'has' CATEGORY
          | 'if' PARAM 'has' CATEGORY
          | 'if' DOMAINSPEC 'has' CATEGORY
          | 'if' PARAM 'has' DOMAINSPEC

ANDSPEC ::= HASSPEC 'and' HASSPEC

ORSPEC ::= HASSPEC
         | HASSPEC 'or' ANDSPEC
         | ANDSPEC 'or' HAPSPEC
         | ANDSPEC 'or' ANDSPEC
         | ORSPEC 'or' HASSPEC
         | ORSPEC 'or' ANDSPEC
         | ORSPEC 'or' ORSPEC

IFSPEC ::= HASSPEC
         | ANDSPEC
         | ORSPEC

FUNCSPEC ::= '(' PARAM '->' PARAM ')'

DPSPEC ::= '%'
         | PARAM
         | DOMAINSPEC
         | FUNCSPEC

TWOLIST ::= '(' DPSPEC ',' DPSPEC ')'

CONSTANT ::= '0'
           | '1'

PREFIX ::= '#?'
         | '-?'

INFIX ::= '?**?'
        | '?*?'
        | '?-?'

UNIONSPEC ::= 'Union' '(' DPSPEC ',' '"failed"' ')'

KEYSPEC ::= Symbol ':' DOMAIN
          | Symbol ':' PARAM

RECORDSPEC ::= 'Record' '(' KEYSPEC [',' KEYSPEC]* ')'

CONSTANT ':' '()'    '->' '%'
CONSTANT ':' '()'    '->' '%' IFSPEC
PREFIX   ':' DPSPEC  '->' DPSPEC
PREFIX   ':' DPSPEC  '->' DPSPEC IFSPEC
INFIX    ':' TWOLIST '->' DPSPEC
INFIX    ':' TWOLIST '->' DPSPEC IFSPEC
INFIX    ':' TWOLIST '->' UNIONSPEC
'?..?' ':' '(' '%' ',' '%' ')' '->' '%'
'?..?' ':' '(' 'S' ',' 'S' ')' '->' '%'
'?.?'  ':' '(' '%' ',' DPSPEC ')' '->' DPSPEC
'?.count'  ':' '(' '%' ',' 'count' ')' '->' DPSPEC
'?.first  ':' '(' '%' ',' 'first' ')' '->' RECORDSPEC
'?.last  ':' '(' '%' ',' 'last' ')' '->' DPSPEC
'?.last  ':' '(' '%' ',' 'last' ')' '->' DP
'?.last  ':' '(' '%' ',' 'last' ')' '->' RECORDSPEC
'?.last  ':' '(' '%' ',' 'last' ')' '->' PARAM
'?.left  ':' '(' '%' ',' 'left' ')' '->' '%'
'?.rest  ':' '(' '%' ',' 'rest' ')' '->' '%'
'?.right  ':' '(' '%' ',' 'right' ')' '->' '%'
'?.sort  ':' '(' '%' ',' 'sort' ')' '->' '%'
'?.unique  ':' '(' '%' ',' 'unique' ')' '->' '%'
'?.value  ':' '(' '%' ',' 'value' ')' '->' RECORDSPEC
'?.value  ':' '(' '%' ',' 'value' ')' '->' PARAM
'?.value  ':' '(' '%' ',' 'value' ')' '->' DOMAINSPEC
```

The Compiler

Sane Interpreter

— defmacro must —

```
(defmacro must (FSMname input)
  `(let (text)
     (if (setq text ,FSMname ,input)
       (progn (format t "ACCEPT ~a~%" text) t)
       (progn (format t "FAILED ~a~%" text) nil))))
```

———-

— defmacro expect —

```
(defmacro expect (FSMname input)
  `(let (text)
     (if (setq text ,FSMname ,input)
       (format t "ACCEPT ~a~%" text)
       (format t "FAILED ~a~%" text))))
```

———-

This is the top loop of the Sane interpreter.

It defines a function **bye** which only exists for the lifetime of the interpreter. When called from the Sane command line, **bye** returns the gensym. If that gensym is seen the Sane interpreter makes the **bye** unbound and exits. Note that we save any existing definition of **bye** and restore it when the interpreter exits.

sane : **nil** → **nil**
— defun sane —

```
(defun sane ()
  (let (input result)
    (labels (
      (printType (thing)
        (typecase thing
         (string "String")
         (array "Array")
         (bit-vector "Bit-vector")
         (character "Character")
         (complex "Complex")
         (null "Null")
         (list "List")
         (cons "Cons")
         (float "Float")
         (integer "Integer")
         (rational "Rational")
         (number "Number")
         (ratio "Ratio")
         (sequence "Sequence")
         (t "t")
         (symbol "Symbol")
         (vector "Vector")))
      (localHelp ()
        ; define a help command
        (setq result "beyond it"))
    )
    (loop do
      (format t "S> ")
      (force-output)
      (setq input (read))
      (cond
        ((eq input 'bye) (return))
        ((eq input 'help) (localHelp))
        (t (setq result (eval input))))
      (format t "  ~s~%       Type: ~a~%" result (printType result))
      (terpri)))))
```

———————-

Axioms and Logic Rules

Dieudonné suggests that casual reasoning is a childhood disease of mathematical areas.
– Arthur Jaffe and Frank Quinn [Jaff93]

What mathematicians are accomplishing is to advance human understanding of mathematics.
– William P. Thurston [Thur94]

If you make the tools, you make the rules
– Andrew Dana Hudson [Huds19]

Collected here are some axioms that need to be distributed among the various categories so that they can be inherited at the proper time.

There is an interesting design question.

Axiom already has a mechanism for inheriting signatures from categories. That is, we can get a plus signature from, say, the Integer category.

Suppose we follow the same pattern. Currently Axiom inherits certain so-called "attributes", such as ApproximateAttribute, which implies that the computation results are only approximate.

We could adapt the same mechanism to inherit the Transitive property. In fact, if we follow the "tiny theory" approach of Carette and Farmer where each property is in its own inheritable category, then we can "mix and match" the axioms at will.

An axiom category would also export a function. This function would essentially be a "tactic" used in a proof. It would modify the proof step by applying the function to the step.

Theorems could have the same structure.

This allows theorems to be constructed at run time (since Axiom supports "First Class Dynamic Types".

In addition, this design can be "pushed down" into the Spad language so that Spad statements (e.g. assignment) had proof-related properties. A ramge such as [1..10] would provide explicit bounds in a proof "by language definition". Defining the logical properties of language statements in this way would make it easier to construct proofs since the invariants would be partially constructed already.

The design merges the computer algebra inheritance structure with the proof of algorithms structure, all under the same mechanism.

Reflexivity

$$\frac{}{e = e}$$

Symmetry

$$\frac{e_2 = e_1}{e_1 = e_2}$$

Transitivity

$$\frac{e_1 = e_3 \qquad e_3 = e_2}{e_1 = e_2}$$

Congruence

$$\frac{f = f' \qquad e_1 = e_1' \qquad \ldots \qquad e_n = e_n'}{f(e_1, \ldots, e_n) = f'(e_1', \ldots, e_n')}$$

For SemiRings we know that

$$\begin{array}{rcl}
(a+b)+c & = & a+(b+c) \\
0+a & = & a \\
a+0 & = & a \\
a+b & = & b+a \\
(a \times b) \times c & = & a \times (b \times c) \\
1 \times a & = & a \\
a \times 1 & = & a \\
a \times (b+c) & = & (a \times b) + (a \times c) \\
(a+b) \times c & = & (a \times c) + (b \times c) \\
0 \times a & = & 0 \\
a \times 0 & = & 0
\end{array}$$

For Rng we have

An rng is a set R with two binary operations $(+, -)$ such that

- $(R, +)$ is an abelian group
- $(R, -)$ is a semigroup
- Multiplication distributes over additin
- $f(x+y) = f(x) + f(y)$
- $f(x * y) = f(x) * f(y)$

An rng does not have an identity element.

Variable

$$\langle x^\tau, \sigma, \Gamma \rangle \to \langle \sigma(x^\tau), \sigma, \Gamma \rangle$$

Call-Arguments

$$\frac{\langle e_i, \sigma, \Gamma \rangle \to \langle e_i', \sigma', \Gamma \rangle}{e_0(v_1, \ldots, e_i, \ldots, e_n), \sigma, \Gamma \rangle \to \langle e_0(v_1, \ldots, e_i', \ldots, e_n), \sigma', \Gamma \rangle}$$

where the x_i are parameters of v_0

Call-Operator

$$\frac{\langle e_0, \sigma, \Gamma\rangle \to \langle e_0, \sigma', \Gamma\rangle}{\langle e_0(v_1, \ldots, v_n), \sigma, \Gamma\rangle \to \langle e_0'(v_1, \ldots, v_n), \sigma', \Gamma\rangle}$$

where the x_i are the parameters of v_0

Call

$$\langle v_0^{\tau_0}(v_1^{\tau_1}, \ldots, v_n^{\tau_n}), \sigma, \Gamma\rangle \to \langle v_0^{\tau_0}[v_1^{\tau_1}/x_1, \ldots, v_n^{\tau_n}/x_n], \sigma, \Gamma\rangle$$

where the x_i are parameters of v_0

Qualified Expression

$$\frac{\begin{array}{c}\langle \delta_1, \sigma, \Gamma_1\rangle \to \langle \delta_1', \sigma, \Gamma_2\rangle \qquad \langle \delta_2, \sigma, \Gamma_2\rangle \to \langle \delta_2', \sigma, \Gamma_3\rangle \\ \cdots \quad \langle \delta_n, \sigma, \Gamma_n\rangle \to \langle \delta_n', \sigma, \Gamma_{n+1}\rangle \quad \langle e, \sigma, \Gamma_{n+1}\rangle \to \langle e', \sigma', \Gamma_{n+2}\rangle\end{array}}{\langle e \text{ where } \delta_1 \cdots \delta_n, \sigma, \Gamma_1\rangle \to \langle e', \sigma', \Gamma_{n+2}\rangle}$$

Sequence-Head

$$\frac{\langle s_1, \sigma_1, \Gamma_1\rangle \to \langle s_1', \sigma_2, \Gamma_2\rangle}{\langle s_1; s_2, \sigma_1, \Gamma_1\rangle \to \langle s_1'; s_2, \sigma_2, \Gamma_2\rangle}$$

Sequence-Tail

$$\langle v_1; s_2, \sigma, \Gamma\rangle \to \langle s_2, \sigma, \Gamma\rangle$$

If-True

$$\frac{\langle s_1, \sigma, \Gamma\rangle \to \langle s_1', \sigma', \Gamma'\rangle}{\langle \text{ if true then } s_1; s_2, \sigma, \Gamma\rangle \to \langle s_1', \sigma', \Gamma'\rangle}$$

If-False

$$\frac{\langle s_2, \sigma, \Gamma\rangle \to \langle s_2', \sigma', \Gamma'\rangle}{\langle \text{ if false then } s_1; s_2, \sigma, \Gamma\rangle \to \langle s_2', \sigma', \Gamma'\rangle}$$

Assignment-Left

$$\frac{\langle e_1, \sigma_0, \Gamma\rangle \to \langle e_1', \sigma_1, \Gamma\rangle}{\langle e_1 := e_2, \sigma, \Gamma\rangle \to \langle e_1' := e_2, \sigma_1, \Gamma\rangle}$$

Assignment-Right

$$\frac{\langle e_2, \sigma, \Gamma\rangle \to \langle e_2', \sigma', \Gamma\rangle}{\langle l := e_2, \sigma, \Gamma\rangle \to \langle l := e_2', \sigma', \Gamma\rangle}$$

Assignment

$$\frac{}{\langle l := v^\tau, \sigma, \Gamma\rangle \to \langle v^\tau, \sigma[v^\tau/l], \Gamma\rangle}$$

Immediate Definition

$$\frac{\langle e, \sigma, \Gamma\rangle \to \langle e', \sigma_1, \Gamma\rangle}{\langle x : \tau := e, \sigma, \Gamma\rangle \to \langle x : \tau := e', \sigma_1, \Gamma\rangle}$$

Immediate Definition

$$\langle x : \tau := v^\tau, \sigma, \Gamma\rangle \to \langle v^\tau, \sigma, \Gamma, x^\tau == v^\tau\rangle$$

The Categories

One of the primary concerns is **Coherence** [Bott19] of these type definitions. Because we are using CLOS the system automatically computes the coherence between types.

Axiom computes type resolution based on *modemaps* but the talk by Bottu and Mardirosian [Bott19] does type resolution by creating unique names instead.

There are patterns in the following code. Violating these patterns will generate obscure bugs so be careful. Lets assume a given class.

```
(defclass |GcdDomainType| (|LeftOreRingType| |IntegralDomainType|)
  ((parents :initform '(|LeftOreRing| |IntegralDomain|))
   (name :initform "GcdDomain")
   (level :initform 12)
   (abbreviation :initform 'GCDDOM)))

(defvar |GcdDomain|
  (progn
    (push '|GcdDomain| *Categories*)
    (make-instance '|GcdDomainType|)))
```

Notice the following properties:

The classname **GcdDomainType** is escaped so that it will retain its case no matter which lisp is used.

The classname **GcdDomainType** ends with "Type".

The classes it depends on all end with "Type".

The parent attribute list is in the same order as the list used for inheritance. Failure to do this will result in an incorrect ordering of lookups.

The parent attributes do NOT end in with "Type" as they are instances. Putting "Type" in the name will result in an obscure circular reference error at compile time.

The name is given at the class level. It must be provided as it is used in printing the object.

The level attribute is read-only.

The **GcdDomain** variable adds its name to the global variable *Categories* so we maintain a current list.

We need to call make-instance at compile time in order to force MOP::finalize-inheritance, otherwise the compiler will find some random reason to complain.

1.34 Level 1

This is a searchable list of all of the Categories in level 1.

— defvar level1 —

```
(defvar level1
 '(|AdditiveValuationAttribute| |ApproximateAttribute|
   |ArbitraryExponentAttribute| |ArbitraryPrecisionAttribute|
   |ArcHyperbolicFunctionCategory| |ArcTrigonometricFunctionCategory|
   |AttributeRegistry| |BasicType| |CanonicalAttribute|
   |CanonicalClosedAttribute| |CanonicalUnitNormalAttribute| |CentralAttribute|
   |CoercibleTo| |CombinatorialFunctionCategory| |CommutativeStarAttribute|
   |ConvertibleTo| |ElementaryFunctionCategory| |Eltable|
   |FiniteAggregateAttribute| |HyperbolicFunctionCategory| |InnerEvalable|
   |JacobiIdentityAttribute| |LazyRepresentationAttribute| |LeftUnitaryAttribute|
   |ModularAlgebraicGcdOperations| |MultiplicativeValuationAttribute|
   |NoZeroDivisorsAttribute| |NotherianAttribute| |NullSquareAttribute|
   |OpenMath| |PartialTranscendentalFunctions| |PartiallyOrderedSetAttribute|
   |PrimitiveFunctionCategory| |RadicalCategory| |RetractableTo|
   |RightUnitaryAttribute| |ShallowlyMutableAttribute| |SpecialFunctionCategory|
   |TrigonometricFunctionCategory| |Type| |UnitsKnownAttribute|))
```

———-

1.34.1 AdditiveValuationAttribute

— defclass AdditiveValuationAttributeType —

```
(defclass |AdditiveValuationAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AdditiveValuationAttributeType")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATADDVA)
   (comment :initform (list
     "The class of all euclidean domains such that"
     "euclideanSize(a*b) = EuclideanSize(a)+euclideanSize(b)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AdditiveValuationAttribute|
  (progn
    (push '|AdditiveValuationAttribute| *Categories*)
    (make-instance '|AdditiveValuationAttributeType|)))
```

———-

1.34.2 ApproximateAttribute

— defclass ApproximateAttributeType —

```
(defclass |ApproximateAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ApproximateAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATAPPRO)
   (comment :initform (list
     "An approximation to the real numbers."))))

(defvar |ApproximateAttribute|
  (progn
    (push '|ApproximateAttribute| *Categories*)
    (make-instance '|ApproximateAttributeType|)))
```

———-

1.34.3 ArbitraryExponentAttribute

— defclass ArbitraryExponentAttributeType —

```
(defclass |ArbitraryExponentAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ArbitraryExponentAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATARBEX)
   (comment :initform (list
     "Approximate numbers with arbitrarily large exponents"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ArbitraryExponentAttribute|
  (progn
    (push '|ArbitraryExponentAttribute| *Categories*)
    (make-instance '|ArbitraryExponentAttributeType|)))
```

———-

1.34.4 ArbitraryPrecisionAttribute

— defclass ArbitraryPrecisionAttributeType —

```
(defclass |ArbitraryPrecisionAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ArbitraryPrecisionAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATARBPR)
   (comment :initform (list
     "Approximate numbers for which the user can set the precision"
     "for subsequent calculations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ArbitraryPrecisionAttribute|
  (progn
    (push '|ArbitraryPrecisionAttribute| *Categories*)
    (make-instance '|ArbitraryPrecisionAttributeType|)))
```

———-

1.34.5 ArcHyperbolicFunctionCategory

— defclass ArcHyperbolicFunctionCategoryType —

```
(defclass |ArcHyperbolicFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ArcHyperbolicFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'AHYP)
   (comment :initform (list
     "Category for the inverse hyperbolic trigonometric functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|acosh| '(%) '(%)
       '("acosh(x) returns the hyperbolic arc-cosine of x."))
     (make-signature '|acoth| '(%) '(%)
       '("acoth(x) returns the hyperbolic arc-cotangent of x."))
     (make-signature '|acsch| '(%) '(%)
       '("acsch(x) returns the hyperbolic arc-cosecant of x."))
     (make-signature '|asech| '(%) '(%)
       '("asech(x) returns the hyperbolic arc-secant of x."))
     (make-signature '|asinh| '(%) '(%)
       '("asinh(x) returns the hyperbolic arc-sine of x."))
     (make-signature '|atanh| '(%) '(%)
       '("atanh(x) returns the hyperbolic arc-tangent of x."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ArcHyperbolicFunctionCategory|
  (progn
    (push '|ArcHyperbolicFunctionCategory| *Categories*)
    (make-instance '|ArcHyperbolicFunctionCategoryType|)))
```

———-

1.34.6 ArcTrigonometricFunctionCategory

— defclass ArcTrigonometricFunctionCategoryType —

```
(defclass |ArcTrigonometricFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ArcTrigonometricFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATRIG)
   (comment :initform (list
     "Category for the inverse trigonometric functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|acos| '(%) '(%)
       '("acos(x) returns the arc-cosine of x. When evaluated"
         "into some subset of the complex numbers, one"
         "branch cut for acos lies along the negative real axis"
         "to the left of -1 (inclusive), continuous with the"
         "upper half plane, the other along the positive real axis to"
         "the right of 1 (inclusive), continuous with the lower half"
         "plane."))
     (make-signature '|acot| '(%) '(%)
       '("acot(x) returns the arc-cotangent of x."))
     (make-signature '|acsc| '(%) '(%)
       '("acsc(x) returns the arc-cosecant of x."))
     (make-signature '|asec| '(%) '(%)
       '("asec(x) returns the arc-secant of x."))
     (make-signature '|asin| '(%) '(%)
       '("asin(x) returns the arc-sine of x. When evaluated into some"
         "subset of the complex numbers, one branch cut for asin lies"
         "along the negative real axis to the left of -1 (inclusive),"
         "continuous with the upper half plane, the other along the"
         "positive real axis to the right of 1 (inclusive), continuous"
         "with the lower half plane."))
     (make-signature '|atan| '(%) '(%)
       '("atan(x) returns the arc-tangent of x. When evaluated into some"
         "subset of the complex numbers, one branch cut for atan lies"
         "along the positive imaginary axis above %i (exclusive),"
         "continuous with the left half plane, the other along the"
         "negative imaginary axis below -%i (exclusive) continuous"
         "with the right half plane. The domain does not contain %i and -%i"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ArcTrigonometricFunctionCategory|
  (progn
    (push '|ArcTrigonometricFunctionCategory| *Categories*)
    (make-instance '|ArcTrigonometricFunctionCategoryType|)))
```

———-

1.34.7 AttributeRegistry

— defclass AttributeRegistryType —

```
(defclass |AttributeRegistryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AttributeRegistry")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATTREG)
   (comment :initform (list
     "This category exports the attributes in the AXIOM Library"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|finiteAggregate| () ()
       '("finiteAggregate is true if it is an aggregate with a"
         "finite number of elements."))
     (make-signature '|commutative("*")| () ()
       '("commutative(\"*\") is true if it has an operation"
         "\"*\": (D,D) -> D} which is commutative."))
     (make-signature '|shallowlyMutable| () ()
       '("shallowlyMutable is true if its values"
         "have immediate components that are updateable (mutable)."
         "Note that the properties of any component domain are"
         "irrevelant to the shallowlyMutable proper."))
     (make-signature '|unitsKnown| () ()
       '("unitsKnown is true if a monoid (a multiplicative semigroup"
         "with a 1) has unitsKnown means that"
         "the operation recip can only return \"failed\""
         "if its argument is not a unit."))
     (make-signature '|leftUnitary| () ()
       '("leftUnitary is true if 1 * x = x for all x."))
     (make-signature '|rightUnitary| () ()
       '("rightUnitary is true if x * 1 = x for all x."))
     (make-signature '|noZeroDivisors| () ()
       '("noZeroDivisors is true if x * y ~= 0 implies"
         "both x and y are non-zero."))
     (make-signature '|canonicalUnitNormal| () ()
       '("canonicalUnitNormal is true if we can choose a canonical"
         "representative for each class of associate elements, that is"
         "associates?(a,b) returns true if and only if"
         "unitCanonical(a) = unitCanonical(b)."))
     (make-signature '|canonicalsClosed| () ()
       '("canonicalsClosed is true if"
         "unitCanonical(a)*unitCanonical(b) = unitCanonical(a*b)."))
     (make-signature '|arbitraryPrecision| () ()
       '("arbitraryPrecision means the user can set the"
         "precision for subsequent calculations."))
     (make-signature '|partiallyOrderedSet| () ()
       '("partiallyOrderedSet is true if"
         "a set with < which is transitive,"
         "but not(a < b or a = b)"
         "does not necessarily imply b<a."))
     (make-signature '|central| () ()
       '("central is true if, given an algebra over a ring R,"
         "the image of R is the center of the algebra, For example,"
         "the set of members of the algebra which commute with all"
```

```
        "others is precisely the image of R in the algebra."))
    (make-signature '|noetherian| () ()
      '("noetherian is true if all of its ideals are"
        "finitely generated."))
    (make-signature '|additiveValuation| () ()
      '("additiveValuation implies"
        "euclideanSize(a*b)=euclideanSize(a)+euclideanSize(b)."))
    (make-signature '|multiplicativeValuation| () ()
      '("multiplicativeValuation implies"
        "euclideanSize(a*b)=euclideanSize(a)*euclideanSize(b)."))
    (make-signature '|NullSquare| () ()
      '("NullSquare means that [x,x] = 0 holds."
        "See LieAlgebra."))
    (make-signature '|JacobiIdentity| () ()
      '("JacobiIdentity means that"
        "[x,[y,z]]+[y,[z,x]]+[z,[x,y]] = 0 holds."
        "See LieAlgebra."))
    (make-signature '|canonical| () ()
      '("canonical is true if and only if distinct elements have"
        "distinct data structures. For example, a domain of mathematical"
        "objects  which has the canonical attribute means that two"
        "objects are mathematically equal if and only if their data"
        "structures are equal."))
    (make-signature '|approximate| () ()
      '("approximate means 'is an approximation to the"
        "real numbers'."))
    ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AttributeRegistry|
  (progn
    (push '|AttributeRegistry| *Categories*)
    (make-instance '|AttributeRegistryType|)))
```

———–

1.34.8 BasicType

— defclass BasicTypeType —

```
(defclass |BasicTypeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BasicType")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'BASTYPE)
   (comment :initform (list
     "BasicType is the basic category for describing a collection"
     "of elements with = (equality)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
    (make-signature '|=| '(|Boolean|) '(% %)
      '("x=y tests if x and y are equal.") () t)
    (make-signature '|~=| '(|Boolean|) '(% %)
       '("x~=y tests if x and y are not equal.") () t)
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicType|
  (progn
    (push '|BasicType| *Categories*)
    (make-instance '|BasicTypeType|)))
```

———–

```
(
 ?~=? : (%,%) -> Boolean
 (~= (((|Boolean|) $ $) 7))

 ?=? : (%,%) -> Boolean
 (= (((|Boolean|) $ $) 6)))
```

BASTYPE Source Code

```
)abbrev category BASTYPE BasicType
--% BasicType
++ Description:
++ BasicType is the basic category for describing a collection
++ of elements with = (equality).

BasicType() : Category == SIG where

  SIG ==> with

    "=" : (%,%) -> Boolean
      ++ x=y tests if x and y are equal.

    "~=" : (%,%) -> Boolean
      ++ x~=y tests if x and y are not equal.

   add

     _~_=(x:%,y:%) : Boolean == not(x=y)
```

BASTYPE Lisp Code (other)

```
BasicType : Category == Type with
  =: (%,%) -> Boolean
  ~=: (%,%) -> Boolean

(def BasicType;AL nil)
(def BasicType;
  (lambda ()
    (bind ((g (Join (Type)
               (mkCategory domain
         '(((= ((Boolean) $ $)) true)
   ((~= ((Boolean) $ $)) true)
   ((before? ((Boolean $ $)) true))
```

```
 nil '((Boolean)) nil))))
          (store (tref g 0) '(BasicType))
  g))))

(def BasicType
  (lambda ()
    (when ((not (eq BasicType;AL nil)) BasicType;AL)
          (t (store BasicType;AL (BasicType;))))))
```

BASTYPE.lsp

```
(/VERSIONCHECK 2)

(SETQ |BasicType;AL| (QUOTE NIL))

(DEFUN |BasicType| NIL
 (LET (#:G1588)
   (COND
     (|BasicType;AL|)
     (T (SETQ |BasicType;AL| (|BasicType;|))))))

(DEFUN |BasicType;| NIL
 (PROG (#0=#1=#:G1586)
   (RETURN
     (PROG1
       (LETT #0#
         (|Join|
           (|mkCategory|
             (QUOTE |domain|)
             (QUOTE (((= ((|Boolean|) $ $)) T)
                     ((~= ((|Boolean|) $ $)) T)))
             NIL
             (QUOTE ((|Boolean|))) NIL))
         |BasicType|)
       (SETELT #0# 0 (QUOTE (|BasicType|)))))))

(SETF (GET (QUOTE |BasicType|) (QUOTE NILADIC)) T)
```

BASTYPE index.kaf

```
1057
(SETQ |$CategoryFrame|
  (|put|
    (QUOTE |BasicType|)
    (QUOTE |isCategory|)
    T
    (|addModemap|
      (QUOTE |BasicType|)
      (QUOTE (|BasicType|))
      (QUOTE ((|Category|)))
      T
      (QUOTE |BasicType|)
      |$CategoryFrame|)))
(SETF (GET (QUOTE |BasicType|) (QUOTE NILADIC)) T)
(|BasicType|)
```

```
|category|
(((|BasicType|) (|Category|)) (T |BasicType|))
(|Join| (CATEGORY |domain| (SIGNATURE = ((|Boolean|) $ $)) (SIGNATURE ~= ((|Boolean|) $ $))))
"/research/20190531/mnt/ubuntu/../../src/algebra/BASTYPE.spad"
((~= (*1 *2 *1 *1) (AND (|ofCategory| *1 (|BasicType|)) (|isDomain| *2 (|Boolean|))))
  (= (*1 *2 *1 *1) (AND (|ofCategory| *1 (|BasicType|)) (|isDomain| *2 (|Boolean|)))))
((~= (((|Boolean|) $ $) 7)) (= (((|Boolean|) $ $) 6)))
((|BASTYPE-;~=;2SB;1| ((|Boolean|) S S)))
BASTYPE
((|constructor|
 (NIL "BasicType is the basic category for describing a collection of elements with = (equality)."))
 (~= (((|Boolean|) $ $) "\\spad{x~=y} tests if \\spad{x} and \\spad{y} are not equal."))
 (= (((|Boolean|) $ $) "\\spad{x=y} tests if \\spad{x} and \\spad{y} are equal.")))
(("slot1Info" 0 NIL)
 ("documentation" 0 770)
 ("ancestors" 0 NIL)
 ("parents" 0 NIL)
 ("abbreviation" 0 762)
 ("predicates" 0 NIL)
 ("attributes" 0 NIL)
 ("signaturesAndLocals" 0 720)
 ("superDomain" 0 NIL)
 ("operationAlist" 0 665)
 ("modemaps" 0 494)
 ("sourceFile" 0 431)
 ("constructorCategory" 0 337)
 ("constructorModemap" 0 290)
 ("constructorKind" 0 279)
 ("constructorForm" 0 265)
 ("NILADIC" 0 214)
 ("compilerInfo" 0 20))
```

BASICTYPE database struct

```
)lisp (showdatabase '|BasicType|)
getdatabase call: BasicType            CONSTRUCTORKIND
CONSTRUCTORKIND: category
getdatabase call: BasicType            COSIG
COSIG: (NIL)
getdatabase call: BasicType            OPERATION
OPERATION: NIL
CONSTRUCTORMODEMAP:
getdatabase call: BasicType            CONSTRUCTORMODEMAP
getdatabase miss: BasicType            CONSTRUCTORMODEMAP

(((|BasicType|) (|Category|)) (T |BasicType|))
CONSTRUCTORCATEGORY:
getdatabase call: BasicType            CONSTRUCTORCATEGORY
getdatabase miss: BasicType            CONSTRUCTORCATEGORY

(|Join| (CATEGORY |domain| (SIGNATURE = ((|Boolean|) $ $))
            (SIGNATURE ~= ((|Boolean|) $ $))))
OPERATIONALIST:
getdatabase call: BasicType            OPERATIONALIST
getdatabase miss: BasicType            OPERATIONALIST

((~= (((|Boolean|) $ $) 7)) (= (((|Boolean|) $ $) 6)))
```

```
MODEMAPS:
getdatabase call: BasicType            MODEMAPS
getdatabase miss: BasicType            MODEMAPS

((~= (*1 *2 *1 *1)
     (AND (|ofCategory| *1 (|BasicType|)) (|isDomain| *2 (|Boolean|))))
 (= (*1 *2 *1 *1)
    (AND (|ofCategory| *1 (|BasicType|))
         (|isDomain| *2 (|Boolean|)))))
getdatabase call: BasicType            HASCATEGORY
HASCATEGORY: NIL
getdatabase call: BasicType            OBJECT
OBJECT: /mnt/c/Users/markb/EXE/axiom/mnt/ubuntu/algebra/BASTYPE.o
getdatabase call: BasicType            NILADIC
NILADIC: T
getdatabase call: BasicType            ABBREVIATION
ABBREVIATION: BASTYPE
getdatabase call: BasicType            CONSTRUCTOR?
CONSTRUCTOR?: T
getdatabase call: BasicType            CONSTRUCTOR
CONSTRUCTOR: NIL
getdatabase call: BasicType            DEFAULTDOMAIN
DEFAULTDOMAIN: NIL
getdatabase call: BasicType            ANCESTORS
ANCESTORS: NIL
getdatabase call: BasicType            SOURCEFILE
SOURCEFILE: bookvol10.2.pamphlet
getdatabase call: BasicType            CONSTRUCTORFORM
getdatabase miss: BasicType            CONSTRUCTORFORM
CONSTRUCTORFORM: (BasicType)
getdatabase call: BasicType            CONSTRUCTORARGS
getdatabase call: BasicType            CONSTRUCTORFORM
CONSTRUCTORARGS: NIL
getdatabase call: BasicType            ATTRIBUTES
getdatabase miss: BasicType            ATTRIBUTES
ATTRIBUTES: NIL
PREDICATES:
getdatabase call: BasicType            PREDICATES
getdatabase miss: BasicType            PREDICATES

NILgetdatabase call: BasicType            DOCUMENTATION
getdatabase miss: BasicType            DOCUMENTATION
DOCUMENTATION:
  ((constructor (NIL BasicType is the basic category for describing a collection of elements with = (equality).))
   (~= (((Boolean) $ $) \spad{x~=y} tests if \spad{x} and \spad{y} are not equal.))
   (= (((Boolean) $ $) \spad{x=y} tests if \spad{x} and \spad{y} are equal.)))
getdatabase call: BasicType            PARENTS
PARENTS: NIL
Value = NIL
```

BASTYPE-.lsp

```
(/VERSIONCHECK 2)

(DEFUN |BASTYPE-;~=;2SB;1| (|x| |y| $)
  (COND
    ((SPADCALL |x| |y| (QREFELT $ 8)) (QUOTE NIL))
    ((QUOTE T) (QUOTE T))))

(DEFUN |BasicType&| (|#1|)
  (PROG (DV$1 |dv$| $ |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devaluate| |#1|) . #0=(|BasicType&|))
        (LETT |dv$| (LIST (QUOTE |BasicType&|) DV$1) . #0#)
        (LETT $ (MAKE-ARRAY 10) . #0#)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3 (LETT |pv$| (|buildPredVector| 0 0 NIL) . #0#))
        (|stuffDomainSlots| $) (QSETREFV $ 6 |#1|)
        $))))

(SETF
  (GET (QUOTE |BasicType&|) (QUOTE |infovec|))
  (LIST
   (QUOTE #(NIL NIL NIL NIL NIL NIL (|local| |#1|) (|Boolean|) (0 . =) |BASTYPE-;~=;2SB;1|))
   (QUOTE #(~= 6))
   (QUOTE NIL)
   (CONS
     (|makeByteWordVec2| 1 (QUOTE NIL))
     (CONS
       (QUOTE #())
       (CONS (QUOTE #()) (|makeByteWordVec2| 9 (QUOTE (2 6 7 0 0 8 2 0 7 0 0 9))))))
   (QUOTE |lookupComplete|)))
```

BASTYPE- index.kaf

```
1540
(|updateSlot1DataBase| (QUOTE (|BasicType&| (NIL (~= ((7 0 0) 9))))))

(SETF
  (GET (QUOTE |BasicType&|) (QUOTE |infovec|))
  (LIST
    (QUOTE #(NIL NIL NIL NIL NIL NIL (|local| |#1|) (|Boolean|) (0 . =) |BASTYPE-;~=;2SB;1|))
    (QUOTE #(~= 6))
    (QUOTE NIL)
    (CONS
      (|makeByteWordVec2| 1 (QUOTE NIL))
      (CONS
        (QUOTE #())
        (CONS
          (QUOTE #())
          (|makeByteWordVec2| 9 (QUOTE (2 6 7 0 0 8 2 0 7 0 0 9))))))
    (QUOTE |lookupComplete|)))

(SETQ |$CategoryFrame|
  (|put|
    (QUOTE |BasicType&|)
    (QUOTE |isFunctor|)
    (QUOTE (((~= ((|Boolean|) $ $)) T (ELT $ 9))))
  (|addModemap|
    (QUOTE |BasicType&|)
    (QUOTE (|BasicType&| |#1|))
    (QUOTE ((CATEGORY |domain| (SIGNATURE ~= ((|Boolean|) |#1| |#1|))) (|BasicType|)))
    T
    (QUOTE |BasicType&|)
    (|put|
      (QUOTE |BasicType&|)
      (QUOTE |mode|)
      (QUOTE (|Mapping| (CATEGORY |domain| (SIGNATURE ~= ((|Boolean|) |#1| |#1|))) (|BasicType|)))
      |$CategoryFrame|))))
  (|BasicType&| S)
  |domain|
  (((|BasicType&| |#1|)
    (CATEGORY |domain| (SIGNATURE ~= ((|Boolean|) |#1| |#1|)))
    (|BasicType|))
   (T |BasicType&|))
  (CATEGORY |domain| (SIGNATURE ~= ((|Boolean|) |#1| |#1|)))
  "/research/20190531/mnt/ubuntu/../../src/algebra/BASTYPE.spad"
  ((~= (((|Boolean|) $ $) 9)))
  ((|BASTYPE-;~=;2SB;1| ((|Boolean|) S S)))
  BASTYPE-
  ((|constructor|
     (NIL "BasicType is the basic category for describing a collection of elements with = (equality)."))
   (~= (((|Boolean|) $ $) "\\spad{x~=y} tests if \\spad{x} and \\spad{y} are not equal."))
   (= (((|Boolean|) $ $) "\\spad{x=y} tests if \\spad{x} and \\spad{y} are equal.")))
  (|BasicType&| (NIL (~= ((7 0 0) 9))))
  (("slot1Info" 0 1502)
   ("documentation" 0 1215)
   ("ancestors" 0 NIL)
   ("parents" 0 NIL)
   ("abbreviation" 0 1206)
   ("predicates" 0 NIL)
   ("attributes" 0 NIL)
   ("signaturesAndLocals" 0 1164)
   ("superDomain" 0 NIL)
   ("operationAlist" 0 1135)
   ("modemaps" 0 NIL)
   ("sourceFile" 0 1072)
   ("constructorCategory" 0 1013)
   ("constructorModemap" 0 899)
   ("constructorKind" 0 890)
   ("constructorForm" 0 873)
   ("compilerInfo" 0 429)
   ("loadTimeStuff" 0 90)
   ("slot1DataBase" 0 20))
```

BASICTYPE database struct

```
)lisp (showdatabase '|BasicType&|)
getdatabase call: BasicType&          CONSTRUCTORKIND
CONSTRUCTORKIND: domain
getdatabase call: BasicType&          COSIG
COSIG: (NIL T)
getdatabase call: BasicType&          OPERATION
OPERATION: NIL
CONSTRUCTORMODEMAP:
getdatabase call: BasicType&          CONSTRUCTORMODEMAP
getdatabase miss: BasicType&          CONSTRUCTORMODEMAP

(((|BasicType&| |#1|)
  (CATEGORY |domain| (SIGNATURE ~= ((|Boolean|) |#1| |#1|)))
  (|BasicType|))
 (T |BasicType&|))
CONSTRUCTORCATEGORY:
getdatabase call: BasicType&           CONSTRUCTORCATEGORY
getdatabase miss: BasicType&           CONSTRUCTORCATEGORY

(CATEGORY |domain| (SIGNATURE ~= ((|Boolean|) |#1| |#1|)))
OPERATIONALIST:
getdatabase call: BasicType&           OPERATIONALIST
getdatabase miss: BasicType&           OPERATIONALIST

((~= (((|Boolean|) $ $) 9)))
MODEMAPS:
getdatabase call: BasicType&           MODEMAPS
getdatabase miss: BasicType&           MODEMAPS

NILgetdatabase call: BasicType&           HASCATEGORY
HASCATEGORY: NIL
getdatabase call: BasicType&           OBJECT
OBJECT: /mnt/c/Users/markb/EXE/axiom/mnt/ubuntu/algebra/BASTYPE-.o
getdatabase call: BasicType&           NILADIC
NILADIC: NIL
getdatabase call: BasicType&           ABBREVIATION
ABBREVIATION: BASTYPE-
getdatabase call: BasicType&           CONSTRUCTOR?
CONSTRUCTOR?: T
getdatabase call: BasicType&           CONSTRUCTOR
CONSTRUCTOR: NIL
getdatabase call: BasicType&           DEFAULTDOMAIN
DEFAULTDOMAIN: NIL
getdatabase call: BasicType&           ANCESTORS
ANCESTORS: NIL
getdatabase call: BasicType&           SOURCEFILE
SOURCEFILE: NIL
getdatabase call: BasicType&           CONSTRUCTORFORM
getdatabase miss: BasicType&           CONSTRUCTORFORM
CONSTRUCTORFORM: (BasicType& S)
getdatabase call: BasicType&           CONSTRUCTORARGS
getdatabase call: BasicType&           CONSTRUCTORFORM
CONSTRUCTORARGS: (S)
getdatabase call: BasicType&           ATTRIBUTES
getdatabase miss: BasicType&           ATTRIBUTES
ATTRIBUTES: NIL
PREDICATES:
getdatabase call: BasicType&           PREDICATES
getdatabase miss: BasicType&           PREDICATES

NILgetdatabase call: BasicType&           DOCUMENTATION
getdatabase miss: BasicType&           DOCUMENTATION
DOCUMENTATION: ((constructor (NIL BasicType is the basic category for describing a collection of elements with = (e
getdatabase call: BasicType&           PARENTS
PARENTS: NIL
Value = NIL
```

1.34.9 CanonicalAttribute

— defclass CanonicalAttributeType —

```
(defclass |CanonicalAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CanonicalAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATCANON)
   (comment :initform (list
     "The class of all domains which have canonical represenntation,"
     "that is, mathematically equal elements have the same data structure."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CanonicalAttribute|
  (progn
    (push '|CanonicalAttribute| *Categories*)
    (make-instance '|CanonicalAttributeType|)))
```

———-

1.34.10 CanonicalClosedAttribute

— defclass CanonicalClosedAttributeType —

```
(defclass |CanonicalClosedAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CanonicalClosedAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATCANCL)
   (comment :initform (list
     "The class of all integral domains such that"
     "unitCanonical(a)*unitCanonical(b) = unitCanonical(a*b)}"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CanonicalClosedAttribute|
  (progn
    (push '|CanonicalClosedAttribute| *Categories*)
    (make-instance '|CanonicalClosedAttributeType|)))
```

———-

1.34.11 CanonicalUnitNormalAttribute

— defclass CanonicalUnitNormalAttributeType —

```
(defclass |CanonicalUnitNormalAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CanonicalUnitNormalAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATCUNOR)
   (comment :initform (list
     "The class of all integral domains such that we can choose a canonical"
     "representative for each class of associate elements. That is,"
     "associates?(a,b) returns true if and only if"
     "unitCanonical(a) = unitCanonical(b)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CanonicalUnitNormalAttribute|
  (progn
    (push '|CanonicalUnitNormalAttribute| *Categories*)
    (make-instance '|CanonicalUnitNormalAttributeType|)))
```

——————-

1.34.12 CentralAttribute

— defclass CentralAttributeType —

```
(defclass |CentralAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CentralAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATCENRL)
   (comment :initform (list
     "Central is true if, given an algebra over a ring R, the image of R"
     "is the center of the algebra. For example, the set of members of the"
     "algebra which commute with all others is precisely the image of R"
     "in the algebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CentralAttribute|
  (progn
    (push '|CentralAttribute| *Categories*)
    (make-instance '|CentralAttributeType|)))
```

———-

1.34.13 CoercibleTo

— defclass CoercibleToType —

```
(defclass |CoercibleToType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CoercibleTo")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'KOERCE)
   (comment :initform (list
     "A is coercible to B means any element of A can automatically be"
     "converted into an element of B by the interpreter."))
   (argslist :initform (list (make-instance 'FSM-VAR :variable 'S :paramtype '|Type|)))
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| 'S '(%)
       '("coerce(a) transforms a into an element of S."))))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CoercibleTo|
  (progn
    (push '|CoercibleTo| *Categories*)
    (make-instance '|CoercibleToType|)))
```

———-

1.34.14 CombinatorialFunctionCategory

— defclass CombinatorialFunctionCategoryType —

```
(defclass |CombinatorialFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CombinatorialFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'CFCAT)
   (comment :initform (list
     "Category for the usual combinatorial functions;"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|binomial| '% '(% %)
       '("binomial(n,r) returns the \spad{(n,r)} binomial coefficient"
         "(often denoted in the literature by \spad{C(n,r)})."
         "Note that \spad{C(n,r) = n!/(r!(n-r)!)} where \spad{n >= r >= 0}.")
       '("[binomial(5,i) for i in 0..5]"))
     (make-signature '|factorial| '% '(%)
       '("factorial(n) computes the factorial of n"
         "(denoted in the literature by \spad{n!})"
```

```
        "Note that \spad{n! = n (n-1)! when n > 0}; also, \spad{0! = 1}."))
     (make-signature '|permutation| '% '(% %)
       '("permutation(n, m) returns the number of"
        "permutations of n objects taken m at a time."
        "Note that \spad{permutation(n,m) = n!/(n-m)!}."))))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CombinatorialFunctionCategory|
  (progn
    (push '|CombinatorialFunctionCategory| *Categories*)
    (make-instance '|CombinatorialFunctionCategoryType|)))
```

———-

1.34.15 CommutativeStarAttribute

— defclass CommutativeStarAttributeType —

```
(defclass |CommutativeStarAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CommutativeStarAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATCS)
   (comment :initform (list
     "The class of all commutative semigroups in multiplicative notation."
     "In other words domain D with '*': (D,D) -> D} which is"
     "commutative. Typically applied to rings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommutativeStarAttribute|
  (progn
    (push '|CommutativeStarAttribute| *Categories*)
    (make-instance '|CommutativeStarAttributeType|)))
```

———-

1.34.16 ConvertibleTo

— defclass ConvertibleToType —

```
(defclass |ConvertibleToType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ConvertibleTo")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'KONVERT)
   (comment :initform (list
     "A is convertible to B means any element of A"
     "can be converted into an element of B,"
     "but not automatically by the interpreter."))
   (argslist :initform (list (make-instance 'FSM-VAR :variable 'S :paramtype '|Type|)))
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|convert| 'S '(%)
       '("convert(a) transforms a into an element of S."))))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ConvertibleTo|
  (progn
    (push '|ConvertibleTo| *Categories*)
    (make-instance '|ConvertibleToType|)))
```

———-

1.34.17 ElementaryFunctionCategory

— defclass ElementaryFunctionCategoryType —

```
(defclass |ElementaryFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ELEMFUN)
   (comment :initform (list
     "Category for the elementary functions;"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|log| '% '(%)
       '("log(x) returns the natural logarithm of x. When evaluated"
        "into some subset of the complex numbers, the branch cut lies"
        "along the negative real axis, continuous with quadrant II. The"
        "domain does not contain the origin."))
     (make-signature '|exp| '% '(%)
       '("exp(x) returns %e to the power x."))
     (make-signature '|**| '% '(% %)
       '("x**y returns x to the power y.") () t)))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionCategory|
  (progn
    (push '|ElementaryFunctionCategory| *Categories*)
    (make-instance '|ElementaryFunctionCategoryType|)))
```

———-

1.34.18 Eltable

— defclass EltableType —

```
(defclass |EltableType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Eltable")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ELTAB)
   (comment :initform (list
     "An eltable over domains D and I is a structure which can be viewed"
     "as a function from D to I. Examples of eltable structures range from"
     "data structures, For example, those of type List, to algebraic"
     "structures like Polynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|elt| '(|Index|) '(% S)
       '("elt(u,i) (also written: u . i) returns the element of u indexed by i."
         "Error: if i is not an index of u."))))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Eltable|
  (progn
    (push '|Eltable| *Categories*)
    (make-instance '|EltableType|)))
```

———-

1.34.19 FiniteAggregateAttribute

— defclass FiniteAggregateAttributeType —

```
(defclass |FiniteAggregateAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteAggregateAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATFINAG)
   (comment :initform (list
     "The class of all aggregates with a finite number of arguments"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteAggregateAttribute|
  (progn
    (push '|FiniteAggregateAttribute| *Categories*)
    (make-instance '|FiniteAggregateAttributeType|)))
```

———-

1.34.20 HyperbolicFunctionCategory

— defclass HyperbolicFunctionCategoryType —

```
(defclass |HyperbolicFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "HyperbolicFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'HYPCAT)
   (comment :initform (list
     "Category for the hyperbolic trigonometric functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|cosh| '(%) '(%)
       '("cosh(x) returns the hyperbolic cosine of x."))
     (make-signature '|coth| '(%) '(%)
       '("coth(x) returns the hyperbolic cotangent of x."))
     (make-signature '|csch| '(%) '(%)
       '("csch(x) returns the hyperbolic cosecant of x."))
     (make-signature '|sech| '(%) '(%)
       '("sech(x) returns the hyperbolic secant of x."))
     (make-signature '|sinh| '(%) '(%)
       '("sinh(x) returns the hyperbolic sine of x."))
     (make-signature '|tanh| '(%) '(%)
       '("tanh(x) returns the hyperbolic tangent of x."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HyperbolicFunctionCategory|
  (progn
    (push '|HyperbolicFunctionCategory| *Categories*)
    (make-instance '|HyperbolicFunctionCategoryType|)))
```

———-

1.34.21 InnerEvalable

— defclass InnerEvalableType —

```
(defclass |InnerEvalableType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerEvalable")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'IEVALAB)
   (comment :initform (list
     "This category provides eval operations."
     "A domain may belong to this category if it is possible to make"
     "'evaluation' substitutions.  The difference between this"
     "and Evalable is that the operations in this category"
     "specify the substitution as a pair of arguments rather than as"
     "an equation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|eval| '(%) '(% A B)
       '("eval(f, x, v) replaces x by v in f."))
     (make-signature '|eval| '(%) '(% (|List| A) (|List| B))
       '("eval(f, [x1,...,xn], [v1,...,vn]) replaces xi by vi in f."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerEvalable|
  (progn
    (push '|InnerEvalable| *Categories*)
    (make-instance '|InnerEvalableType|)))
```

———-

1.34.22 JacobiIdentityAttribute

— defclass JacobiIdentityAttributeType —

```
(defclass |JacobiIdentityAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "JacobiIdentityAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATJACID)
   (comment :initform (list
     "JacobiIdentity means that [x,[y,z]]+[y,[z,x]]+[z,[x,y]] = 0 holds."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |JacobiIdentityAttribute|
  (progn
    (push '|JacobiIdentityAttribute| *Categories*)
    (make-instance '|JacobiIdentityAttributeType|)))
```

———-

1.34.23 LazyRepresentationAttribute

— defclass LazyRepresentationAttributeType —

```
(defclass |LazyRepresentationAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LazyRepresentationAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATLR)
   (comment :initform (list
     "The class of all domains which have a lazy representation"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LazyRepresentationAttribute|
  (progn
    (push '|LazyRepresentationAttribute| *Categories*)
    (make-instance '|LazyRepresentationAttributeType|)))
```

———-

1.34.24 LeftUnitaryAttribute

— defclass LeftUnitaryAttributeType —

```
(defclass |LeftUnitaryAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LeftUnitaryAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATLUNIT)
   (comment :initform (list
     "LeftUnitary is true if 1 * x = x for all x."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeftUnitaryAttribute|
  (progn
    (push '|LeftUnitaryAttribute| *Categories*)
    (make-instance '|LeftUnitaryAttributeType|)))
```

———-

1.34.25 ModularAlgebraicGcdOperations

— defclass ModularAlgebraicGcdOperationsType —

```
(defclass |ModularAlgebraicGcdOperationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ModularAlgebraicGcdOperations")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'MAGCDOC)
   (comment :initform (list
     "This category specifies operations needed by"
     "ModularAlgebraicGcd package.  Since we have multiple"
     "implementations we specify interface here and put"
     "implementations in separate packages.  Most operations"
     "are done using special purpose abstract representation."
     "Apropriate types are passesd as parametes: MPT is type"
     "of modular polynomials in one variable with coefficients"
     "in some algebraic extension.  MD is type of modulus."
     "Final results are converted to packed representation,"
     "with coefficients (from prime field) stored in one"
     "array and exponents (in main variable and in auxilary"
     "variables representing generators of algebrac extension)"
     "stored in parallel array."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|pseudoRem| 'MPT '(MPT MPT MD)
       '("pseudoRem(x, y, m) computes pseudoremainder of x by y"
         "modulo m."))
     (make-signature '|canonicalIfCan| '(|Union| MPT "failed") '(MPT MD)
       '("canonicalIfCan(x, m) tries to divide x by its leading"
         "coefficient modulo m."))
     (make-signature '|packModulus| '(|Union| MD "failed")
                                  '((|List| MP) (|List| |Symbol|) |Integer|)
       '("packModulus(lp, ls, p) converts lp, ls and prime p which"
         "together describe algebraic extension to packed"
         "representation."))
     (make-signature '|MPtoMPT| '(MPT) '(MP |Symbol| (|List| |Symbol|) MD)
       '("MPtoMPT(p, s, ls, m) coverts p to packed represntation."))
     (make-signature '|zero?| '(|Boolean|) '(MPT)
       '("zero?(x) checks if x is zero."))
     (make-signature '|degree| '(|Integer|) '(MPT)
       '("degree(x) gives degree of x."))
     (make-signature '|packExps| '(|SortedExponentVector|) '(|Integer| |Integer| MD)
       '("packExps(d, s, m) produces vector of exponents up"
         "to degree d.  s is size (degree) of algebraic extension."
         "Use together with repack1."))
     (make-signature '|repacke1| '(|Void|) '(MPT PA |Integer| MD)
       '("repack1(x, a, d, m) stores coefficients of x in a."
         "d is degree of x.  Corresponding exponents are given"
         "by packExps."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularAlgebraicGcdOperations|
  (progn
    (push '|ModularAlgebraicGcdOperations| *Categories*)
    (make-instance '|ModularAlgebraicGcdOperationsType|)))
```

———–

1.34.26 MultiplicativeValuationAttribute

— defclass MultiplicativeValuationAttributeType —

```
(defclass |MultiplicativeValuationAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MultiplicativeValuationAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATMULVA)
   (comment :initform (list
     "The class of all euclidean domains such that"
     "euclideanSize(a*b)=euclideanSize(a)*euclideanSize(b)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultiplicativeValuationAttribute|
  (progn
    (push '|MultiplicativeValuationAttribute| *Categories*)
    (make-instance '|MultiplicativeValuationAttributeType|)))
```

————-

1.34.27 NoZeroDivisorsAttribute

— defclass NoZeroDivisorsAttributeType —

```
(defclass |NoZeroDivisorsAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NoZeroDivisorsAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATNZDIV)
   (comment :initform (list
     "The class of all semirings such that x * y ~= 0 implies"
     "both x and y are non-zero."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NoZeroDivisorsAttribute|
  (progn
    (push '|NoZeroDivisorsAttribute| *Categories*)
    (make-instance '|NoZeroDivisorsAttributeType|)))
```

————-

1.34.28 NotherianAttribute

— defclass NotherianAttributeType —

```
(defclass |NotherianAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NotherianAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATNOTHR)
   (comment :initform (list
     "Notherian is true if all of its ideals are finitely generated."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NotherianAttribute|
  (progn
    (push '|NotherianAttribute| *Categories*)
    (make-instance '|NotherianAttributeType|)))
```

———-

1.34.29 NullSquareAttribute

— defclass NullSquareAttributeType —

```
(defclass |NullSquareAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NullSquareAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATNULSQ)
   (comment :initform (list
     "NullSquare means that [x,x] = 0 holds. See LieAlgebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NullSquareAttribute|
  (progn
    (push '|NullSquareAttribute| *Categories*)
    (make-instance '|NullSquareAttributeType|)))
```

———-

1.34.30 OpenMath

— defclass OpenMathType —

```
(defclass |OpenMathType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OpenMath")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'OM)
   (comment :initform (list
     "OpenMath provides operations for exporting an object"
     "in OpenMath format."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|OMwrite| '(|String|) '(%)
       '("OMwrite(u) returns the OpenMath XML encoding of u as a"
         "complete OpenMath object."))
     (make-signature '|OMwrite| '(|String|) '(% |Boolean|)
       '("OMwrite(u, true) returns the OpenMath XML encoding of u"
         "as a complete OpenMath object; OMwrite(u, false) returns the"
         "OpenMath XML encoding of u as an OpenMath fragment."))
     (make-signature '|OMwrite| '(|Void|) '(|OpenMathDevice| %)
       '("OMwrite(dev, u) writes the OpenMath form of u to the"
         "OpenMath device dev as a complete OpenMath object."))
     (make-signature '|OMwrite| '(|Void|) '(|OpenMathDevice| % |Boolean|)
       '("OMwrite(dev, u, true) writes the OpenMath form of u to"
         "the OpenMath device dev as a complete OpenMath object;"
         "OMwrite(dev, u, false) writes the object as an OpenMath fragment."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMath|
  (progn
    (push '|OpenMath| *Categories*)
    (make-instance '|OpenMathType|)))
```

———-

1.34.31 PartialTranscendentalFunctions

— defclass PartialTranscendentalFunctionsType —

```
(defclass |PartialTranscendentalFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PartialTranscendentalFunctions")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'PTRANFN)
   (comment :initform (list
     "A package which provides partial transcendental"
     "functions, for example, functions which return an answer or 'failed'"
```

```
    "This is the description of any package which provides partial"
    "functions on a domain belonging to TranscendentalFunctionCategory."))
  (argslist :initform nil)
  (macros :initform nil)
  (withlist :initform (list
    (make-signature '|nthRootIfCan| '(|Union| K "failed") '(K NNI)
      '("nthRootIfCan(z,n) returns the nth root of z if possible,"
        "and \"failed\" otherwise."))
    (make-signature '|expIfCan| '(|Union| K "failed") '(K)
      '("expIfCan(z) returns exp(z) if possible, and \"failed\" otherwise."))
    (make-signature '|logIfCan| '(|Union| K "failed") '(K)
      '("logIfCan(z) returns log(z) if possible, and \"failed\" otherwise."))
    (make-signature '|sinIfCan| '(|Union| K "failed") '(K)
      '("sinIfCan(z) returns sin(z) if possible, and \"failed\" otherwise."))
    (make-signature '|cosIfCan| '(|Union| K "failed") '(K)
      '("cosIfCan(z) returns cos(z) if possible, and \"failed\" otherwise."))
    (make-signature '|tanIfCan| '(|Union| K "failed") '(K)
      '("tanIfCan(z) returns tan(z) if possible, and \"failed\" otherwise."))
    (make-signature '|cotIfCan| '(|Union| K "failed") '(K)
      '("cotIfCan(z) returns cot(z) if possible, and \"failed\" otherwise."))
    (make-signature '|secIfCan| '(|Union| K "failed") '(K)
      '("secIfCan(z) returns sec(z) if possible, and \"failed\" otherwise."))
    (make-signature '|cscIfCan| '(|Union| K "failed") '(K)
      '("cscIfCan(z) returns csc(z) if possible, and \"failed\" otherwise."))
    (make-signature '|asinIfCan| '(|Union| K "failed") '(K)
      '("asinIfCan(z) returns asin(z) if possible, and \"failed\" otherwise."))
    (make-signature '|acosIfCan| '(|Union| K "failed") '(K)
      '("acosIfCan(z) returns acos(z) if possible, and \"failed\" otherwise."))
    (make-signature '|atanIfCan| '(|Union| K "failed") '(K)
      '("atanIfCan(z) returns atan(z) if possible, and \"failed\" otherwise."))
    (make-signature '|acotIfCan| '(|Union| K "failed") '(K)
      '("acotIfCan(z) returns acot(z) if possible, and \"failed\" otherwise."))
    (make-signature '|asecIfCan| '(|Union| K "failed") '(K)
      '("asecIfCan(z) returns asec(z) if possible, and \"failed\" otherwise."))
    (make-signature '|acscIfCan| '(|Union| K "failed") '(K)
      '("acscIfCan(z) returns acsc(z) if possible, and \"failed\" otherwise."))
    (make-signature '|sinhIfCan| '(|Union| K "failed") '(K)
      '("sinhIfCan(z) returns sinh(z) if possible, and \"failed\" otherwise."))
    (make-signature '|coshIfCan| '(|Union| K "failed") '(K)
      '("coshIfCan(z) returns cosh(z) if possible, and \"failed\" otherwise."))
    (make-signature '|tanhIfCan| '(|Union| K "failed") '(K)
      '("tanhIfCan(z) returns tanh(z) if possible, and \"failed\" otherwise."))
    (make-signature '|cothIfCan| '(|Union| K "failed") '(K)
      '("cothIfCan(z) returns coth(z) if possible, and \"failed\" otherwise."))
    (make-signature '|sechIfCan| '(|Union| K "failed") '(K)
      '("sechIfCan(z) returns sech(z) if possible, and \"failed\" otherwise."))
    (make-signature '|cschIfCan| '(|Union| K "failed") '(K)
      '("cschIfCan(z) returns csch(z) if possible, and \"failed\" otherwise."))
    (make-signature '|asinhIfCan| '(|Union| K "failed") '(K)
      '("asinhIfCan(z) returns asinh(z) if possible, and \"failed\" otherwise."))
    (make-signature '|acoshIfCan| '(|Union| K "failed") '(K)
      '("acoshIfCan(z) returns acosh(z) if possible, and \"failed\" otherwise."))
    (make-signature '|atanhIfCan| '(|Union| K "failed") '(K)
      '("atanhIfCan(z) returns atanh(z) if possible, and \"failed\" otherwise."))
    (make-signature '|acothIfCan| '(|Union| K "failed") '(K)
      '("acothIfCan(z) returns acoth(z) if possible, and \"failed\" otherwise."))
    (make-signature '|asechIfCan| '(|Union| K "failed") '(K)
      '("asechIfCan(z) returns asech(z) if possible, and \"failed\" otherwise."))
    (make-signature '|acschIfCan| '(|Union| K "failed") '(K)
      '("acschIfCan(z) returns acsch(z) if possible, and \"failed\" otherwise."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialTranscendentalFunctions|
  (progn
    (push '|PartialTranscendentalFunctions| *Categories*)
    (make-instance '|PartialTranscendentalFunctionsType|)))
```

———-

1.34.32 PartiallyOrderedSetAttribute

— defclass PartiallyOrderedSetAttributeType —

```
(defclass |PartiallyOrderedSetAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PartiallyOrderedSetAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATPOSET)
   (comment :initform (list
     "PartiallyOrderedSet is true if a set with < is transitive,"
     "but not(a <b or a = b). It does not imply b < a"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartiallyOrderedSetAttribute|
  (progn
    (push '|PartiallyOrderedSetAttribute| *Categories*)
    (make-instance '|PartiallyOrderedSetAttributeType|)))
```

———-

1.34.33 PrimitiveFunctionCategory

— defclass PrimitiveFunctionCategoryType —

```
(defclass |PrimitiveFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PrimitiveFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'PRIMCAT)
   (comment :initform (list
     "Category for the functions defined by integrals"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|integral| '(%) '(% |Symbol|)
       '("integral(f, x) returns the formal integral of f dx."))
     (make-signature '|integral| '(%) '(% (|SegmentBinding| %))
       '("integral(f, x = a..b) returns the formal definite integral"
         "of f dx for x between a and b."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveFunctionCategory|
  (progn
    (push '|PrimitiveFunctionCategory| *Categories*)
    (make-instance '|PrimitiveFunctionCategoryType|)))
```

———-

1.34.34 RadicalCategory

— defclass RadicalCategoryType —

```
(defclass |RadicalCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RadicalCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'RADCAT)
   (comment :initform (list
     "The RadicalCategory is a model for the rational numbers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|sqrt| '(%) '(%)
       '("sqrt(x) returns the square root of x. The branch cut lies along"
         "the negative real axis, continuous with quadrant II."))
     (make-signature '|nthRoot| '(%) '(% |Integer|)
       '("nthRoot(x,n) returns the nth root of x."))
     (make-signature '|**| '(%) '(% (|Fraction| |Integer|))
       '("x ** y is the rational exponentiation of x by the power y.") () t)
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadicalCategory|
  (progn
    (push '|RadicalCategory| *Categories*)
    (make-instance '|RadicalCategoryType|)))
```

———-

1.34.35 RetractableTo

— defclass RetractableToType —

```
(defclass |RetractableToType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RetractableTo")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'RETRACT)
   (comment :initform (list
     "A is retractable to B means that some elements if A can be converted"
     "into elements of B and any element of B can be converted into an"
     "element of A."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(%) '(S)
       '("coerce(a) transforms a into an element of %."))
     (make-signature '|retractIfCan| '(|Union| S "failed") '(%)
       '("retractIfCan(a) transforms a into an element of S if possible."
         "Returns \"failed\" if a cannot be made into an element of S."))
     (make-signature '|retract| '(S) '(%)
       '("retract(a) transforms a into an element of S if possible."
         "Error: if a cannot be made into an element of S."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RetractableTo|
  (progn
    (push '|RetractableTo| *Categories*)
    (make-instance '|RetractableToType|)))
```

————-

1.34.36 RightUnitaryAttribute

— defclass RightUnitaryAttributeType —

```
(defclass |RightUnitaryAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RightUnitaryAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATRUNIT)
   (comment :initform (list
     "RightUnitary is true if x * 1 = x for all x."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RightUnitaryAttribute|
  (progn
    (push '|RightUnitaryAttribute| *Categories*)
    (make-instance '|RightUnitaryAttributeType|)))
```

———-

1.34.37 ShallowlyMutableAttribute

— defclass ShallowlyMutableAttributeType —

```
(defclass |ShallowlyMutableAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ShallowlyMutableAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATSHMUT)
   (comment :initform (list
     "The class of all domains which have immediate components that"
     "are updateable in place (mutable). The properties of any component"
     "domain are irrevelant to the ShallowlyMutableAttribute."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ShallowlyMutableAttribute|
  (progn
    (push '|ShallowlyMutableAttribute| *Categories*)
    (make-instance '|ShallowlyMutableAttributeType|)))
```

———-

1.34.38 SpecialFunctionCategory

— defclass SpecialFunctionCategoryType —

```
(defclass |SpecialFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SpecialFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'SPFCAT)
   (comment :initform (list
     "Category for the other special functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|abs| '(%) '(%)
       '("abs(x) returns the absolute value of x."))
     (make-signature '|Gamma| '(%) '(%)
       '("Gamma(x) is the Euler Gamma function."))
     (make-signature '|Beta| '(%) '(% %)
       '("Beta(x,y) is Gamma(x) * Gamma(y)/Gamma(x+y)."))
     (make-signature '|digamma| '(%) '(%)
       '("digamma(x) is the logarithmic derivative of Gamma(x)"
         "(often written psi(x) in the literature)."))
```

```
    (make-signature '|polygamma| '(%) '(% %)
      '("polygamma(k,x) is the \spad{k-th} derivative of digamma(x),"
        "(often written psi(k,x) in the literature)."))
    (make-signature '|Gamma| '(%) '(% %)
      '("Gamma(a,x) is the incomplete Gamma function."))
    (make-signature '|besselJ| '(%) '(% %)
      '("besselJ(v,z) is the Bessel function of the first kind."))
    (make-signature '|besselY| '(%) '(% %)
      '("besselY(v,z) is the Bessel function of the second kind."))
    (make-signature '|besselI| '(%) '(% %)
      '("besselI(v,z) is the modified Bessel function of the first kind."))
    (make-signature '|besselK| '(%) '(% %)
      '("besselK(v,z) is the modified Bessel function of the second kind."))
    (make-signature '|airyAi| '(%) '(%)
      '("airyAi(x) is the Airy function Ai(x)."))
    (make-signature '|airyBi| '(%) '(%)
      '("airyBi(x) is the Airy function Bi(x)."))
  ))
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |SpecialFunctionCategory|
  (progn
    (push '|SpecialFunctionCategory| *Categories*)
    (make-instance '|SpecialFunctionCategoryType|)))
```

————-

1.34.39 TrigonometricFunctionCategory

— defclass TrigonometricFunctionCategoryType —

```
(defclass |TrigonometricFunctionCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TrigonometricFunctionCategory")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'TRIGCAT)
   (comment :initform (list
     "Category for the trigonometric functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|cos| '(%) '(%)
       '("cos(x) returns the cosine of x."))
     (make-signature '|cot| '(%) '(%)
       '("cot(x) returns the cotangent of x."))
     (make-signature '|csc| '(%) '(%)
       '("csc(x) returns the cosecant of x."))
     (make-signature '|sec| '(%) '(%)
       '("sec(x) returns the secant of x."))
     (make-signature '|sin| '(%) '(%)
       '("sin(x) returns the sine of x."))
     (make-signature '|tan| '(%) '(%)
       '("tan(x) returns the tangent of x."))
  ))
```

```
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TrigonometricFunctionCategory|
  (progn
    (push '|TrigonometricFunctionCategory| *Categories*)
    (make-instance '|TrigonometricFunctionCategoryType|)))
```

———-

1.34.40 Type

— defclass TypeType —

```
(defclass |TypeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Type")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'TYPE)
   (comment :initform (list
     "The fundamental Type."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Type|
  (progn
    (push '|Type| *Categories*)
    (make-instance '|TypeType|)))
```

———-

1.34.41 UnitsKnownAttribute

— defclass UnitsKnownAttributeType —

```
(defclass |UnitsKnownAttributeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnitsKnownAttribute")
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'ATUNIKN)
   (comment :initform (list
     "The class of all monoids (multiplicative semigroups with a 1)"
     "such that the operation recop can only return 'failed'"
     "if its argument is not a unit."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnitsKnownAttribute|
  (progn
    (push '|UnitsKnownAttribute| *Categories*)
    (make-instance '|UnitsKnownAttributeType|)))
```

———–

1.35 Level 2

— defvar level2 —

```
(defvar level2
 '(|Aggregate| |CombinatorialOpsCategory| |EltableAggregate| |Evalable|
   |FortranProgramCategory| |FullyRetractableTo| |Logic| |Patternable|
   |PlottablePlaneCurveCategory| |PlottableSpaceCurveCategory| |RealConstant|
   |SegmentCategory| |SetCategory| |TranscendentalFunctionCategory|))
```

———–

1.35.1 Aggregate

— defclass AggregateType —

```
(defclass |AggregateType| (|TypeType|)
  ((parents :initform '(|Type|))
   (name :initform "Aggregate")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'AGG)
   (comment :initform (list
     "The notion of aggregate serves to model any data structure aggregate,"
     "designating any collection of objects, with heterogenous or homogeneous"
     "members, with a finite or infinite number of members, explicitly or"
     "implicitly represented. An aggregate can in principle represent"
     "everything from a string of characters to abstract sets such"
     "as 'the set of x satisfying relation r(x)'"
     "An attribute 'finiteAggregate' is used to assert that a domain"
     "has a finite number of elements."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|eq?| '(|Boolean|) '(% %)
       '("eq?(u,v) tests if u and v are same objects."))
     (make-signature '|copy| '(%) '(%)
       '("copy(u) returns a top-level (non-recursive) copy of u."
         "Note that for collections, copy(u) == [x for x in u]."))
     (make-signature '|empty| '(%) ()
       '("empty()$D creates an aggregate of type D with 0 elements."
         "Note that The $D can be dropped if understood by context,"
         "for example u: D := empty()."))
     (make-signature '|empty?| '(|Boolean|) '(%)
       '("empty?(u) tests if u has 0 elements."))
     (make-signature '|less?| '(|Boolean|) '(% |NonNegativeInteger|)
       '("less?(u,n) tests if u has less than n elements."))
     (make-signature '|more?| '(|Boolean|) '(% |NonNegativeInteger|)
       '("more?(u,n) tests if u has greater than n elements."))
     (make-signature '|size?| '(|Boolean|) '(% |NonNegativeInteger|)
       '("size?(u,n) tests if u has exactly n elements."))
     (make-signature '|sample| '(%) '(|constant|)
       '("sample yields a value of type %"))))
   (haslist :initform (list
     (make-haslist (list
       (make-hasclause '|%| '|finiteAggregate|
         (list
           (make-signature '|#| '(|NonNegativeInteger|) '(%)
             '("# u returns the number of items in u.") () t)))))))
   (addlist :initform nil)))

(defvar |Aggregate|
  (progn
    (push '|Aggregate| *Categories*)
    (make-instance '|AggregateType|)))
```

————-

1.35.2 CombinatorialOpsCategory

— defclass CombinatorialOpsCategoryType —

```
(defclass |CombinatorialOpsCategoryType| (|CombinatorialFunctionCategoryType|)
  ((parents :initform '(|CombinatorialFunctionCategory|))
   (name :initform "CombinatorialOpsCategory")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'COMBOPC)
   (comment :initform (list
     "CombinatorialOpsCategory is the category obtaining by adjoining"
     "summations and products to the usual combinatorial operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|factorials| '(%) '(%)
       '("factorials(f) rewrites the permutations and binomials in f"
         "in terms of factorials"))
     (make-signature '|factorials| '(%) '(% |Symbol|)
       '("factorials(f, x) rewrites the permutations and binomials in f"
         "involving x in terms of factorials"))
     (make-signature '|summation| '(%) '(% |Symbol|)
       '("summation(f(n), n) returns the formal sum S(n) which verifies"
         "S(n+1) - S(n) = f(n)"))
     (make-signature '|summation| '(%) '(% (|SegmentBinding| %))
       '("summation(f(n), n = a..b) returns f(a) + ... + f(b) as a"
         "formal sum"))
     (make-signature '|product| '(%) '(% |Symbol|)
       '("product(f(n), n) returns the formal product P(n) which verifies"
         "P(n+1)/P(n) = f(n)"))
     (make-signature '|product| '(%) '(% (|SegmentBinding| %))
       '("product(f(n), n = a..b) returns f(a) * ... * f(b) as a"
         "formal product"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CombinatorialOpsCategory|
  (progn
    (push '|CombinatorialOpsCategory| *Categories*)
    (make-instance '|CombinatorialOpsCategoryType|)))
```

———-

1.35.3 EltableAggregate

— defclass EltableAggregateType —

```
(defclass |EltableAggregateType| (|EltableType|)
  ((parents :initform '(|Eltable|))
   (name :initform "EltableAggregate")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'ELTAGG)
   (comment :initform (list
     "An eltable aggregate is one which can be viewed as a function."
     "For example, the list [1,7,4] can applied to 0,1, and 2 respectively"
     "will return the integers 1, 7, and 4; thus this list may be viewed as"
     "mapping 0 to 1, 1 to 7 and 2 to 4. In general, an aggregate"
     "can map members of a domain Dom to an image domain Im."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|elt| '(|Im|) '(% |Dom| |Im|)
       '("elt(u, x, y) applies u to x if x is in the domain of u,"
         "and returns y otherwise."
         "For example, if u is a polynomial in \axiom{x} over the rationals,"
         "elt(u,n,0) may define the coefficient of x"
         "to the power n, returning 0 when n is out of range."))
     (make-signature '|qelt| '(|Im|) '(% |Dom|)
       '("qelt(u, x) applies u to x without checking whether"
         "x is in the domain of u. If x is not"
         "in the domain of u a memory-access violation may occur."
         "If a check on whether x is in the domain of u"
         "is required, use the function elt."))
   ))
   (haslist :initform (list
     (make-haslist (list
       (make-hasclause '|%| '|shallowlyMutable|
         (list
           (make-signature '|setelt| '(|Im|) '(% |Dom| |Im|)
             '("setelt(u,x,y) sets the image of x to be y under u,"
               "assuming x is in the domain of u."
               "Error: if x is not in the domain of u."))
           (make-signature '|qsetelt_!| '(|Im|) '(% |Dom| |Im|)
             '("qsetelt!(u,x,y) sets the image of x to be y"
               "under u, without checking that x is in"
               "the domain of u."
               "If such a check is required use the function setelt."))))))))
   (addlist :initform nil)))

(defvar |EltableAggregate|
  (progn
    (push '|EltableAggregate| *Categories*)
    (make-instance '|EltableAggregateType|)))
```

———–

1.35.4 Evalable

— defclass EvalableType —

```
(defclass |EvalableType| (|InnerEvalableType|)
  ((parents :initform '(|InnerEvalable|))
   (name :initform "Evalable")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'EVALAB)
   (comment :initform (list
     "This category provides eval operations."
     "A domain may belong to this category if it is possible to make"
     "'evaluation' substitutions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|eval| '(%) '(% (|Equation| R))
       '("eval(f,x = v) replaces x by v in f."))
     (make-signature '|eval| '(%) '($ (List (Equation R)))
       '("eval(f, [x1 = v1,...,xn = vn]) replaces xi by vi in f."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Evalable|
  (progn
    (push '|Evalable| *Categories*)
    (make-instance '|EvalableType|)))
```

———–

1.35.5 FortranProgramCategory

— defclass FortranProgramCategoryType —

```
(defclass |FortranProgramCategoryType| (|TypeType| |CoercibleToType|)
  ((parents :initform '(|Type| |CoercibleTo|))
   (name :initform "FortranProgramCategory")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'FORTCAT)
   (comment :initform (list
     "FortranProgramCategory provides various models of FORTRAN subprograms."
     "These can be transformed into actual FORTRAN code."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|outputAsFortran| '(|Void|) '(%)
       '("outputAsFortran(u) translates u into a legal FORTRAN subprogram."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranProgramCategory|
  (progn
    (push '|FortranProgramCategory| *Categories*)
    (make-instance '|FortranProgramCategoryType|)))
```

———–

1.35.6 FullyRetractableTo

— defclass FullyRetractableToType —

```
(defclass |FullyRetractableToType| (|RetractableToType|)
  ((parents :initform '(|RetractableTo|))
   (name :initform "FullyRetractableTo")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'FRETRCT)
   (comment :initform (list
     "A is fully retractable to B means that A is retractable to B and"
     "if B is retractable to the integers or rational numbers then so is A."
     "In particular, what we are asserting is that there are no integers"
     "(rationals) in A which don't retract into B."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullyRetractableTo|
  (progn
    (push '|FullyRetractableTo| *Categories*)
    (make-instance '|FullyRetractableToType|)))
```

———–

1.35.7 Logic

— defclass LogicType —

```
(defclass |LogicType| (|BasicTypeType|)
  ((parents :initform '(|BasicType|))
   (name :initform "Logic")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'LOGIC)
   (comment :initform (list
     "Logic provides the basic operations for lattices,"
     "for example, boolean algebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|-| '(%) '(%)
       '("~(x) returns the logical complement of x.") () t)
     (make-signature '|/\\| '(%) '(% %)
       '("/\\ returns the logical `meet', for example, `and'.") () t)
     (make-signature '|\\/| '(%) '(% %)
       '("\\/ returns the logical `join', for example, `or'.") () t)
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Logic|
  (progn
    (push '|Logic| *Categories*)
    (make-instance '|LogicType|)))
```

———-

1.35.8 Patternable

— defclass PatternableType —

```
(defclass |PatternableType| (|ConvertibleToType|)
  ((parents :initform '(|ConvertibleTo|))
   (name :initform "Patternable")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'PATAB)
   (comment :initform (list
     "Category of sets that can be converted to useful patterns"
     "An object S is Patternable over an object R if S can"
     "lift the conversions from R into Pattern(Integer) and"
     "Pattern(Float) to itself"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Patternable|
  (progn
    (push '|Patternable| *Categories*)
    (make-instance '|PatternableType|)))
```

———-

1.35.9 PlottablePlaneCurveCategory

— defclass PlottablePlaneCurveCategoryType —

```
(defclass |PlottablePlaneCurveCategoryType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "PlottablePlaneCurveCategory")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'PPCURVE)
   (comment :initform (list
     "PlotablePlaneCurveCategory is the category of curves in the plane"
     "which may be plotted via the graphics facilities. Functions are"
     "provided for obtaining lists of lists of points, representing the"
     "branches of the curve, and for determining the ranges of the"
     "x-coordinates and y-coordinates of the points on the curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|listBranches| '(L L (POINT)) '(%)
       '("listBranches(c) returns a list of lists of points, representing the"
         "branches of the curve c."))
     (make-signature '|xRange| '((SEG SF)) '(%)
       '("xRange(c) returns the range of the x-coordinates of the points"
         "on the curve c."))
     (make-signature '|yRange| '((SEG SF)) '(%)
       '("yRange(c) returns the range of the y-coordinates of the points"
         "on the curve c."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlottablePlaneCurveCategory|
  (progn
    (push '|PlottablePlaneCurveCategory| *Categories*)
    (make-instance '|PlottablePlaneCurveCategoryType|)))
```

———-

1.35.10 PlottableSpaceCurveCategory

— defclass PlottableSpaceCurveCategoryType —

```
(defclass |PlottableSpaceCurveCategoryType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "PlottableSpaceCurveCategory")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'PSCURVE)
   (comment :initform (list
     "PlottableSpaceCurveCategory is the category of curves in"
     "3-space which may be plotted via the graphics facilities.  Functions are"
     "provided for obtaining lists of lists of points, representing the"
     "branches of the curve, and for determining the ranges of the"
     "x-, y-, and z-coordinates of the points on the curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|listBranches| '(L L POINT) '(%)
       '("listBranches(c) returns a list of lists of points, representing the"
         "branches of the curve c."))
     (make-signature '|xRange| '((SEG SF)) '(%)
       '("xRange(c) returns the range of the x-coordinates of the points"
         "on the curve c."))
     (make-signature '|yRange| '((SEG SF)) '(%)
       '("yRange(c) returns the range of the y-coordinates of the points"
         "on the curve c."))
     (make-signature '|zRange| '((SEG SF)) '(%)
       '("zRange(c) returns the range of the z-coordinates of the points"
         "on the curve c."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlottableSpaceCurveCategory|
  (progn
    (push '|PlottableSpaceCurveCategory| *Categories*)
    (make-instance '|PlottableSpaceCurveCategoryType|)))
```

————-

1.35.11 RealConstant

— defclass RealConstantType —

```
(defclass |RealConstantType| (|ConvertibleToType|)
  ((parents :initform '(|ConvertibleTo|))
   (name :initform "RealConstant")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'REAL)
   (comment :initform (list
     "The category of real numeric domains, that is, convertible to floats."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealConstant|
  (progn
    (push '|RealConstant| *Categories*)
    (make-instance '|RealConstantType|)))
```

———-

1.35.12 SegmentCategory

— defclass SegmentCategoryType —

```
(defclass |SegmentCategoryType| (|TypeType|)
  ((parents :initform '(|Type|))
   (name :initform "SegmentCategory")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'SEGCAT)
   (comment :initform (list
     "This category provides operations on ranges, or segments"
     "as they are called."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|SEGMENT| '(%) '(S S)
       '("l..h creates a segment with l and h as the endpoints."))
     (make-signature '|BY| '(%) '(% |Integer|)
       '("s by n creates a new segment in which only every"
         "n-th element is used."))
     (make-signature '|lo| '(S) '(%)
       '("lo(s) returns the first endpoint of s."
         "Note that lo(l..h) = l."))
     (make-signature '|hi| '(S) '(%)
       '("hi(s) returns the second endpoint of s."
         "Note that hi(l..h) = h."))
     (make-signature '|low| '(S) '(%)
       '("low(s) returns the first endpoint of s."
         "Note that low(l..h) = l."))
     (make-signature '|high| '(S) '(%)
       '("high(s) returns the second endpoint of s."
         "Note that high(l..h) = h."))
     (make-signature '|incr| '(|Integer|) '(%)
       '("incr(s) returns n, where s is a segment in which every"
         "n-th element is used."
         "Note that incr(l..h by n) = n."))
     (make-signature '|segment| '(%) '(S S)
       '("segment(i,j) is an alternate way to create the segment i..j."))
     (make-signature '|convert| '(%) '(S)
       '("convert(i) creates the segment i..i."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SegmentCategory|
  (progn
    (push '|SegmentCategory| *Categories*)
    (make-instance '|SegmentCategoryType|)))
```

———-

1.35.13 SetCategory

— defclass SetCategoryType —

```
(defclass |SetCategoryType| (|BasicTypeType| |CoercibleToType|)
  ((parents :initform '(|BasicType| |CoercibleTo|))
   (name :initform "SetCategory")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'SETCAT)
   (comment :initform (list
     "SetCategory is the basic category for describing a collection"
     "of elements with = (equality) and coerce to output form."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|hash| '(|SingleInteger|) '(%)
       '("hash(s) calculates a hash code for s."))
     (make-signature '|latex| '(|String|) '(%)
       '("latex(s) returns a LaTeX-printable output representation of s."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SetCategory|
  (progn
    (push '|SetCategory| *Categories*)
    (make-instance '|SetCategoryType|)))
```

———-

1.35.14 TranscendentalFunctionCategory

— defclass TranscendentalFunctionCategoryType —

```
(defclass |TranscendentalFunctionCategoryType| (|ArcHyperbolicFunctionCategoryType|
                                                |ArcTrigonometricFunctionCategoryType|
                                                |ElementaryFunctionCategoryType|
                                                |HyperbolicFunctionCategoryType|
                                                |TrigonometricFunctionCategoryType|)
  ((parents :initform '(|ArcHyperbolicFunctionCategory|
                        |ArcTrigonometricFunctionCategory|
                        |ElementaryFunctionCategory|
                        |HyperbolicFunctionCategory|
                        |TrigonometricFunctionCategory|))
   (name :initform "TranscendentalFunctionCategory")
   (marker :initform 'category)
   (level :initform 2)
   (abbreviation :initform 'TRANFUN)
   (comment :initform (list
     "Category for the transcendental elementary functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|pi| '(%) ()
       '("pi() returns the constant pi."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalFunctionCategory|
  (progn
    (push '|TranscendentalFunctionCategory| *Categories*)
    (make-instance '|TranscendentalFunctionCategoryType|)))
```

———-

1.36 Level 3

— defvar level3 —

```
(defvar level3
 '(|AbelianSemiGroup| |BlowUpMethodCategory| |Comparable| |FileCategory|
   |FileNameCategory| |Finite| |FortranFunctionCategory| |FortranMatrixCategory|
   |FortranMatrixFunctionCategory| |FortranVectorCategory|
   |FortranVectorFunctionCategory| |FullyEvalableOver| |GradedModule|
   |HomogeneousAggregate| |IndexedDirectProductCategory|
   |LiouvillianFunctionCategory| |Monad| |NumericalIntegrationCategory|
   |NumericalOptimizationCategory| |OrderedSet|
   |OrdinaryDifferentialEquationsSolverCategory|
   |PartialDifferentialEquationsSolverCategory| |PatternMatchable|
   |RealRootCharacterizationCategory| |SExpressionCategory|
   |SegmentExpansionCategory| |SemiGroup| |SetCategoryWithDegree| |StepThrough|
   |ThreeSpaceCategory|))
```

———-

1.36.1 AbelianSemiGroup

— defclass AbelianSemiGroupType —

```
(defclass |AbelianSemiGroupType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "AbelianSemiGroup")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'ABELSG)
   (comment :initform (list
     "The class of all additive (commutative) semigroups, that is,"
     "a set with a commutative and associative operation +."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|+| '(%) '(% %)
       '("x+y computes the sum of x and y.") () t)
     (make-signature '|*| '(%) '(|PositiveInteger| %)
       '("n*x computes the left-multiplication of x by the positive"
         "integer n. This is equivalent to adding x to itself n times.")
       () t)
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AbelianSemiGroup|
  (progn
    (push '|AbelianSemiGroup| *Categories*)
    (make-instance '|AbelianSemiGroupType|)))
```

———-

1.36.2 BlowUpMethodCategory

— defclass BlowUpMethodCategoryType —

```
(defclass |BlowUpMethodCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "BlowUpMethodCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'BLMETCT)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(%) '((|List| |Integer|)) ())
     (make-signature '|excepCoord| '(|Integer|) '(%) ())
     (make-signature '|chartCoord| '(|Integer|) '(%) ())
     (make-signature '|transCoord| '(|Integer|) '(%) ())
     (make-signature '|createHN| '(%) '(|Integer| |Integer| |Integer|
                                        |Integer| |Integer| |Boolean|
       (|Union| "left" "center" "right" "vertical" "horizontal")) ())
     (make-signature '|ramifMult| '(|Integer|) '(%) ())
     (make-signature '|infClsPt?| '(|Boolean|) '(%) ())
     (make-signature '|quotValuation| '(|Integer|) '(%) ())
     (make-signature '|type| '(%) '((|Union| "left" "center" "right"
                                             "vertical" "horizontal")) ())
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BlowUpMethodCategory|
  (progn
    (push '|BlowUpMethodCategory| *Categories*)
    (make-instance '|BlowUpMethodCategoryType|)))
```

———-

1.36.3 Comparable

— defclass ComparableType —

```
(defclass |ComparableType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Comparable")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'COMPAR)
   (comment :initform (list
     "The class of set equipped with possibly unnatural linear order"
     "(needed for technical reasons)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|smaller?| '(|Boolean|) '(% %)
       '("smaller?(x,y) is a strict local total ordering on the elements of the set"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Comparable|
  (progn
    (push '|Comparable| *Categories*)
    (make-instance '|ComparableType|)))
```

———-

1.36.4 FileCategory

— defclass FileCategoryType —

```
(defclass |FileCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "FileCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FILECAT)
   (comment :initform (list
     "This category provides an interface to operate on files in the"
     "computer's file system.  The precise method of naming files"
     "is determined by the Name parameter.  The type of the contents"
     "of the file is determined by S."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|open| '(%) '(|Name|)
       '("open(s) returns the file s open for input."))
     (make-signature '|open| '(%) '(|Name| |IOMode|)
       '("open(s,mode) returns a file s open for operation in the"
         "indicated mode: \"input\" or \"output\"."))
     (make-signature '|reopen!| '(%) '(% |IOMode|)
       '("reopen!(f,mode) returns a file f reopened for operation in the"
         "indicated mode: \"input\" or \"output\"."
         "reopen!(f,\"input\") will reopen the file f for input."))
     (make-signature '|close!| '(%) '(%)
       '("close!(f) returns the file f closed to input and output."))
     (make-signature '|name| '(|Name|) '(%)
       '("name(f) returns the external name of the file f."))
     (make-signature '|iomode| '(|IOMode|) '(%)
       '("iomode(f) returns the status of the file f. The input/output"
         "status of f may be \"input\", \"output\" or \"closed\" mode."))
     (make-signature '|read!| '(S) '(%)
       '("read!(f) extracts a value from file f. The state of f is"
         "modified so a subsequent call to read! will return"
         "the next element."))
     (make-signature '|write!| '(S) '(% S)
       '("write!(f,s) puts the value s into the file f."
         "The state of f is modified so subsequents call to write!"
         "will append one after another."))
     (make-signature '|flush| '(|Void|) '(%)
       '("flush(f) makes sure that buffered data is written out."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FileCategory|
  (progn
    (push '|FileCategory| *Categories*)
    (make-instance '|FileCategoryType|)))
```

———–

1.36.5 FileNameCategory

— defclass FileNameCategoryType —

```
(defclass |FileNameCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "FileNameCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FNCAT)
   (comment :initform (list
     "This category provides an interface to names in the file system."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(|IOMode|) '(%)
       '("iomode(f) returns the status of the file f. The input/output"
         "status of f may be \"input\", \"output\" or \"closed\" mode."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FileNameCategory|
  (progn
    (push '|FileNameCategory| *Categories*)
    (make-instance '|FileNameCategoryType|)))
```

———-

1.36.6 Finite

— defclass FiniteType —

```
(defclass |FiniteType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Finite")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FINITE)
   (comment :initform (list
     "The category of domains composed of a finite set of elements."
     "We include the functions lookup and index"
     "to give a bijection between the finite set and an initial"
     "segment of positive integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|size| '(|NonNegativeInteger|) ()
       '("size() returns the number of elements in the set."))
     (make-signature '|index| '(%)  '(|PositiveInteger|)
       '("index(i) takes a positive integer i less than or equal"
         "to size() and"
         "returns the i-th element of the set."
         "This operation establishs a bijection"
         "between the elements of the finite set and 1..size()."))
     (make-signature '|lookup| '(|PositiveInteger|) '(%)
      '("lookup(x) returns a positive integer such that"
        "x = index lookup x."))
     (make-signature '|random| '(%)  '()
       '("random() returns a random element from the set."))
     (make-signature '|enumerate| '(|List %|) '()
       '("enumerate() returns a list of elements of the set")
       '("enumerate()$OrderedVariableList([p,q])"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Finite|
  (progn
    (push '|Finite| *Categories*)
    (make-instance '|FiniteType|)))
```

———-

1.36.7 FortranFunctionCategory

— defclass FortranFunctionCategoryType —

```
(defclass |FortranFunctionCategoryType| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "FortranFunctionCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FORTFN)
   (comment :initform (list
     "FortranFunctionCategory is the category of arguments to"
     "NAG Library routines which return (sets of) function values."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(%) '((|List| |FortranCode|))
       '("coerce(e) takes an object from List FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%)  '(|FortranCode|)
       '("coerce(e) takes an object from FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%)
                    '(|Record(localSymbols:SymbolTable,code:List(FortranCode))|)
       '("coerce(e) takes the component of e from"
         "List FortranCode and uses it as the body of the ASP,"
         "making the declarations in the SymbolTable component."))
     (make-signature '|retract| '(%) '((|Expression| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed")) '((|Expression| |Float|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%)  '((|Expression| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed")) '((|Expression| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Polynomial| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed")) '((|Polynomial| |Float|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Polynomial| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed")) '((|Polynomial| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Fraction| (|Polynomial| |Float|)))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                '((|Fraction| (|Polynomial| |Float|)))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Fraction| (|Polynomial| |Integer|)))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                '((|Fraction| (|Polynomial| |Integer|)))
```

```
      '("retractIfCan(e) tries to convert e into an ASP, checking that"
        "legal Fortran-77 is produced."))

   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranFunctionCategory|
  (progn
    (push '|FortranFunctionCategory| *Categories*)
    (make-instance '|FortranFunctionCategoryType|)))
```

———-

1.36.8 FortranMatrixCategory

— defclass FortranMatrixCategoryType —

```
(defclass |FortranMatrixCategoryType| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "FortranMatrixCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FMC)
   (comment :initform (list
     "FortranMatrixCategory provides support for"
     "producing Functions and Subroutines when the input to these"
     "is an AXIOM object of type Matrix or in domains"
     "involving FortranCode."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(%)  '((|Matrix| |MachineFloat|))
       '("coerce(v) produces an ASP which returns the value of v."))
     (make-signature '|coerce| '(%) '((|List| |FortranCode|))
       '("coerce(e) takes an object from List FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%) '(|FortranCode|)
       '("coerce(e) takes an object from List FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%)
                     '((|Record| |localSymbols:SymbolTable| |code:List(FortranCode)|))
      '("coerce(e) takes the component of e from"
        "List FortranCode and uses it as the body of the ASP,"
        "making the declarations in the SymbolTable component."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranMatrixCategory|
  (progn
    (push '|FortranMatrixCategory| *Categories*)
    (make-instance '|FortranMatrixCategoryType|)))
```

———-

1.36.9 FortranMatrixFunctionCategory

— defclass FortranMatrixFunctionCategoryType —

```
(defclass |FortranMatrixFunctionCategoryType| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "FortranMatrixFunctionCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FMFUN)
   (comment :initform (list
     "FortranMatrixFunctionCategory provides support for"
     "producing Functions and Subroutines representing matrices of"
     "expressions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(%) '((|List| |FortranCode|))
       '("coerce(e) takes an object from List FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%) '(|FortranCode|)
       '("coerce(e) takes an object from FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%)
                     '((|Record| |localSymbols:SymbolTable| |code:List(FortranCode)|))
      '("coerce(e) takes the component of e from"
        "List FortranCode and uses it as the body of the ASP,"
        "making the declarations in the SymbolTable component."))
     (make-signature '|retract| '(%) '((|Matrix| |Expression| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                 '((|Matrix| |Expression| |Float|))
       '("d retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Matrix| |Expression| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                 '((|Matrix| |Expression| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Matrix| |Polynomial| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                 '((|Matrix| |Polynomial| |Float|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Matrix| |Polynomial| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                 '((|Matrix| |Polynomial| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Matrix| |Fraction| |Polynomial| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                     '((|Matrix| |Fraction| |Polynomial| |Float|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Matrix| |Fraction| |Polynomial| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                     '((|Matrix| |Fraction| |Polynomial| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranMatrixFunctionCategory|
  (progn
    (push '|FortranMatrixFunctionCategory| *Categories*)
    (make-instance '|FortranMatrixFunctionCategoryType|)))
```

———-

1.36.10 FortranVectorCategory

— defclass FortranVectorCategoryType —

```
(defclass |FortranVectorCategoryType| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "FortranVectorCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FVC)
   (comment :initform (list
     "FortranVectorCategory provides support for"
     "producing Functions and Subroutines when the input to these"
     "is an AXIOM object of type Vector or in domains"
     "involving FortranCode."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(%)  '((|Vector| |MachineFloat|))
       '("coerce(v) produces an ASP which returns the value of v."))
     (make-signature '|coerce| '(%) '((|List| |FortranCode|))
       '("coerce(e) takes an object from List FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%) '(|FortranCode|)
       '("coerce(e) takes an object from FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%)
                    '((|Record| |localSymbols:SymbolTable| |code:List(FortranCode)|))
      '("coerce(e) takes the component of e from"
        "List FortranCode and uses it as the body of the ASP,"
        "making the declarations in the SymbolTable component."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranVectorCategory|
  (progn
    (push '|FortranVectorCategory| *Categories*)
    (make-instance '|FortranVectorCategoryType|)))
```

———-

1.36.11 FortranVectorFunctionCategory

— defclass FortranVectorFunctionCategoryType —

```
(defclass |FortranVectorFunctionCategoryType| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "FortranVectorFunctionCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FVFUN)
   (comment :initform (list
     "FortranVectorFunctionCategory is the catagory of arguments"
     "to NAG Library routines which return the values of vectors of functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|coerce| '(%) '((|List| |FortranCode|))
       '("coerce(e) takes an object from List FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%) '(|FortranCode|)
       '("coerce(e) takes an object from FortranCode and"
         "uses it as the body of an ASP."))
     (make-signature '|coerce| '(%)
                 '((|Record| |localSymbols:SymbolTable| |code:List(FortranCode)|))
      '("coerce(e) takes the component of e from"
        "List FortranCode and uses it as the body of the ASP,"
        "making the declarations in the SymbolTable component."))
     (make-signature '|retract| '(%) '((|Vector| |Expression| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                '((|Vector| |Expression| |Float|))
       '("d retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Vector| |Expression| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                '((|Vector| |Expression| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Vector| |Polynomial| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                '((|Vector| |Polynomial| |Float|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
        "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Vector| |Polynomial| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                   '((|Vector| |Polynomial| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Vector| |Fraction| |Polynomial| |Float|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                   '((|Vector| |Fraction| |Polynomial| |Float|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retract| '(%) '((|Vector| |Fraction| |Polynomial| |Integer|))
       '("retract(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
     (make-signature '|retractIfCan| '((|Union| % "failed"))
                                   '((|Vector| |Fraction| |Polynomial| |Integer|))
       '("retractIfCan(e) tries to convert e into an ASP, checking that"
         "legal Fortran-77 is produced."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranVectorFunctionCategory|
  (progn
    (push '|FortranVectorFunctionCategory| *Categories*)
    (make-instance '|FortranVectorFunctionCategoryType|)))
```

———-

1.36.12 FullyEvalableOver

— defclass FullyEvalableOverType —

```
(defclass |FullyEvalableOverType| (|EltableType| |EvalableType|)
  ((parents :initform '(|Eltable| |Evalable|))
   (name :initform "FullyEvalableOver")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'FEVALAB)
   (comment :initform (list
     "This category provides a selection of evaluation operations"
     "depending on what the argument type R provides."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullyEvalableOver|
  (progn
    (push '|FullyEvalableOver| *Categories*)
    (make-instance '|FullyEvalableOverType|)))
```

———-

1.36.13 GradedModule

— defclass GradedModuleType —

```
(defclass |GradedModuleType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "GradedModule")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'GRMOD)
   (comment :initform (list
     "GradedModule(R,E) denotes 'E-graded R-module', that is, collection of"
     "R-modules indexed by an abelian monoid E."
     "An element g of G[s] for some specific s in E"
     "is said to be an element of G with degree s."
     "Sums are defined in each module G[s] so two elements of G"
     "have a sum if they have the same degree."
     " "
     "Morphisms can be defined and composed by degree to give the"
     "mathematical category of graded modules."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|degree| '(|E|) '(%)
       '("degree(g) names the degree of g.  The set of all elements"
         "of a given degree form an R-module."))
     (make-signature '|0| '(%)  '(|constant|)
       '("0 denotes the zero of degree 0."))
     (make-signature '|*| '(%)  '(|(R, %)|)
       '("r*g is left module multiplication.") () t)
     (make-signature '|*| '(%)  '(|(%, R)|)
       '("g*r is right module multiplication.") () t)
     (make-signature '|-| '(%)  '(%)
       '("-g is the additive inverse of g in the module of elements"
         "of the same grade as g.") () t)
     (make-signature '|+| '(%)  '(|(%, %)|)
       '("g+h is the sum of g and h in the module of elements of"
         "the same degree as g and h.  Error: if g and h"
         "have different degrees." () t))
     (make-signature '|-| '(%)  '(|(%, %)|)
       '("g-h is the difference of g and h in the module of elements of"
         "the same degree as g and h.  Error: if g and h"
         "have different degrees.") () t)
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GradedModule|
  (progn
    (push '|GradedModule| *Categories*)
    (make-instance '|GradedModuleType|)))
```

———-

1.36.14 HomogeneousAggregate

— defclass HomogeneousAggregateType —

```
(defclass |HomogeneousAggregateType| (|AggregateType| |EvalableType| |SetCategoryType|)
  ((parents  :initform '(|SetCategory| |Aggregate| |Evalable|))
   (name :initform "HomogeneousAggregate")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'HOAGG)
   (comment :initform (list
     "A homogeneous aggregate is an aggregate of elements all of the"
     "same type."
     "In the current system, all aggregates are homogeneous."
     "Two attributes characterize classes of aggregates."
     "Aggregates from domains with attribute finiteAggregate"
     "have a finite number of members."
     "Those with attribute shallowlyMutable allow an element"
     "to be modified or updated without changing its overall value."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HomogeneousAggregate|
  (progn
    (push '|HomogeneousAggregate| *Categories*)
    (make-instance '|HomogeneousAggregateType|)))
```

———-

1.36.15 IndexedDirectProductCategory

— defclass IndexedDirectProductCategoryType —

```
(defclass |IndexedDirectProductCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "IndexedDirectProductCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'IDPC)
   (comment :initform (list
     "This category represents the direct product of some set with"
     "respect to an ordered indexing set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|map| '|%| '((|->| A A))
       '("map(f,z) returns the new element created by applying the"
         "function f to each component of the direct product element z."))
     (make-signature '|monomial| '(|%|) '(A S)
       '("monomial(a,s) constructs a direct product element with the s"
         "component set to a"))
     (make-signature '|leadingCoefficient| '(A) '(|%|)
       '("leadingCoefficient(z) returns the coefficient of the leading"
         "(with respect to the ordering on the indexing set)"
         "monomial of z."
         "Error: if z has no support."))
     (make-signature '|leadingSupport| '(S) '(|%|)
       '("leadingSupport(z) returns the index of leading"
         "(with respect to the ordering on the indexing set) monomial of z."
         "Error: if z has no support."))
     (make-signature '|reductum| '(|%|) '(|%|)
       '("reductum(z) returns a new element created by removing the"
         "leading coefficient/support pair from the element z."
         "Error: if z has no support."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductCategory|
  (progn
    (push '|IndexedDirectProductCategory| *Categories*)
    (make-instance '|IndexedDirectProductCategoryType|)))
```

————-

1.36.16 LiouvillianFunctionCategory

— defclass LiouvillianFunctionCategoryType —

```
(defclass |LiouvillianFunctionCategoryType| (|PrimitiveFunctionCategoryType|
                                             |TranscendentalFunctionCategoryType|)
  ((parents :initform '(|PrimitiveFunctionCategory| |TranscendentalFunctionCategory|))
   (name :initform "LiouvillianFunctionCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'LFCAT)
   (comment :initform (list
     "Category for the transcendental Liouvillian functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|Ei| '(|%|) '(|%|)
      '("Ei(x) returns the exponential integral of x, that is,"
        "the integral of exp(x)/x dx."))

    (make-signature '|Si| '(|%|) '(|%|)
      '("Si(x) returns the sine integral of x, that is,"
        "the integral of sin(x) / x dx."))

    (make-signature '|Ci| '(|%|) '(|%|)
      '("Ci(x) returns the cosine integral of x, that is,"
        "the integral of cos(x) / x dx."))

    (make-signature '|li| '(|%|) '(|%|)
      '("li(x) returns the logarithmic integral of x, that is,"
        "the integral of dx / log(x)."))

    (make-signature '|dilog| '(|%|) '(|%|)
      '("dilog(x) returns the dilogarithm of x, that is,"
        "the integral of log(x) / (1 - x) dx."))

    (make-signature '|erf| '(|%|) '(|%|)
      '("erf(x) returns the error function of x, that is,"
        "2 / sqrt(%pi) times the integral of exp(-x**2) dx."))

    (make-signature '|fresnelS| '(|%|) '(|%|)
      '("fresnelS(x) is the Fresnel integral S, defined by"
        "S(x) = integrate(sin(t^2),t=0..x)"))

    (make-signature '|fresnelC| '(|%|) '(|%|)
      '("fresnelC(x) is the Fresnel integral C, defined by"
        "C(x) = integrate(cos(t^2),t=0..x)"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LiouvillianFunctionCategory|
  (progn
    (push '|LiouvillianFunctionCategory| *Categories*)
    (make-instance '|LiouvillianFunctionCategoryType|)))
```

———-

1.36.17 Monad

— defclass MonadType —

```
(defclass |MonadType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Monad")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'MONAD)
   (comment :initform (list
     "Monad is the class of all multiplicative monads, that is sets"
     "with a binary operation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
    (make-signature '|*| '(|%|) '(|%| |%|)
      '("a*b is the product of a and b in a set with"
        "a binary operation.") () t)
    (make-signature '|rightPower| '(|%|) '(|%| |PositiveInteger|)
      '("rightPower(a,n) returns the n-th right power of a,"
        "that is, rightPower(a,n) := rightPower(a,n-1) * a and"
        "rightPower(a,1) := a."))
    (make-signature '|leftPower| '(|%|) '(|%| |PositiveInteger|)
      '("leftPower(a,n) returns the n-th left power of a,"
        "that is, leftPower(a,n) := a * leftPower(a,n-1) and"
        "leftPower(a,1) := a."))
    (make-signature '|**| '(|%|) '(|%| |PositiveInteger|)
       '("a**n returns the n-th power of a,"
         "defined by repeated squaring.") () t)
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Monad|
  (progn
    (push '|Monad| *Categories*)
    (make-instance '|MonadType|)))
```

———-

1.36.18 NumericalIntegrationCategory

— defclass NumericalIntegrationCategoryType —

```
(defclass |NumericalIntegrationCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "NumericalIntegrationCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'NUMINT)
   (comment :initform (list
     "NumericalIntegrationCategory is the category for"
     "describing the set of Numerical Integration domains with"
     "measure and numericalIntegration."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalIntegrationCategory|
  (progn
    (push '|NumericalIntegrationCategory| *Categories*)
    (make-instance '|NumericalIntegrationCategoryType|)))
```

———-

1.36.19 NumericalOptimizationCategory

— defclass NumericalOptimizationCategoryType —

```
(defclass |NumericalOptimizationCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "NumericalOptimizationCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'OPTCAT)
   (comment :initform (list
     "NumericalOptimizationCategory is the category for"
     "describing the set of Numerical Optimization domains with"
     "measure and optimize."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalOptimizationCategory|
  (progn
    (push '|NumericalOptimizationCategory| *Categories*)
    (make-instance '|NumericalOptimizationCategoryType|)))
```

———-

1.36.20 OrderedSet

— defclass OrderedSetType —

```
(defclass |OrderedSetType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "OrderedSet")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'ORDSET)
   (comment :initform (list
     "The class of totally ordered sets, that is, sets such that for each "
     "pair of elements (a,b)"
     "exactly one of the following relations holds a<b or a=b or b<a"
     "and the relation is transitive, that is, a<b and b<c => a<c."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
    (make-signature '|<| '(|Boolean|) '(|%| |%|)
      '("x < y is a strict total ordering on the elements of the set.") () t)
    (make-signature '|>| '(|Boolean|) '(|%| |%|)
      '("x > y is a greater than test.") () t)
    (make-signature '|>=| '(|Boolean|) '(|%| |%|)
      '("x >= y is a greater than or equal test.") () t)
    (make-signature '|<=| '(|Boolean|) '(|%| |%|)
      '("x <= y is a less than or equal test.") () t)
    (make-signature '|max| '(|%|) '(|%| |%|)
      '("max(x,y) returns the maximum of x and y relative to '<'"))
    (make-signature '|min| '(|%|) '(|%| |%|)
      '("min(x,y) returns the minimum of x and y relative to '<'"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedSet|
  (progn
    (push '|OrderedSet| *Categories*)
    (make-instance '|OrderedSetType|)))
```

———-

1.36.21 OrdinaryDifferentialEquationsSolverCategory

— defclass OrdinaryDifferentialEquationsSolverCategoryType —

```
(defclass |OrdinaryDifferentialEquationsSolverCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "OrdinaryDifferentialEquationsSolverCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'ODECAT)
   (comment :initform (list
     "OrdinaryDifferentialEquationsSolverCategory is the"
     "category for describing the set of ODE solver domains"
     "with measure and ODEsolve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdinaryDifferentialEquationsSolverCategory|
  (progn
    (push '|OrdinaryDifferentialEquationsSolverCategory| *Categories*)
    (make-instance '|OrdinaryDifferentialEquationsSolverCategoryType|)))
```

———-

1.36.22 PartialDifferentialEquationsSolverCategory

— defclass PartialDifferentialEquationsSolverCategoryType —

```
(defclass |PartialDifferentialEquationsSolverCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "PartialDifferentialEquationsSolverCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'PDECAT)
   (comment :initform (list
     "PartialDifferentialEquationsSolverCategory is the"
     "category for describing the set of PDE solver domains"
     "with measure and PDEsolve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialDifferentialEquationsSolverCategory|
  (progn
    (push '|PartialDifferentialEquationsSolverCategory| *Categories*)
    (make-instance '|PartialDifferentialEquationsSolverCategoryType|)))
```

———-

1.36.23 PatternMatchable

— defclass PatternMatchableType —

```
(defclass |PatternMatchableType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "PatternMatchable")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'PATMAB)
   (comment :initform (list
     "A set R is PatternMatchable over S if elements of R can"
     "be matched to patterns over S."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
    (make-signature '|sign| '(Z) '(|ThePols| |%|)
      '("sign(pol,aRoot) gives the sign of pol interpreted as aRoot"))
    (make-signature '|zero?| '(|Boolean|) '(|ThePols| |%|)
      '("zero?(pol,aRoot) answers if pol interpreted as aRoot is 0"))
    (make-signature '|negative?| '(|Boolean|) '(|ThePols| |%|)
      '("negative?(pol,aRoot) answers if pol interpreted as aRoot is negative"))
    (make-signature '|positive?| '(|Boolean|) '(|ThePols| |%|)
      '("positive?(pol,aRoot) answers if pol interpreted as aRoot is positive"))
    (make-signature '|recip| '((|Union| |ThePols| "failed")) '(|ThePols| |%|)
      '("recip(pol,aRoot) tries to inverse pol interpreted as aRoot"))
    (make-signature '|definingPolynomial| '(|ThePols|) '(|%|)
      '("definingPolynomial(aRoot) gives a polynomial"
        "such that definingPolynomial(aRoot).aRoot = 0"))
    (make-signature '|allRootsOf| '((|List| |%|))  '(|ThePols|)
      '("allRootsOf(pol) creates all the roots of pol"
        "in the Real Closure, assumed in order."))
    (make-signature '|rootOf| '(|Union| $ "failed") '(|ThePols| N)
      '("rootOf(pol,n) gives the nth root for the order of the Real Closure"))
    (make-signature '|approximate| '(|TheField|) '(|ThePols| $ |TheField|)
      '("approximate(term,root,prec) gives an approximation"
        "of term over root with precision prec"))
    (make-signature '|relativeApprox| '(|TheField|) '(|ThePols| $ |TheField|)
      '("approximate(term,root,prec) gives an approximation"
        "of term over root with precision prec"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchable|
  (progn
    (push '|PatternMatchable| *Categories*)
    (make-instance '|PatternMatchableType|)))
```

———-

1.36.24 RealRootCharacterizationCategory

— defclass RealRootCharacterizationCategoryType —

```
(defclass |RealRootCharacterizationCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "RealRootCharacterizationCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'RRCC)
   (comment :initform (list
     "RealRootCharacterizationCategory provides common access"
     "functions for all real roots of polynomials"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|sign| '(Z) '(|ThePols| %)
       '("sign(pol,aRoot) gives the sign of pol"
         "interpreted as aRoot"))
     (make-signature '|zero?| '(|Boolean|) '(|ThePols| %)
       '("zero?(pol,aRoot) answers if pol"
         "interpreted as aRoot is 0"))
     (make-signature '|negative?| '(|Boolean|) '(|ThePols| %)
       '("negative?(pol,aRoot) answers if pol"
         "interpreted as aRoot is negative"))
     (make-signature '|positive?| '(|Boolean|) '(|ThePols| %)
       '("positive?(pol,aRoot) answers if pol"
         "interpreted as aRoot is positive"))
     (make-signature '|recip| '((|Union| |ThePols| "failed")) '(|ThePols| %)
       '("recip(pol,aRoot) tries to inverse pol"
         "interpreted as aRoot"))
     (make-signature '|definingPolynomial| '(|ThePols|) '(%)
       '("definingPolynomial(aRoot) gives a polynomial"
         "such that definingPolynomial(aRoot).aRoot = 0"))
     (make-signature '|allRootsOf| '((|List| %)) '(|ThePols|)
       '("allRootsOf(pol) creates all the roots of pol"
         "in the Real Closure, assumed in order."))
     (make-signature '|rootOf| '((|Union| % "failed")) '(|ThePols| N)
       '("rootOf(pol,n) gives the nth root for the order of the"
         "Real Closure"))
     (make-signature '|approximate| '(|TheField|) '(|ThePols| % |TheField|)
       '("approximate(term,root,prec) gives an approximation"
         "of term over root with precision prec"))
     (make-signature '|relativeApprox| '(|TheField|) '(|ThePols| % |TheField|)
       '("approximate(term,root,prec) gives an approximation"
         "of term over root with precision prec"))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealRootCharacterizationCategory|
  (progn
    (push '|RealRootCharacterizationCategory| *Categories*)
    (make-instance '|RealRootCharacterizationCategoryType|)))
```

———-

1.36.25 SExpressionCategory

— defclass SExpressionCategoryType —

```
(defclass |SExpressionCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SExpressionCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'SEXCAT)
   (comment :initform (list
     "This category allows the manipulation of Lisp values while keeping"
     "the grunge fairly localized."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|eq| '(|Boolean|) '(% %)
       '("eq(s, t) is true if EQ(s,t) is true in Lisp."))

     (make-signature '|null?| '(|Boolean|) '(%)
       '("null?(s) is true if s is the S-expression ()."))

     (make-signature '|atom?| '(|Boolean|) '(%)
       '("atom?(s) is true if s is a Lisp atom."))

     (make-signature '|pair?| '(|Boolean|) '(%)
       '("pair?(s) is true if s has is a non-null Lisp list."))

     (make-signature '|list?| '(|Boolean|) '(%)
       '("list?(s) is true if s is a Lisp list, possibly ()."))

     (make-signature '|string?| '(|Boolean|) '(%)
       '("string?(s) is true if s is an atom and belong to Str."))

     (make-signature '|symbol?| '(|Boolean|) '(%)
       '("symbol?(s) is true if s is an atom and belong to Sym."))

     (make-signature '|integer?| '(|Boolean|) '(%)
       '("integer?(s) is true if s is an atom and belong to Int."))

     (make-signature '|float?| '(|Boolean|) '(%)
       '("float?(s) is true if s is an atom and belong to Flt."))

     (make-signature '|destruct| '(|List %|) '(%)
       '("destruct((a1,...,an)) returns the list [a1,...,an]."))

     (make-signature '|string| '(|Str|) '(%)
       '("string(s) returns s as an element of Str."
         "Error: if s is not an atom that also belongs to Str."))

    (make-signature '| symbol| '(|Sym|) '(%)
       '("symbol(s) returns s as an element of Sym."
         "Error: if s is not an atom that also belongs to Sym."))

    (make-signature '| integer| '(|Int|) '(%)
       '("integer(s) returns s as an element of Int."
         "Error: if s is not an atom that also belongs to Int."))
    (make-signature '| float| '(|Flt|) '(%)
       '("float(s) returns s as an element of Flt;"
         "Error: if s is not an atom that also belongs to Flt."))

    (make-signature '| expr| '(|Expr|) '(%)
       '("expr(s) returns s as an element of Expr;"
         "Error: if s is not an atom that also belongs to Expr."))

    (make-signature '| convert| '(%) '(|List| %)
       '("convert([a1,...,an]) returns an S-expression (a1,...,an)."))

     (make-signature '|convert| '(%) '(|Str|)
       '("convert(x) returns the Lisp atom x;"))

     (make-signature '|convert| '(%) '(|Sym|)
       '("convert(x) returns the Lisp atom x."))

     (make-signature '|convert| '(%) '(|Int|)
       '("convert(x) returns the Lisp atom x."))

     (make-signature '|convert| '(%) '(|Flt|)
       '("convert(x) returns the Lisp atom x."))

     (make-signature '|convert| '(%) '(|Expr|)
       '("convert(x) returns the Lisp atom x."))

     (make-signature '|car| '(%) '(%)
       '("car((a1,...,an)) returns a1."))

     (make-signature '|cdr| '(%) '(%)
       '("cdr((a1,...,an)) returns (a2,...,an)."))

     (make-signature '|"#"| '(|Integer|) '(%)
       '("#((a1,...,an)) returns n."))

     (make-signature '|elt| '(%) '(% |Integer|)
       '("elt((a1,...,an), i) returns ai."))

     (make-signature '|elt| '(%) '(% (|List| |Integer|))
       '("elt((a1,...,an), [i1,...,im]) returns (a_i1,...,a_im)."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SExpressionCategory|
  (progn
    (push '|SExpressionCategory| *Categories*)
    (make-instance '|SExpressionCategoryType|)))
```

———-

1.36.26 SegmentExpansionCategory

— defclass SegmentExpansionCategoryType —

```
(defclass |SegmentExpansionCategoryType| (|SegmentCategoryType|)
  ((parents :initform '(|SegmentCategory|))
   (name :initform "SegmentExpansionCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'SEGXCAT)
   (comment :initform (list
     "This category provides an interface for expanding segments to"
     "a stream of elements."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|expand| '(L) '(|List| %)
       '("expand(l) creates a new value of type L in which each segment"
         "l..h by k is replaced with l, l+k, ... lN,"
         "where lN <= h < lN+k."
         "For example, expand [1..4, 7..9] = [1,2,3,4,7,8,9]."))
     (make-signature '|expand| '(L) '(%)
       '("expand(l..h by k) creates value of type L with elements"
         "l, l+k, ... lN where lN <= h < lN+k."
         "For example, expand(1..5 by 2) = [1,3,5]."))
     (make-signature '|map| '(L) '(|S -> S| %)
       '("map(f,l..h by k) produces a value of type L by applying f"
         "to each of the succesive elements of the segment, that is,"
         "[f(l), f(l+k), ..., f(lN)], where lN <= h < lN+k."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SegmentExpansionCategory|
  (progn
    (push '|SegmentExpansionCategory| *Categories*)
    (make-instance '|SegmentExpansionCategoryType|)))
```

———-

1.36.27 SemiGroup

— defclass SemiGroupType —

```
(defclass |SemiGroupType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SemiGroup")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'SGROUP)
   (comment :initform (list
     "The class of all multiplicative semigroups, that is, a set"
     "with an associative operation *."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|*| '(%) '(% %)
       '("x*y returns the product of x and y.") () t)

     (make-signature '|**| '(%) '(% |PositiveInteger|)
       '("x**n returns the repeated product of x n times, exponentiation.") () t)
     (make-signature '|^| '(%) '(% |PositiveInteger|)
       '("x^n returns the repeated product of x n times, exponentiation.") () t)
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SemiGroup|
  (progn
    (push '|SemiGroup| *Categories*)
    (make-instance '|SemiGroupType|)))
```

———-

1.36.28 SetCategoryWithDegree

— defclass SetCategoryWithDegreeType —

```
(defclass |SetCategoryWithDegreeType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SetCategoryWithDegree")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'SETCATD)
   (comment :initform (list
     "This is part of the PAFF package, related to projective space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|degree| '(|PositiveInteger|) '(%) ())
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SetCategoryWithDegree|
  (progn
    (push '|SetCategoryWithDegree| *Categories*)
    (make-instance '|SetCategoryWithDegreeType|)))
```

———-

1.36.29 StepThrough

— defclass StepThroughType —

```
(defclass |StepThroughType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "StepThrough")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'STEP)
   (comment :initform (list
     "A class of objects which can be 'stepped through'."
     "Repeated applications of nextItem is guaranteed never to"
     "return duplicate items and only return 'failed' after exhausting"
     "all elements of the domain."
     "This assumes that the sequence starts with init()."
     "For infinite domains, repeated application"
     "of nextItem is not required to reach all possible domain elements"
     "starting from any initial element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|init| '(%) '(|constant|)
       '("init() chooses an initial object for stepping."))

     (make-signature '|nextItem| '(|Union| % "failed") '(%)
       '("nextItem(x) returns the next item, or \"failed\""
         "if domain is exhausted."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StepThrough|
  (progn
    (push '|StepThrough| *Categories*)
    (make-instance '|StepThroughType|)))
```

———-

1.36.30 ThreeSpaceCategory

— defclass ThreeSpaceCategoryType —

```
(defclass |ThreeSpaceCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "ThreeSpaceCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'SPACEC)
   (comment :initform (list
     "The category ThreeSpaceCategory is used for creating"
     "three dimensional objects using functions for defining points, curves,"
     "polygons, constructs and the subspaces containing them."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|create3Space| '(%) '()
       '("create3Space() creates a ThreeSpace object capable of "
         "holding point, curve, mesh components and any combination."))

     (make-signature '|create3Space| '(%) '(SUBSPACE)
       '("create3Space(s) creates a ThreeSpace object containing"
         "objects pre-defined within some SubSpace s."))

     (make-signature '|numberOfComponents| '(NNI) '(%)
       '("numberOfComponents(s) returns the number of distinct"
         "object components in the indicated ThreeSpace, s, such"
```

```
    "as points, curves, polygons, and constructs."))

(make-signature '|numberOfComposites| '(NNI) '(%)
  '("numberOfComposites(s) returns the number of supercomponents,"
    "or composites, in the ThreeSpace, s; Composites are "
    "arbitrary groupings of otherwise distinct and unrelated components;"
    "A ThreeSpace need not have any composites defined at all"
    "and, outside of the requirement that no component can belong"
    "to more than one composite at a time, the definition and"
    "interpretation of composites are unrestricted."))

(make-signature '|merge| '(%) '((|List| %))
  '("merge([s1,s2,...,sn]) will create a new ThreeSpace that"
    "has the components of all the ones in the list; Groupings of "
    "components into composites are maintained."))

(make-signature '|merge| '(%) '(% %)
  '("merge(s1,s2) will create a new ThreeSpace that has the"
    "components of s1 and s2; Groupings of components"
    "into composites are maintained."))

(make-signature '|composite| '(%) '((|List| %))
  '("composite([s1,s2,...,sn]) will create a new ThreeSpace"
    "that is a union of all the components from each "
    "ThreeSpace in the parameter list, grouped as a composite."))

(make-signature '|components| '(|List| %) '(%)
  '("components(s) takes the ThreeSpace s, and creates a list "
    "containing a unique ThreeSpace for each single component "
    "of s. If s has no components defined, the list returned is empty."))

(make-signature '|composites| '((|List| %)) '(%)
  '("composites(s) takes the ThreeSpace s, and creates a list "
    "containing a unique ThreeSpace for each single composite "
    "of s. If s has no composites defined (composites need to be "
    "explicitly created), the list returned is empty. Note that not all "
    "the components need to be part of a composite."))

(make-signature '|copy| '(%) '(%)
  '("copy(s) returns a new ThreeSpace that is an exact copy "
    "of s."))

(make-signature '|enterPointData| '(NNI) '(% (|List| POINT))
  '("enterPointData(s,[p0,p1,...,pn]) adds a list of points from p0 "
    "through pn to the ThreeSpace, s, and returns the index, "
    "to the starting point of the list;"))

(make-signature '|modifyPointData| '(%) '(% NNI POINT)
  '("modifyPointData(s,i,p) changes the point at the indexed "
    "location i in the ThreeSpace, s, to that of point p."
    "This is useful for making changes to a point which has been "
    "transformed."))

(make-signature '|point| '(%) '(% POINT)
  '("point(s,p) adds a point component defined by the point, p, "
    "specified as a list from List(R), to the "
    "ThreeSpace, s, where R is the Ring over "
    "which the point is defined."))
```

```
     (make-signature '|point| '(%) '(% (|List| R))
       '("point(s,[x,y,z]) adds a point component defined by a list of "
         "elements which are from the PointDomain(R) to the "
         "ThreeSpace, s, where R is the Ring over "
         "which the point elements are defined."))

     (make-signature '|point| '(%) '(% NNI)
       '("point(s,i) adds a point component which is placed into a component"
         "list of the ThreeSpace, s, at the index given by i."))

     (make-signature '|point| '(%) '(POINT)
       '("point(p) returns a ThreeSpace object which is composed "
         "of one component, the point p."))

     (make-signature '|point| '(POINT) '(%)
       '("point(s) checks to see if the ThreeSpace, s, is "
         "composed of only a single point and if so, returns the point. "
         "An error is signaled otherwise."))

     (make-signature '|point?| '(B) '(%)
       '("point?(s) queries whether the ThreeSpace, s, is "
         "composed of a single component which is a point and returns the "
         "boolean result."))

     (make-signature '|curve| '(%) '(% (|List| POINT))
       '("curve(s,[p0,p1,...,pn]) adds a space curve component defined by a "
         "list of points p0 through pn, to the "
         "ThreeSpace s."))

     (make-signature '|curve| '(%) '(% (|List| |List| R))
       '("curve(s,[[p0],[p1],...,[pn]]) adds a space curve which is a list of "
         "points p0 through pn defined by lists of elements from the domain "
         "PointDomain(m,R), where R is the Ring over which "
         "the point elements are defined and m is the dimension of the "
         "points, to the ThreeSpace s."))

;     (make-signature '|curve| '(%) '(|List| POINT)
;       '("curve([p0,p1,p2,...,pn]) creates a space curve defined"
;         "by the list of points p0 through pn, and returns the "
;         "ThreeSpace whose component is the curve."))
;
;     (make-signature '|curve| '(|List| POINT) '(%)
;       '("curve(s) checks to see if the ThreeSpace, s, is "
;         "composed of a single curve defined by a list of points and if so, "
;         "returns the curve, that is, list of points. An error is signaled "
;         "otherwise."))
;
;     (make-signature '|curve?| '(B) '(%)
;       '("curve?(s) queries whether the ThreeSpace, s, is a curve, "
;         "that is, has one component, a list of list of points, and returns "
;         "true if it is, or false otherwise."))
;
;     (make-signature '|closedCurve| '(%) '(% (|List| POINT))
;       '("closedCurve(s,[p0,p1,...,pn,p0]) adds a closed curve component "
;         "which is a list of points defined by the first element p0 through "
;         "the last element pn and back to the first element p0 again, to the "
;         "ThreeSpace s."))
;
;     (make-signature '|closedCurve| '(%) '(|List| (|List| R))
;       '("closedCurve(s,[[lr0],[lr1],...,[lrn],[lr0]]) adds a closed curve "
;         "component defined by a list of points lr0 through "
;         "lrn, which are lists of elements from the domain "
;         "PointDomain(m,R), where R is the Ring over which "
;         "the point elements are defined and m is the dimension of the "
;         "points, in which the last element of the list of points contains "
;         "a copy of the first element list, lr0."
;         "The closed curve is added to the ThreeSpace, s."))
;
;     (make-signature '|closedCurve| '(%) '(|List| POINT)
;       '("closedCurve(lp) sets a list of points defined by the first element"
;         "of lp through the last element of lp and back to the first elelment"
;         "again and returns a ThreeSpace whose component is the"
;         "closed curve defined by lp."))
;
;     (make-signature '|closedCurve| '(List| POINT) '(%)
;       '("closedCurve(s) checks to see if the ThreeSpace, s, is "
;         "composed of a single closed curve component defined by a list of "
;         "points in which the first point is also the last point, all of "
;         "which are from the domain PointDomain(m,R) and if so, "
;         "returns the list of points.  An error is signaled otherwise."))
;
;     (make-signature '|closedCurve?| '(B) '(%)
;       '("closedCurve?(s) returns true if the ThreeSpace s "))
;      ++ contains a single closed curve component, that is, the first element
;      ++ of the curve is also the last element, or false otherwise.
;
;     (make-signature '|polygon| '(%) '(|List| POINT)
;       '("polygon(s,[p0,p1,...,pn]) adds a polygon component defined by a "
;         "list of points, p0 throught pn, to the ThreeSpace s."))
;
;     (make-signature '|polygon| '(%) '(% |List| (|List| R))
;       '("polygon(s,[[r0],[r1],...,[rn]]) adds a polygon component defined"
;         "by a list of points r0 through rn, which are lists of"
;         "elements from the domain PointDomain(m,R) to the "
;         "ThreeSpace s, where m is the dimension of the points"
;         "and R is the Ring over which the points are defined."))
;
;     (make-signature '|polygon| '(%) '(|List| POINT)
;       '("polygon([p0,p1,...,pn]) creates a polygon defined by a list of "
;         "points, p0 through pn, and returns a ThreeSpace whose "
;         "component is the polygon."))
;
;     (make-signature '|polygon| '(|List| POINT) '(%)
;       '("polygon(s) checks to see if the ThreeSpace, s, is "
;         "composed of a single polygon component defined by a list of "
;         "points, and if so, returns the list of points;  An error is "
;         "signaled otherwise."))
;
;     (make-signature '|polygon?| '(B) '(%)
;       '("polygon?(s) returns true if the ThreeSpace s contains "
;         "a single polygon component, or false otherwise."))
;
;     (make-signature '|mesh| '(%) '(% (|List| (|List POINT)) (|List| PROP) PROP)
;       '("mesh(s,[[p0],[p1],...,[pn]],[props],prop) adds a surface component, "
;         "defined over a list curves which contains lists of points, to the "
;         "ThreeSpace s; props is a list which contains the "
;         "subspace component properties for each surface parameter, and "
;         "prop is the subspace component property by which the points are "
;          "defined."))
;
;     (make-signature '|mesh| '(%) '(% (|List| (|List| (|List| R))) (|List| PROP) PROP)
;       '("mesh(s, LLLR, [props], prop)"
;          "where LLLR is of the form:"
;          "[[[r10]...,[r1m]],[[r20]...,[r2m]],...,[[rn0]...,[rnm]]],"
;          "adds a surface component to the ThreeSpace s, which is "
;          "defined over a rectangular domain of size WxH where W is the number "
;          "of lists of points from the domain PointDomain(R) and H is "
;          "the number of elements in each of those lists; lprops is the list "
;          "of the subspace component properties for each curve list, and "
;          "prop is the subspace component property by which the points are "
;          "defined."))
;
;     (make-signature '|mesh| '(%) '(% (|List| (|List| POINT)) B B)
;       '("mesh(s, LLP, close1, close2) "
;          "where LLP is of the form [[p0],[p1],...,[pn]] adds a surface "
;          "component to the ThreeSpace, which is defined over a "
;          "list of curves, in which each of these curves is a list of points. "
;          "The boolean arguments close1 and close2 indicate how the surface "
;          "is to be closed. Argument close1 equal true"
;          "means that each individual list (a curve) is to be closed, that is,"
;          "the last point of the list is to be connected to the first point."
;          "Argument close2 equal true "
;          "means that the boundary at one end of the surface is to be"
;          "connected to the boundary at the other end, that is, the boundaries "
;          "are defined as the first list of points (curve) and "
;          "the last list of points (curve)."))
;
;     (make-signature '|mesh| '(%) '(% (|List| (|List| (|List| R))) B B)
;       '("mesh(s, LLLR, close1, close2)"
;          "where LLLR is of the form"
;          "[[[r10]...,[r1m]],[[r20]...,[r2m]],...,[[rn0]...,[rnm]]],"
;          "adds a surface component to the ThreeSpace s, which is "
;          "defined over a rectangular domain of size WxH where W is the number "
;          "of lists of points from the domain PointDomain(R) and H is "
;          "the number of elements in each of those lists; the booleans close1 "
;          "and close2 indicate how the surface is to be closed: if close1 is "
;          "true this means that each individual list (a curve) is to be "
;          "closed (that is,"
;          "the last point of the list is to be connected to the first point);"
;          "if close2 is true, this means that the boundary at one end of the"
;          "surface is to be connected to the boundary at the other end"
;          "(the boundaries are defined as the first list of points (curve)"
;          "and the last list of points (curve))."))
;
;     (make-signature '|mesh| '(%) '(|List| (|List| POINT))
;       '("mesh([[p0],[p1],...,[pn]]) creates a surface defined by a list of "
;          "curves which are lists, p0 through pn, of points, and returns a "
;          "ThreeSpace whose component is the surface."))
;
;     (make-signature '|mesh| '(%) '(|List| (|List| POINT) B B)
;       '("mesh([[p0],[p1],...,[pn]], close1, close2) creates a surface "
;          "defined over a list of curves, p0 through pn, which are lists of "
;          "points; the booleans close1 and close2 indicate how the surface is "
;          "to be closed: close1 set to true means that each individual list "
;          "(a curve) is to be closed (that is, the last point of the list is "
;          "to be connected to the first point); close2 set to true means "
;          "that the boundary at one end of the surface is to be connected to "
;         "the boundary at the other end (the boundaries are defined as the "
;         "first list of points (curve) and the last list of points (curve)); "
;         "the ThreeSpace containing this surface is returned."))
;
;     (make-signature '|mesh| '(|List| (|List| POINT)) '(%)
;       '("mesh(s) checks to see if the ThreeSpace, s, is "
;         "composed of a single surface component defined by a list curves "
;         "which contain lists of points, and if so, returns the list of "
;         "lists of points;  An error is signaled otherwise."))
;
;     (make-signature '|mesh?| '(B) '(%)
;       '("mesh?(s) returns true if the ThreeSpace s is composed "
;         "of one component, a mesh comprising a list of curves which are lists"
;         "of points, or returns false if otherwise"))
;
;     (make-signature '|lp| '(|List| POINT) '(%)
;       '("lp(s) returns the list of points component which the "
;         "ThreeSpace, s, contains; these points are used by "
;         "reference, that is, the component holds indices referring to the "
;         "points rather than the points themselves. This allows for sharing "
;         "of the points."))
;
;     (make-signature '|lllip| '(|List| (|List| (|List| NNI))) '(%)
;       '("lllip(s) checks to see if the ThreeSpace, s, is "
;         "composed of a list of components, which are lists of curves, "
;         "which are lists of indices to points, and if so, returns the list "
;         "of lists of lists;  An error is signaled otherwise."))
;
;     (make-signature '|lllp| '(|List| (|List| (|List| POINT))) '(%)
;       '("lllp(s) checks to see if the ThreeSpace, s, is "
;         "composed of a list of components, which are lists of curves, "
;         "which are lists of points, and if so, returns the list of "
;         "lists of lists;  An error is signaled otherwise."))
;
;     (make-signature '|llprop| '(|List| (|List| PROP)) '(%)
;       '("llprop(s) checks to see if the ThreeSpace, s, is "
;         "composed of a list of curves which are lists of the"
;         "subspace component properties of the curves, and if so, returns the "
;         "list of lists;  An error is signaled otherwise."))
;
;     (make-signature '|lprop| '(|List| PROP) '(%)
;       '("lprop(s) checks to see if the ThreeSpace, s, is "
;         "composed of a list of subspace component properties, and if so, "
;         "returns the list;  An error is signaled otherwise."))
;
;     (make-signature '|objects| '(OBJ3D) '(%)
;       '("objects(s) returns the ThreeSpace, s, in the form of a "
;         "3D object record containing information on the number of points, "
;         "curves, polygons and constructs comprising the "
;         "ThreeSpace.."))
;
;     (make-signature '|check| '(%) '(%)
;       '("check(s) returns lllpt, list of lists of lists of point information "
;         "about the ThreeSpace} s.")
;
;     (make-signature '|subspace| '(SUBSPACE) '(%)
;       '("subspace(s) returns the SubSpace which holds all the "
;         "point information in the ThreeSpace, s."))
;
```

```
;     (make-signature '|coerce| '(O) '(%)
;       '("coerce(s) returns the ThreeSpace s to Output format."))
   ))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ThreeSpaceCategory|
  (progn
    (push '|ThreeSpaceCategory| *Categories*)
    (make-instance '|ThreeSpaceCategoryType|)))
```

———-

1.37 Level 4

— defvar level4 —

```
(defvar level4
 '(|AbelianMonoid| |AffineSpaceCategory| |BagAggregate| |CachableSet|
   |Collection| |DifferentialVariableCategory| |ExpressionSpace|
   |FullyPatternMatchable| |GradedAlgebra| |IndexedAggregate|
   |InfinitlyClosePointCategory| |MonadWithUnit| |Monoid|
   |OrderedAbelianSemiGroup| |OrderedFinite| |PlacesCategory|
   |ProjectiveSpaceCategory| |RecursiveAggregate| |TwoDimensionalArrayCategory|))
```

———-

1.37.1 AbelianMonoid

— defclass AbelianMonoidType —

```
(defclass |AbelianMonoidType| (|AbelianSemiGroupType|)
  ((parents :initform '(|AbelianSemiGroup|))
   (name :initform "AbelianMonoid")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'ABELMON)
   (comment :initform (list
     "The class of multiplicative monoids, that is, semigroups with an"
     "additive identity element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AbelianMonoid|
  (progn
    (push '|AbelianMonoid| *Categories*)
    (make-instance '|AbelianMonoidType|)))
```

———-

1.37.2 AffineSpaceCategory

— defclass AffineSpaceCategoryType —

```
(defclass |AffineSpaceCategoryType| (|SetCategoryWithDegreeType|)
  ((parents :initform '(|SetCategoryWithDegree|))
   (name :initform "AffineSpaceCategory")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'AFSPCAT)
   (comment :initform (list
     "The following is all the categories and domains related to projective"
     "space and part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffineSpaceCategory|
  (progn
    (push '|AffineSpaceCategory| *Categories*)
    (make-instance '|AffineSpaceCategoryType|)))
```

——————-

1.37.3 BagAggregate

— defclass BagAggregateType —

```
(defclass |BagAggregateType| (|HomogeneousAggregateType|)
  ((parents :initform '(|HomogeneousAggregate|))
   (name :initform "BagAggregate")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'BGAGG)
   (comment :initform (list
     "A bag aggregate is an aggregate for which one can insert and extract"
     "objects, and where the order in which objects are inserted determines"
     "the order of extraction."
     "Examples of bags are stacks, queues, and dequeues."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BagAggregate|
  (progn
    (push '|BagAggregate| *Categories*)
    (make-instance '|BagAggregateType|)))
```

——————-

1.37.4 CachableSet

— defclass CachableSetType —

```
(defclass |CachableSetType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "CachableSet")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'CACHSET)
   (comment :initform (list
     "A cachable set is a set whose elements keep an integer as part"
     "of their structure."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CachableSet|
  (progn
    (push '|CachableSet| *Categories*)
    (make-instance '|CachableSetType|)))
```

————-

1.37.5 Collection

— defclass CollectionType —

```
(defclass |CollectionType| (|ConvertibleToType| |HomogeneousAggregateType|)
  ((parents  :initform '(|ConvertibleTo| |HomogeneousAggregate|))
   (name :initform "Collection")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'CLAGG)
   (comment :initform (list
     "A collection is a homogeneous aggregate which can built from"
     "list of members. The operation used to build the aggregate is"
     "generically named construct. However, each collection"
     "provides its own special function with the same name as the"
     "data type, except with an initial lower case letter, For example,"
     "list for List, flexibleArray for FlexibleArray, and so on."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Collection|
  (progn
    (push '|Collection| *Categories*)
    (make-instance '|CollectionType|)))
```

———-

1.37.6 DifferentialVariableCategory

— defclass DifferentialVariableCategoryType —

```
(defclass |DifferentialVariableCategoryType| (|RetractableToType| |OrderedSetType|)
  ((parents :initform '(|RetractableTo| |OrderedSet|))
   (name :initform "DifferentialVariableCategory")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'DVARCAT)
   (comment :initform (list
     "DifferentialVariableCategory constructs the"
     "set of derivatives of a given set of"
     "(ordinary) differential indeterminates."
     "If x,...,y is an ordered set of differential indeterminates,"
     "and the prime notation is used for differentiation, then"
     "the set of derivatives (including"
     "zero-th order) of the differential indeterminates is"
     "x,x',x'',..., y,y',y'',..."
     "(Note that in the interpreter, the n-th derivative of y is displayed as"
     "y with a subscript n.)  This set is"
     "viewed as a set of algebraic indeterminates, totally ordered in a"
     "way compatible with differentiation and the given order on the"
     "differential indeterminates.  Such a total order is called a"
     "ranking of the differential indeterminates."
     " "
     "A domain in this category is needed to construct a differential"
     "polynomial domain.  Differential polynomials are ordered"
     "by a ranking on the derivatives,  and by an order (extending the"
     "ranking) on"
     "on the set of differential monomials.  One may thus associate"
     "a domain in this category with a ranking of the differential"
     "indeterminates, just as one associates a domain in the category"
     "OrderedAbelianMonoidSup with an ordering of the set of"
     "monomials in a set of algebraic indeterminates.  The ranking"
     "is specified through the binary relation <."
     "For example, one may define"
     "one derivative to be less than another by lexicographically comparing"
     "first the order, then the given order of the differential"
     "indeterminates appearing in the derivatives.  This is the default"
     "implementation."
     " "
     "The notion of weight generalizes that of degree.  A"
     "polynomial domain may be made into a graded ring"
     "if a weight function is given on the set of indeterminates,"
     "Very often, a grading is the first step in ordering the set of"
     "monomials.  For differential polynomial domains, this"
     "constructor provides a function weight, which"
     "allows the assignment of a non-negative number to each derivative of a"
     "differential indeterminate.  For example, one may define"
     "the weight of a derivative to be simply its order"
     "(this is the default assignment)."
     "This weight function can then be extended to the set of"
     "all differential polynomials, providing a graded ring structure."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialVariableCategory|
  (progn
    (push '|DifferentialVariableCategory| *Categories*)
    (make-instance '|DifferentialVariableCategoryType|)))
```

———-

1.37.7 ExpressionSpace

— defclass ExpressionSpaceType —

```
(defclass |ExpressionSpaceType| (|RetractableToType| |EvalableType| |OrderedSetType|)
  ((parents :initform '(|RetractableTo| |Evalable| |OrderedSet|))
   (name :initform "ExpressionSpace")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'ES)
   (comment :initform (list
     "An expression space is a set which is closed under certain operators"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpace|
  (progn
    (push '|ExpressionSpace| *Categories*)
    (make-instance '|ExpressionSpaceType|)))
```

———-

1.37.8 FullyPatternMatchable

— defclass FullyPatternMatchableType —

```
(defclass |FullyPatternMatchableType| (|TypeType| |PatternMatchableType|)
  ((parents :initform '(|Type| |PatternMatchable|))
   (name :initform "FullyPatternMatchable")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'FPATMAB)
   (comment :initform (list
     "A set S is PatternMatchable over R if S can lift the"
     "pattern-matching functions of S over the integers and float"
     "to itself (necessary for matching in towers)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullyPatternMatchable|
  (progn
    (push '|FullyPatternMatchable| *Categories*)
    (make-instance '|FullyPatternMatchableType|)))
```

———-

1.37.9 GradedAlgebra

— defclass GradedAlgebraType —

```
(defclass |GradedAlgebraType| (|RetractableToType| |GradedModuleType|)
  ((parents :initform '(|RetractableTo| |GradedModule|))
   (name :initform "GradedAlgebra")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'GRALG)
   (comment :initform (list
     "GradedAlgebra(R,E) denotes 'E-graded R-algebra'."
     "A graded algebra is a graded module together with a degree preserving"
     "R-linear map, called the product."
     " "
     "The name 'product' is written out in full so inner and outer products"
     "with the same mapping type can be distinguished by name."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GradedAlgebra|
  (progn
    (push '|GradedAlgebra| *Categories*)
    (make-instance '|GradedAlgebraType|)))
```

———-

1.37.10 IndexedAggregate

— defclass IndexedAggregateType —

```
(defclass |IndexedAggregateType| (|EltableAggregateType| |HomogeneousAggregateType|)
  ((parents  :initform '(|EltableAggregate| |HomogeneousAggregate|))
   (name :initform "IndexedAggregate")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'IXAGG)
   (comment :initform (list
     "An indexed aggregate is a many-to-one mapping of indices to entries."
     "For example, a one-dimensional-array is an indexed aggregate where"
     "the index is an integer.  Also, a table is an indexed aggregate"
     "where the indices and entries may have any type."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedAggregate|
  (progn
    (push '|IndexedAggregate| *Categories*)
    (make-instance '|IndexedAggregateType|)))
```

———-

1.37.11 InfinitlyClosePointCategory

— defclass InfinitlyClosePointCategoryType —

```
(defclass |InfinitlyClosePointCategoryType| (|SetCategoryWithDegreeType|)
  ((parents :initform '(|SetCategoryWithDegree|))
   (name :initform "InfinitlyClosePointCategory")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'INFCLCT)
   (comment :initform (list
     "This category is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfinitlyClosePointCategory|
  (progn
    (push '|InfinitlyClosePointCategory| *Categories*)
    (make-instance '|InfinitlyClosePointCategoryType|)))
```

———-

1.37.12 MonadWithUnit

— defclass MonadWithUnitType —

```
(defclass |MonadWithUnitType| (|MonadType|)
  ((parents :initform '(|Monad|))
   (name :initform "MonadWithUnit")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'MONADWU)
   (comment :initform (list
     "MonadWithUnit is the class of multiplicative monads with unit,"
     "that is, sets with a binary operation and a unit element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonadWithUnit|
  (progn
    (push '|MonadWithUnit| *Categories*)
    (make-instance '|MonadWithUnitType|)))
```

———-

1.37.13 Monoid

— defclass MonoidType —

```
(defclass |MonoidType| (|SemiGroupType|)
  ((parents :initform '(|SemiGroup|))
   (name :initform "Monoid")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'MONOID)
   (comment :initform (list
     "The class of multiplicative monoids, that is, semigroups with a"
     "multiplicative identity element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Monoid|
  (progn
    (push '|Monoid| *Categories*)
    (make-instance '|MonoidType|)))
```

———-

1.37.14 OrderedAbelianSemiGroup

— defclass —OrderedAbelianSemiGroupType —

```
(defclass |OrderedAbelianSemiGroupType| (|AbelianSemiGroupType| |OrderedSetType|)
  ((parents :initform '(|AbelianSemiGroup| |OrderedSet|))
   (name :initform "OrderedAbelianSemiGroup")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'OASGP)
   (comment :initform (list
     "Ordered sets which are also abelian semigroups, such that the addition"
     "preserves the ordering."
     " "
     "Axiom x < y => x+z < y+z"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianSemiGroup|
  (progn
    (push '|OrderedAbelianSemiGroup| *Categories*)
    (make-instance '|OrderedAbelianSemiGroupType|)))
```

———-

1.37.15 OrderedFinite

— defclass OrderedFiniteType —

```
(defclass |OrderedFiniteType| (|OrderedSetType| |FiniteType|)
  ((parents :initform '(|OrderedSet| |Finite|))
   (name :initform "OrderedFinite")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'ORDFIN)
   (comment :initform (list
     "Ordered finite sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedFinite|
  (progn
    (push '|OrderedFinite| *Categories*)
    (make-instance '|OrderedFiniteType|)))
```

———-

1.37.16 PlacesCategory

— defclass PlacesCategoryType —

```
(defclass |PlacesCategoryType| (|SetCategoryWithDegreeType|)
  ((parents :initform '(|SetCategoryWithDegree|))
   (name :initform "PlacesCategory")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'PLACESC)
   (comment :initform (list
     "This is part of the PAFF package, related to projective space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlacesCategory|
  (progn
    (push '|PlacesCategory| *Categories*)
    (make-instance '|PlacesCategoryType|)))
```

———-

1.37.17 ProjectiveSpaceCategory

— defclass ProjectiveSpaceCategoryType —

```
(defclass |ProjectiveSpaceCategoryType| (|SetCategoryWithDegreeType|)
  ((parents :initform '(|SetCategoryWithDegree|))
   (name :initform "ProjectiveSpaceCategory")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'PRSPCAT)
   (comment :initform (list
     "This is part of the PAFF package, related to projective space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectiveSpaceCategory|
  (progn
    (push '|ProjectiveSpaceCategory| *Categories*)
    (make-instance '|ProjectiveSpaceCategoryType|)))
```

———-

1.37.18 RecursiveAggregate

— defclass RecursiveAggregateType —

```
(defclass |RecursiveAggregateType| (|HomogeneousAggregateType|)
  ((parents :initform '(|HomogeneousAggregate|))
   (name :initform "RecursiveAggregate")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'RCAGG)
   (comment :initform (list
     "A recursive aggregate over a type S is a model for a"
     "a directed graph containing values of type S."
     "Recursively, a recursive aggregate is a node"
     "consisting of a value from S and 0 or more children"
     "which are recursive aggregates."
     "A node with no children is called a leaf node."
     "A recursive aggregate may be cyclic for which some operations as noted"
     "may go into an infinite loop."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RecursiveAggregate|
  (progn
    (push '|RecursiveAggregate| *Categories*)
    (make-instance '|RecursiveAggregateType|)))
```

———-

1.37.19 TwoDimensionalArrayCategory

— defclass TwoDimensionalArrayCategoryType —

```
(defclass |TwoDimensionalArrayCategoryType| (|HomogeneousAggregateType|)
  ((parents :initform '(|HomogeneousAggregate|))
   (name :initform "TwoDimensionalArrayCategory")
   (marker :initform 'category)
   (level :initform 4)
   (abbreviation :initform 'ARR2CAT)
   (comment :initform (list
     "Two dimensional array categories and domains"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoDimensionalArrayCategory|
  (progn
    (push '|TwoDimensionalArrayCategory| *Categories*)
    (make-instance '|TwoDimensionalArrayCategoryType|)))
```

———-

1.38 Level 5

— defvar level5 —

```
(defvar level5
 '(|BinaryRecursiveAggregate| |CancellationAbelianMonoid| |DesingTreeCategory|
   |DoublyLinkedAggregate| |Group| |LinearAggregate| |MatrixCategory|
   |OrderedAbelianMonoid| |OrderedMonoid| |PolynomialSetCategory|
   |PriorityQueueAggregate| |QueueAggregate| |SetAggregate| |StackAggregate|
   |UnaryRecursiveAggregate|))
```

———-

1.38.1 BinaryRecursiveAggregate

— defclass BinaryRecursiveAggregateType —

```
(defclass |BinaryRecursiveAggregateType| (|RecursiveAggregateType|)
  ((parents :initform '(|RecursiveAggregate|))
   (name :initform "BinaryRecursiveAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'BRAGG)
   (comment :initform (list
     "A binary-recursive aggregate has 0, 1 or 2 children and serves"
     "as a model for a binary tree or a doubly-linked aggregate structure"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryRecursiveAggregate|
  (progn
    (push '|BinaryRecursiveAggregate| *Categories*)
    (make-instance '|BinaryRecursiveAggregateType|)))
```

———-

1.38.2 CancellationAbelianMonoid

— defclass CancellationAbelianMonoidType —

```
(defclass |CancellationAbelianMonoidType| (|AbelianMonoidType|)
  ((parents :initform '(|AbelianMonoid|))
   (name :initform "CancellationAbelianMonoid")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'CABMON)
   (comment :initform (list
     "This is an AbelianMonoid with the cancellation property,"
     "   a+b = a+c => b=c "
     "This is formalised by the partial subtraction operator,"
     "which satisfies the  Axioms"
     "   c = a+b <=> c-b = a"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CancellationAbelianMonoid|
  (progn
    (push '|CancellationAbelianMonoid| *Categories*)
    (make-instance '|CancellationAbelianMonoidType|)))
```

———-

1.38.3 DesingTreeCategory

— defclass DesingTreeCategoryType —

```
(defclass |DesingTreeCategoryType| (|RecursiveAggregateType|)
  ((parents :initform '(|RecursiveAggregate|))
   (name :initform "DesingTreeCategory")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'DSTRCAT)
   (comment :initform (list
     "This category is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DesingTreeCategory|
  (progn
    (push '|DesingTreeCategory| *Categories*)
    (make-instance '|DesingTreeCategoryType|)))
```

———-

1.38.4 DoublyLinkedAggregate

— defclass DoublyLinkedAggregateType —

```
(defclass |DoublyLinkedAggregateType| (|RecursiveAggregateType|)
  ((parents :initform '(|RecursiveAggregate|))
   (name :initform "DoublyLinkedAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'DLAGG)
   (comment :initform (list
     "A doubly-linked aggregate serves as a model for a doubly-linked"
     "list, that is, a list which can has links to both next and previous"
     "nodes and thus can be efficiently traversed in both directions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoublyLinkedAggregate|
  (progn
    (push '|DoublyLinkedAggregate| *Categories*)
    (make-instance '|DoublyLinkedAggregateType|)))
```

——————-

1.38.5 Group

Voevodsky's [Gray18a] definition of a group:

Let U be a *universe*

A *group* in U is a sequence $(G, e, i, m, \lambda, \rho, \lambda', \rho', \alpha, \iota)$, where

- G is a type of U
- $e : G$
- $i : G \to G$
- $m : G \times G \to G$
- λ is a proof that for every $a : G, m(e, a) = a$
- ρ is a proof that for every $a : G, m(a, e) = a$
- λ' is a proof that for every $a : G, m(i(a), a) = e$
- ρ' is a proof that for every $a : G, m(a, i(a)) = e$
- α is a proof that for every $a, b, c : G : m(m(a, b), c) = m(a, m(b, c))$
- ι is a proof that G is a set

— defclass GroupType —

```
(defclass |GroupType| (|MonoidType|)
  ((parents :initform '(|Monoid|))
   (name :initform "Group")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'GROUP)
   (comment :initform (list
     "The class of multiplicative groups, that is, monoids with"
     "multiplicative inverses."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Group|
  (progn
    (push '|Group| *Categories*)
    (make-instance '|GroupType|)))
```

———-

1.38.6 LinearAggregate

— defclass LinearAggregateType —

```
(defclass |LinearAggregateType| (|CollectionType| |IndexedAggregateType|)
  ((parents :initform '(|Collection| |IndexedAggregate|))
   (name :initform "LinearAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'LNAGG)
   (comment :initform (list
     "A linear aggregate is an aggregate whose elements are indexed by integers."
     "Examples of linear aggregates are strings, lists, and"
     "arrays."
     "Most of the exported operations for linear aggregates are non-destructive"
     "but are not always efficient for a particular aggregate."
     "For example, concat of two lists needs only to copy its first"
     "argument, whereas concat of two arrays needs to copy both"
     "arguments. Most of the operations exported here apply to infinite"
     "objects (for example, streams) as well to finite ones."
     "For finite linear aggregates, see FiniteLinearAggregate."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearAggregate|
  (progn
    (push '|LinearAggregate| *Categories*)
    (make-instance '|LinearAggregateType|)))
```

———-

1.38.7 MatrixCategory

— defclass MatrixCategoryType —

```
(defclass |MatrixCategoryType| (|TwoDimensionalArrayCategoryType|)
  ((parents :initform '(|TwoDimensionalArrayCategory|))
   (name :initform "MatrixCategory")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'MATCAT)
   (comment :initform (list
     "MatrixCategory is a general matrix category which allows"
     "different representations and indexing schemes.  Rows and"
     "columns may be extracted with rows returned as objects of"
     "type Row and colums returned as objects of type Col."
     "A domain belonging to this category will be shallowly mutable."
     "The index of the 'first' row may be obtained by calling the"
     "function minRowIndex.  The index of the 'first' column may"
     "be obtained by calling the function minColIndex.  The index of"
     "the first element of a Row is the same as the index of the"
     "first column in a matrix and vice versa."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixCategory|
  (progn
    (push '|MatrixCategory| *Categories*)
    (make-instance '|MatrixCategoryType|)))
```

———-

1.38.8 OrderedAbelianMonoid

— defclass OrderedAbelianMonoidType —

```
(defclass |OrderedAbelianMonoidType| (|AbelianMonoidType| |OrderedAbelianSemiGroupType|)
  ((parents :initform '(|AbelianMonoid| |OrderedAbelianSemiGroup|))
   (name :initform "OrderedAbelianMonoid")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'OAMON)
   (comment :initform (list
     "Ordered sets which are also abelian monoids, such that the addition"
     "preserves the ordering."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianMonoid|
  (progn
    (push '|OrderedAbelianMonoid| *Categories*)
    (make-instance '|OrderedAbelianMonoidType|)))
```

———-

1.38.9 OrderedMonoid

— defclass OrderedMonoidType —

```
(defclass |OrderedMonoidType| (|OrderedSetType| |MonoidType|)
  ((parents :initform '(|OrderedSet| |Monoid|))
   (name :initform "OrderedMonoid")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'ORDMON)
   (comment :initform (list
     "Ordered sets which are also monoids, such that multiplication"
     "preserves the ordering."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedMonoid|
  (progn
    (push '|OrderedMonoid| *Categories*)
    (make-instance '|OrderedMonoidType|)))
```

———-

1.38.10 PolynomialSetCategory

— defclass PolynomialSetCategoryType —

```
(defclass |PolynomialSetCategoryType| (|CollectionType|)
  ((parents :initform '(|Collection|))
   (name :initform "PolynomialSetCategory")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'PSETCAT)
   (comment :initform (list
     "A category for finite subsets of a polynomial ring."
     "Such a set is only regarded as a set of polynomials and not"
     "identified to the ideal it generates. So two distinct sets may"
     "generate the same the ideal. Furthermore, for R being an"
     "integral domain, a set of polynomials may be viewed as a representation"
     "of the ideal it generates in the polynomial ring R)^(-1) P,"
     "or the set of its zeros (described for instance by the radical of the"
     "previous ideal, or a split of the associated affine variety) and so on."
     "So this category provides operations about those different notions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialSetCategory|
  (progn
    (push '|PolynomialSetCategory| *Categories*)
    (make-instance '|PolynomialSetCategoryType|)))
```

———-

1.38.11 PriorityQueueAggregate

— defclass PriorityQueueAggregateType —

```
(defclass |PriorityQueueAggregateType| (|BagAggregateType|)
  ((parents :initform '(|BagAggregate|))
   (name :initform "PriorityQueueAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'PRQAGG)
   (comment :initform (list
     "A priority queue is a bag of items from an ordered set where the item"
     "extracted is always the maximum element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PriorityQueueAggregate|
  (progn
    (push '|PriorityQueueAggregate| *Categories*)
    (make-instance '|PriorityQueueAggregateType|)))
```

——–

1.38.12 QueueAggregate

— defclass QueueAggregateType —

```
(defclass |QueueAggregateType| (|BagAggregateType|)
  ((parents :initform '(|BagAggregate|))
   (name :initform "QueueAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'QUAGG)
   (comment :initform (list
     "A queue is a bag where the first item inserted is the first"
     "item extracted."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QueueAggregate|
  (progn
    (push '|QueueAggregate| *Categories*)
    (make-instance '|QueueAggregateType|)))
```

——–

1.38.13 SetAggregate

— defclass SetAggregateType —

```
(defclass |SetAggregateType| (|CollectionType|)
  ((parents :initform '(|Collection|))
   (name :initform "SetAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'SETAGG)
   (comment :initform (list
     "A set category lists a collection of set-theoretic operations"
     "useful for both finite sets and multisets."
     "Note however that finite sets are distinct from multisets."
     "Although the operations defined for set categories are"
     "common to both, the relationship between the two cannot"
     "be described by inclusion or inheritance."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SetAggregate|
  (progn
    (push '|SetAggregate| *Categories*)
    (make-instance '|SetAggregateType|)))
```

———-

1.38.14 StackAggregate

— defclass StackAggregateType —

```
(defclass |StackAggregateType| (|BagAggregateType|)
  ((parents :initform '(|BagAggregate|))
   (name :initform "StackAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'SKAGG)
   (comment :initform (list
     "A stack is a bag where the last item inserted is the first item extracted."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StackAggregate|
  (progn
    (push '|StackAggregate| *Categories*)
    (make-instance '|StackAggregateType|)))
```

———-

1.38.15 UnaryRecursiveAggregate

— defclass UnaryRecursiveAggregateType —

```
(defclass |UnaryRecursiveAggregateType| (|RecursiveAggregateType|)
  ((parents :initform '(|RecursiveAggregate|))
   (name :initform "UnaryRecursiveAggregate")
   (marker :initform 'category)
   (level :initform 5)
   (abbreviation :initform 'URAGG)
   (comment :initform (list
     "A unary-recursive aggregate is a one where nodes may have either"
     "0 or 1 children."
     "This aggregate models, though not precisely, a linked"
     "list possibly with a single cycle."
     "A node with one children models a non-empty list, with the"
     "value of the list designating the head, or first,"
     "of the list, and the child designating the tail, or rest,"
     "of the list. A node with no child then designates the empty list."
     "Since these aggregates are recursive aggregates, they may be cyclic."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnaryRecursiveAggregate|
  (progn
    (push '|UnaryRecursiveAggregate| *Categories*)
    (make-instance '|UnaryRecursiveAggregateType|)))
```

————-

1.39 Level 6

— defvar level6 —

```
(defvar level6
 '(|AbelianGroup| |BinaryTreeCategory| |DequeueAggregate| |DictionaryOperations|
   |ExtensibleLinearAggregate| |FiniteLinearAggregate|
   |FreeAbelianMonoidCategory|
   |OrderedCancellationAbelianMonoid| |PermutationCategory| |StreamAggregate|
   |TriangularSetCategory|))
```

————-

1.39.1 AbelianGroup

— defclass AbelianGroupType —

```
(defclass |AbelianGroupType| (|CancellationAbelianMonoidType|)
  ((parents :initform '(|CancellationAbelianMonoid|))
   (name :initform "AbelianGroup")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'ABELGRP)
   (comment :initform (list
     "The class of abelian groups, additive monoids where"
     "each element has an additive inverse."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AbelianGroup|
  (progn
    (push '|AbelianGroup| *Categories*)
    (make-instance '|AbelianGroupType|)))
```

———-

1.39.2 BinaryTreeCategory

— defclass BinaryTreeCategoryType —

```
(defclass |BinaryTreeCategoryType| (|BinaryRecursiveAggregateType|)
  ((parents :initform '(|BinaryRecursiveAggregate|))
   (name :initform "BinaryTreeCategory")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'BTCAT)
   (comment :initform (list
     "BinaryTreeCategory(S) is the category of"
     "binary trees: a tree which is either empty or else is a"
     "node consisting of a value and a left and"
     "right, both binary trees. "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryTreeCategory|
  (progn
    (push '|BinaryTreeCategory| *Categories*)
    (make-instance '|BinaryTreeCategoryType|)))
```

———-

1.39.3 DequeueAggregate

— defclass DequeueAggregateType —

```
(defclass |DequeueAggregateType| (|QueueAggregateType| |StackAggregateType|)
  ((parents :initform '(|QueueAggregate| |StackAggregate|))
   (name :initform "DequeueAggregate")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'DQAGG)
   (comment :initform (list
     "A dequeue is a doubly ended stack, that is, a bag where first items"
     "inserted are the first items extracted, at either the front or"
     "the back end of the data structure."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DequeueAggregate|
  (progn
    (push '|DequeueAggregate| *Categories*)
    (make-instance '|DequeueAggregateType|)))
```

——————-

1.39.4 DictionaryOperations

— defclass DictionaryOperationsType —

```
(defclass |DictionaryOperationsType| (|BagAggregateType| |CollectionType|)
  ((parents :initform '(|BagAggregate| |Collection|))
   (name :initform "DictionaryOperations")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'DIOPS)
   (comment :initform (list
     "This category is a collection of operations common to both"
     "categories Dictionary and MultiDictionary"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DictionaryOperations|
  (progn
    (push '|DictionaryOperations| *Categories*)
    (make-instance '|DictionaryOperationsType|)))
```

——————-

1.39.5 ExtensibleLinearAggregate

— defclass ExtensibleLinearAggregateType —

```
(defclass |ExtensibleLinearAggregateType| (|LinearAggregateType|)
  ((parents :initform '(|LinearAggregate|))
   (name :initform "ExtensibleLinearAggregate")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'ELAGG)
   (comment :initform (list
     "An extensible aggregate is one which allows insertion and deletion of"
     "entries. These aggregates are models of lists and streams which are"
     "represented by linked structures so as to make insertion, deletion, and"
     "concatenation efficient. However, access to elements of these"
     "extensible aggregates is generally slow since access is made from the end."
     "See FlexibleArray for an exception."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExtensibleLinearAggregate|
  (progn
    (push '|ExtensibleLinearAggregate| *Categories*)
    (make-instance '|ExtensibleLinearAggregateType|)))
```

———-

1.39.6 FiniteLinearAggregate

— defclass FiniteLinearAggregateType —

```
(defclass |FiniteLinearAggregateType| (|OrderedSetType| |LinearAggregateType|)
  ((parents :initform '(|OrderedSet| |LinearAggregate|))
   (name :initform "FiniteLinearAggregate")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'FLAGG)
   (comment :initform (list
     "A finite linear aggregate is a linear aggregate of finite length."
     "The finite property of the aggregate adds several exports to the"
     "list of exports from LinearAggregate such as"
     "reverse, sort, and so on."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteLinearAggregate|
  (progn
    (push '|FiniteLinearAggregate| *Categories*)
    (make-instance '|FiniteLinearAggregateType|)))
```

———-

1.39.7 FreeAbelianMonoidCategory

— defclass FreeAbelianMonoidCategoryType —

```
(defclass |FreeAbelianMonoidCategoryType| (|RetractableToType| |CancellationAbelianMonoidType|)
  ((parents :initform '(|RetractableTo| |CancellationAbelianMonoid|))
   (name :initform "FreeAbelianMonoidCategory")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'FAMONC)
   (comment :initform (list
     "A free abelian monoid on a set S is the monoid of finite sums of"
     "the form reduce(+,[ni * si]) where the si's are in S, and the ni's"
     "are in a given abelian monoid. The operation is commutative."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeAbelianMonoidCategory|
  (progn
    (push '|FreeAbelianMonoidCategory| *Categories*)
    (make-instance '|FreeAbelianMonoidCategoryType|)))
```

———-

1.39.8 OrderedCancellationAbelianMonoid

— defclass OrderedCancellationAbelianMonoidType —

```
(defclass |OrderedCancellationAbelianMonoidType| (|CancellationAbelianMonoidType|
                                                  |OrderedAbelianMonoidType|)
  ((parents :initform '(|CancellationAbelianMonoid|
                        |OrderedAbelianMonoid|))
   (name :initform "OrderedCancellationAbelianMonoid")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'OCAMON)
   (comment :initform (list
     "Ordered sets which are also abelian cancellation monoids,"
     "such that the addition preserves the ordering."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedCancellationAbelianMonoid|
  (progn
    (push '|OrderedCancellationAbelianMonoid| *Categories*)
    (make-instance '|OrderedCancellationAbelianMonoidType|)))
```

———-

1.39.9 PermutationCategory

— defclass PermutationCategoryType —

```
(defclass |PermutationCategoryType| (|OrderedSetType| |GroupType|)
  ((parents :initform '(|OrderedSet| |Group|))
   (name :initform "PermutationCategory")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'PERMCAT)
   (comment :initform (list
     "PermutationCategory provides a categorial environment"
     "for subgroups of bijections of a set (that is, permutations)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PermutationCategory|
  (progn
    (push '|PermutationCategory| *Categories*)
    (make-instance '|PermutationCategoryType|)))
```

———-

1.39.10 StreamAggregate

— defclass StreamAggregateType —

```
(defclass |StreamAggregateType| (|LinearAggregateType| |UnaryRecursiveAggregateType|)
  ((parents :initform '(|LinearAggregate| |UnaryRecursiveAggregate|))
   (name :initform "StreamAggregate")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'STAGG)
   (comment :initform (list
     "A stream aggregate is a linear aggregate which possibly has an infinite"
     "number of elements. A basic domain constructor which builds stream"
     "aggregates is Stream. From streams, a number of infinite"
     "structures such power series can be built. A stream aggregate may"
     "also be infinite since it may be cyclic."
     "For example, see DecimalExpansion."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamAggregate|
  (progn
    (push '|StreamAggregate| *Categories*)
    (make-instance '|StreamAggregateType|)))
```

———-

1.39.11 TriangularSetCategory

— defclass TriangularSetCategoryType —

```
(defclass |TriangularSetCategoryType| (|PolynomialSetCategoryType|)
  ((parents :initform '(|PolynomialSetCategory|))
   (name :initform "TriangularSetCategory")
   (marker :initform 'category)
   (level :initform 6)
   (abbreviation :initform 'TSETCAT)
   (comment :initform (list
     "The category of triangular sets of multivariate polynomials"
     "with coefficients in an integral domain."
     "Let R be an integral domain and V a finite ordered set of"
     "variables, say X1 < X2 < ... < Xn."
     "A set S of polynomials in R[X1,X2,...,Xn] is triangular"
     "if no elements of S lies in R, and if two distinct"
     "elements of S have distinct main variables."
     "Note that the empty set is a triangular set. A triangular set is not"
     "necessarily a (lexicographical) Groebner basis and the notion of"
     "reduction related to triangular sets is based on the recursive view"
     "of polynomials. We recall this notion here and refer to [1] for more"
     "details."
     "A polynomial P is reduced w.r.t a non-constant polynomial"
     "Q if the degree of P in the main variable of Q"
     "is less than the main degree of Q."
     "A polynomial P is reduced w.r.t a triangular set T"
     "if it is reduced w.r.t. every polynomial of T."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TriangularSetCategory|
  (progn
    (push '|TriangularSetCategory| *Categories*)
    (make-instance '|TriangularSetCategoryType|)))
```

———-

1.40 Level 7

— defvar level7 —

```
(defvar level7
 '(|Dictionary| |FiniteDivisorCategory| |LazyStreamAggregate| |LeftModule|
   |ListAggregate| |MultiDictionary| |MultisetAggregate| |NonAssociativeRng|
   |OneDimensionalArrayAggregate| |OrderedAbelianGroup| |OrderedAbelianMonoidSup|
   |RegularTriangularSetCategory| |RightModule| |Rng|))
```

———-

1.40.1 Dictionary

— defclass DictionaryType —

```
(defclass |DictionaryType| (|DictionaryOperationsType|)
  ((parents :initform '(|DictionaryOperations|))
   (name :initform "Dictionary")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'DIAGG)
   (comment :initform (list
     "A dictionary is an aggregate in which entries can be inserted,"
     "searched for and removed. Duplicates are thrown away on insertion."
     "This category models the usual notion of dictionary which involves"
     "large amounts of data where copying is impractical."
     "Principal operations are thus destructive (non-copying) ones."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Dictionary|
  (progn
    (push '|Dictionary| *Categories*)
    (make-instance '|DictionaryType|)))
```

———-

1.40.2 FiniteDivisorCategory

— defclass FiniteDivisorCategoryType —

```
(defclass |FiniteDivisorCategoryType| (|AbelianGroupType|)
  ((parents :initform '(|AbelianGroup|))
   (name :initform "FiniteDivisorCategory")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'FDIVCAT)
   (comment :initform (list
     "This category describes finite rational divisors on a curve, that"
     "is finite formal sums SUM(n * P) where the n's are integers and the"
     "P's are finite rational points on the curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteDivisorCategory|
  (progn
    (push '|FiniteDivisorCategory| *Categories*)
    (make-instance '|FiniteDivisorCategoryType|)))
```

———-

1.40.3 LazyStreamAggregate

— defclass LazyStreamAggregateType —

```
(defclass |LazyStreamAggregateType| (|StreamAggregateType|)
  ((parents :initform '(|StreamAggregate|))
   (name :initform "LazyStreamAggregate")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'LZSTAGG)
   (comment :initform (list
     "LazyStreamAggregate is the category of streams with lazy"
     "evaluation.  It is understood that the function 'empty?' will"
     "cause lazy evaluation if necessary to determine if there are"
     "entries.  Functions which call 'empty?', for example 'first' and 'rest',"
     "will also cause lazy evaluation if necessary."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LazyStreamAggregate|
  (progn
    (push '|LazyStreamAggregate| *Categories*)
    (make-instance '|LazyStreamAggregateType|)))
```

———-

1.40.4 LeftModule

— defclass LeftModuleType —

```
(defclass |LeftModuleType| (|AbelianGroupType|)
  ((parents :initform '(|AbelianGroup|))
   (name :initform "LeftModule")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'LMODULE)
   (comment :initform (list
     "This is an abelian group which supports left multiplication by elements of"
     "the rng."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |LeftModule|
  (progn
    (push '|LeftModule| *Categories*)
    (make-instance '|LeftModuleType|)))
```

———-

1.40.5 ListAggregate

— defclass ListAggregateType —

```
(defclass |ListAggregateType| (|ExtensibleLinearAggregateType|
                               |FiniteLinearAggregateType|
                               |StreamAggregateType|)
  ((parents :initform '(|ExtensibleLinearAggregate|
                          |FiniteLinearAggregate|
                          |StreamAggregate|))
   (name :initform "ListAggregate")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'LSAGG)
   (comment :initform (list
     "A list aggregate is a model for a linked list data structure."
     "A linked list is a versatile"
     "data structure. Insertion and deletion are efficient and"
     "searching is a linear operation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListAggregate|
  (progn
    (push '|ListAggregate| *Categories*)
    (make-instance '|ListAggregateType|)))
```

———-

1.40.6 MultiDictionary

— defclass MultiDictionaryType —

```
(defclass |MultiDictionaryType| (|DictionaryOperationsType|)
  ((parents :initform '(|DictionaryOperations|))
   (name :initform "MultiDictionary")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'MDAGG)
   (comment :initform (list
     "A multi-dictionary is a dictionary which may contain duplicates."
```

```
    "As for any dictionary, its size is assumed large so that"
    "copying (non-destructive) operations are generally to be avoided."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultiDictionary|
  (progn
    (push '|MultiDictionary| *Categories*)
    (make-instance '|MultiDictionaryType|)))
```

———–

1.40.7 MultisetAggregate

— defclass MultisetAggregateType —

```
(defclass |MultisetAggregateType| (|SetAggregateType| |MultiDictionaryType|)
  ((parents :initform '(|SetAggregate| |MultiDictionary|))
   (name :initform "MultisetAggregate")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'MSETAGG)
   (comment :initform (list
     "A multi-set aggregate is a set which keeps track of the multiplicity"
     "of its elements."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultisetAggregate|
  (progn
    (push '|MultisetAggregate| *Categories*)
    (make-instance '|MultisetAggregateType|)))
```

———–

1.40.8 NonAssociativeRng

— defclass NonAssociativeRngType —

```
(defclass |NonAssociativeRngType| (|MonadType| |AbelianGroupType|)
  ((parents :initform '(|Monad| |AbelianGroup|))
   (name :initform "NonAssociativeRng")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'NARGN)
   (comment :initform (list
     "NonAssociativeRng is a basic ring-type structure, not necessarily"
     "commutative or associative, and not necessarily with unit."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonAssociativeRng|
  (progn
    (push '|NonAssociativeRng| *Categories*)
    (make-instance '|NonAssociativeRngType|)))
```

———-

1.40.9 OneDimensionalArrayAggregate

— defclass OneDimensionalArrayAggregateType —

```
(defclass |OneDimensionalArrayAggregateType| (|FiniteLinearAggregateType|)
  ((parents  :initform '(|FiniteLinearAggregate|))
   (name :initform "OneDimensionalArrayAggregate")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'A1AGG)
   (comment :initform (list
     "One-dimensional-array aggregates serves as models for one-dimensional"
     "arrays. Categorically, these aggregates are finite linear aggregates"
     "with the shallowlyMutable property, that is, any component of"
     "the array may be changed without affecting the"
     "identity of the overall array."
     "Array data structures are typically represented by a fixed area in"
     "storage and cannot efficiently grow or shrink on demand as can list"
     "structures (see however FlexibleArray for a data structure"
     "which is a cross between a list and an array)."
     "Iteration over, and access to, elements of arrays is extremely fast"
     "(and often can be optimized to open-code)."
     "Insertion and deletion however is generally slow since an entirely new"
     "data structure must be created for the result."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OneDimensionalArrayAggregate|
  (progn
    (push '|OneDimensionalArrayAggregate| *Categories*)
    (make-instance '|OneDimensionalArrayAggregateType|)))
```

———-

1.40.10 OrderedAbelianGroup

— defclass OrderedAbelianGroupType —

```
(defclass |OrderedAbelianGroupType| (|AbelianGroupType| |OrderedCancellationAbelianMonoidType|)
  ((parents :initform '(|AbelianGroup| |OrderedCancellationAbelianMonoid|))
   (name :initform "OrderedAbelianGroup")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'OAGROUP)
   (comment :initform (list
     "Ordered sets which are also abelian groups, such that the"
     "addition preserves the ordering."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianGroup|
  (progn
    (push '|OrderedAbelianGroup| *Categories*)
    (make-instance '|OrderedAbelianGroupType|)))
```

———-

1.40.11 OrderedAbelianMonoidSup

— defclass OrderedAbelianMonoidSupType —

```
(defclass |OrderedAbelianMonoidSupType| (|OrderedCancellationAbelianMonoidType|)
  ((parents  :initform '(|OrderedCancellationAbelianMonoid|))
   (name :initform "OrderedAbelianMonoidSup")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'OAMONS)
   (comment :initform (list
     "This domain is an OrderedAbelianMonoid with a sup"
     "operation added. The purpose of the sup operator"
     "in this domain is to act as a supremum with respect to the"
     "partial order imposed by `-`, rather than with respect to"
     "the total > order (since that is 'max')."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianMonoidSup|
  (progn
    (push '|OrderedAbelianMonoidSup| *Categories*)
    (make-instance '|OrderedAbelianMonoidSupType|)))
```

———-

1.40.12 RegularTriangularSetCategory

— defclass RegularTriangularSetCategoryType —

```
(defclass |RegularTriangularSetCategoryType| (|TriangularSetCategoryType|)
  ((parents :initform '(|TriangularSetCategory|))
   (name :initform "RegularTriangularSetCategory")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'RSETCAT)
   (comment :initform (list
     "The category of regular triangular sets, introduced under"
     "the name regular chains in [1] (and other papers)."
     "In [3] it is proved that regular triangular sets and towers of simple"
     "extensions of a field are equivalent notions."
     "In the following definitions, all polynomials and ideals"
     "are taken from the polynomial ring k[x1,...,xn] where k"
     "is the fraction field of R."
     "The triangular set [t1,...,tm] is regular"
     "iff for every id} the initial of ti+1 is invertible"
     "in the tower of simple extensions associated with [t1,...,ti]."
     "A family [T1,...,Ts] of regular triangular sets"
     "is a split of Kalkbrener of a given ideal I"
     "iff the radical of I is equal to the intersection"
     "of the radical ideals generated by the saturated ideals"
     "of the [T1,...,Ti]."
     "A family [T1,...,Ts] of regular triangular sets"
     "is a split of Kalkbrener of a given triangular set T"
     "iff it is a split of Kalkbrener of the saturated ideal of T."
     "Let K be an algebraic closure of k."
     "Assume that V is finite with cardinality"
     "n and let A be the affine space K^n."
     "For a regular triangular set T let denote by W(T) the"
     "set of regular zeros of T."
     "A family [T1,...,Ts] of regular triangular sets"
     "is a split of Lazard of a given subset S of A"
     "iff the union of the W(Ti) contains S and"
     "is contained in the closure of S (w.r.t. Zariski topology)."
     "A family [T1,...,Ts] of regular triangular sets"
     "is a split of Lazard of a given triangular set T"
     "if it is a split of Lazard of W(T)."
     "Note that if [T1,...,Ts] is a split of Lazard of"
     "T then it is also a split of Kalkbrener of T."
     "The converse is false."
     "This category provides operations related to both kinds of"
     "splits, the former being related to ideals decomposition whereas"
     "the latter deals with varieties decomposition."
     "See the example illustrating the RegularTriangularSet constructor for more"
     "explanations about decompositions by means of regular triangular sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularTriangularSetCategory|
  (progn
    (push '|RegularTriangularSetCategory| *Categories*)
    (make-instance '|RegularTriangularSetCategoryType|)))
```

——————-

1.40.13 RightModule

— defclass RightModuleType —

```
(defclass |RightModuleType| (|AbelianGroupType|)
  ((parents :initform '(|AbelianGroup|))
   (name :initform "RightModule")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'RMODULE)
   (comment :initform (list
     "The category of right modules over an rng (ring not necessarily"
     "with unit). This is an abelian group which supports right"
     "multiplication by elements of the rng."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RightModule|
  (progn
    (push '|RightModule| *Categories*)
    (make-instance '|RightModuleType|)))
```

——————-

1.40.14 Rng

— defclass RngType —

```
(defclass |RngType| (|SemiGroupType| |AbelianGroupType|)
  ((parents :initform '(|SemiGroup| |AbelianGroup|))
   (name :initform "Rng")
   (marker :initform 'category)
   (level :initform 7)
   (abbreviation :initform 'RNG)
   (comment :initform (list
     "The category of associative rings, not necessarily commutative, and not"
     "necessarily with a 1. This is a combination of an abelian group"
     "and a semigroup, with multiplication distributing over addition."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Rng|
  (progn
    (push '|Rng| *Categories*)
    (make-instance '|RngType|)))
```

———-

1.41 Level 8

— defvar level8 —

```
(defvar level8
 '(|BiModule| |BitAggregate| |FiniteSetAggregate| |KeyedDictionary|
   |NonAssociativeRing| |NormalizedTriangularSetCategory|
   |OrderedMultisetAggregate| |Ring| |SquareFreeRegularTriangularSetCategory|
   |StringAggregate| |VectorCategory|))
```

———-

1.41.1 BiModule

— defclass BiModuleType —

```
(defclass |BiModuleType| (|LeftModuleType| |RightModuleType|)
  ((parents :initform '(|LeftModule| |RightModule|))
   (name :initform "BiModule")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'BMODULE)
   (comment :initform (list
     "A BiModule is both a left and right module with respect"
     "to potentially different rings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BiModule|
  (progn
    (push '|BiModule| *Categories*)
    (make-instance '|BiModuleType|)))
```

———-

1.41.2 BitAggregate

— defclass —BitAggregateType —

```
(defclass |BitAggregateType| (|LogicType| |OneDimensionalArrayAggregateType|)
  ((parents :initform '(|Logic| |OneDimensionalArrayAggregate|))
   (name :initform "BitAggregate")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'BTAGG)
   (comment :initform (list
     "The bit aggregate category models aggregates representing large"
     "quantities of Boolean data."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BitAggregate|
  (progn
    (push '|BitAggregate| *Categories*)
    (make-instance '|BitAggregateType|)))
```

————-

1.41.3 FiniteSetAggregate

— defclass FiniteSetAggregateType —

```
(defclass |FiniteSetAggregateType| (|FiniteType| |SetAggregateType| |DictionaryType|)
  ((parents :initform '(|Finite| |SetAggregate| |Dictionary|))
   (name :initform "FiniteSetAggregate")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'FSAGG)
   (comment :initform (list
     "A finite-set aggregate models the notion of a finite set, that is,"
     "a collection of elements characterized by membership, but not"
     "by order or multiplicity."
     "See Set for an example."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteSetAggregate|
  (progn
    (push '|FiniteSetAggregate| *Categories*)
    (make-instance '|FiniteSetAggregateType|)))
```

————-

1.41.4 KeyedDictionary

— defclass KeyedDictionaryType —

```
(defclass |KeyedDictionaryType| (|DictionaryType|)
  ((parents :initform '(|Dictionary|))
   (name :initform "KeyedDictionary")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'KDAGG)
   (comment :initform (list
     "A keyed dictionary is a dictionary of key-entry pairs for which there is"
     "a unique entry for each key."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |KeyedDictionary|
  (progn
    (push '|KeyedDictionary| *Categories*)
    (make-instance '|KeyedDictionaryType|)))
```

———-

1.41.5 NonAssociativeRing

— defclass NonAssociativeRingType —

```
(defclass |NonAssociativeRingType| (|MonadWithUnitType| |NonAssociativeRngType|)
  ((parents :initform '(|MonadWithUnit| |NonAssociativeRng|))
   (name :initform "NonAssociativeRing")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'NASRING)
   (comment :initform (list
     "A NonAssociativeRing is a non associative rng which has a unit,"
     "the multiplication is not necessarily commutative or associative."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonAssociativeRing|
  (progn
    (push '|NonAssociativeRing| *Categories*)
    (make-instance '|NonAssociativeRingType|)))
```

———-

1.41.6 NormalizedTriangularSetCategory

— defclass NormalizedTriangularSetCategoryType —

```
(defclass |NormalizedTriangularSetCategoryType| (|RegularTriangularSetCategoryType|)
  ((parents :initform '(|RegularTriangularSetCategory|))
   (name :initform "NormalizedTriangularSetCategory")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'NTSCAT)
   (comment :initform (list
     "The category of normalized triangular sets. A triangular"
     "set ts is said normalized if for every algebraic"
     "variable v of ts the polynomial select(ts,v)"
     "is normalized w.r.t. every polynomial in collectUnder(ts,v)."
     "A polynomial p is said normalized w.r.t. a non-constant"
     "polynomial q if p is constant or degree(p,mdeg(q)) = 0"
     "and init(p) is normalized w.r.t. q. One of the important"
     "features of normalized triangular sets is that they are regular sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormalizedTriangularSetCategory|
  (progn
    (push '|NormalizedTriangularSetCategory| *Categories*)
    (make-instance '|NormalizedTriangularSetCategoryType|)))
```

——————-

1.41.7 OrderedMultisetAggregate

— defclass OrderedMultisetAggregateType —

```
(defclass |OrderedMultisetAggregateType| (|MultisetAggregateType| |PriorityQueueAggregateType|)
  ((parents :initform '(|MultisetAggregate| |PriorityQueueAggregate|))
   (name :initform "OrderedMultisetAggregate")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'OMSAGG)
   (comment :initform (list
     "An ordered-multiset aggregate is a multiset built over an ordered set S"
     "so that the relative sizes of its entries can be assessed."
     "These aggregates serve as models for priority queues."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedMultisetAggregate|
  (progn
    (push '|OrderedMultisetAggregate| *Categories*)
    (make-instance '|OrderedMultisetAggregateType|)))
```

——————-

1.41.8 Ring

— defclass RingType —

```
(defclass |RingType| (|MonoidType| |LeftModuleType| |RngType|)
  ((parents :initform '(|Monoid| |LeftModule| |Rng|))
   (name :initform "Ring")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'RING)
   (comment :initform (list
     "The category of rings with unity, always associative, but"
     "not necessarily commutative."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Ring|
  (progn
    (push '|Ring| *Categories*)
    (make-instance '|RingType|)))
```

———-

1.41.9 SquareFreeRegularTriangularSetCategory

— defclass SquareFreeRegularTriangularSetCategoryType —

```
(defclass |SquareFreeRegularTriangularSetCategoryType| (|RegularTriangularSetCategoryType|)
  ((parents :initform '(|RegularTriangularSetCategory|))
   (name :initform "SquareFreeRegularTriangularSetCategory")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'SFRTCAT)
   (comment :initform (list
     "The category of square-free regular triangular sets."
     "A regular triangular set ts is square-free if"
     "the gcd of any polynomial p in ts and"
     "differentiate(p,mvar(p)) w.r.t. collectUnder(ts,mvar(p))"
     "has degree zero w.r.t. mvar(p). Thus any square-free regular"
     "set defines a tower of square-free simple extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularTriangularSetCategory|
  (progn
    (push '|SquareFreeRegularTriangularSetCategory| *Categories*)
    (make-instance '|SquareFreeRegularTriangularSetCategoryType|)))
```

———-

1.41.10 StringAggregate

— defclass StringAggregateType —

```
(defclass |StringAggregateType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "StringAggregate")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'SRAGG)
   (comment :initform (list
     "A string aggregate is a category for strings, that is,"
     "one dimensional arrays of characters."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StringAggregate|
  (progn
    (push '|StringAggregate| *Categories*)
    (make-instance '|StringAggregateType|)))
```

———-

1.41.11 VectorCategory

— defclass —VectorCategoryType —

```
(defclass |VectorCategoryType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "VectorCategory")
   (marker :initform 'category)
   (level :initform 8)
   (abbreviation :initform 'VECTCAT)
   (comment :initform (list
     "VectorCategory represents the type of vector like objects,"
     "that is, finite sequences indexed by some finite segment of the"
     "integers. The operations available on vectors depend on the structure"
     "of the underlying components. Many operations from the component domain"
     "are defined for vectors componentwise. It can by assumed that extraction or"
     "updating components can be done in constant time."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |VectorCategory|
  (progn
    (push '|VectorCategory| *Categories*)
    (make-instance '|VectorCategoryType|)))
```

———-

1.42 Level 9

— defvar level9 —

```
(defvar level9
 '(|CharacteristicNonZero| |CharacteristicZero| |CommutativeRing|
   |DifferentialRing| |EntireRing| |LeftAlgebra| |LinearlyExplicitRingOver|
   |Module| |OrderedRing| |PartialDifferentialRing| |PointCategory|
   |SquareFreeNormalizedTriangularSetCategory| |StringCategory| |TableAggregate|))
```

———-

1.42.1 CharacteristicNonZero

— defclass CharacteristicNonZeroType —

```
(defclass |CharacteristicNonZeroType| (|RingType|)
  ((parents :initform '(|Ring|))
   (name :initform "CharacteristicNonZero")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'CHARNZ)
   (comment :initform (list
     "Rings of Characteristic Non Zero"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicNonZero|
  (progn
    (push '|CharacteristicNonZero| *Categories*)
    (make-instance '|CharacteristicNonZeroType|)))
```

———-

1.42.2 CharacteristicZero

— defclass CharacteristicZeroType —

```
(defclass |CharacteristicZeroType| (|RingType|)
  ((parents :initform '(|Ring|))
   (name :initform "CharacteristicZero")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'CHARZ)
   (comment :initform (list
     "Rings of Characteristic Zero."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicZero|
  (progn
    (push '|CharacteristicZero| *Categories*)
    (make-instance '|CharacteristicZeroType|)))
```

———–

1.42.3 CommutativeRing

— defclass CommutativeRingType —

```
(defclass |CommutativeRingType| (|RingType| |BiModuleType|)
  ((parents :initform '(|Ring| |BiModule|))
   (name :initform "CommutativeRing")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'COMRING)
   (comment :initform (list
     "The category of commutative rings with unity, rings where"
     "* is commutative, and which have a multiplicative identity"
     "element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommutativeRing|
  (progn
    (push '|CommutativeRing| *Categories*)
    (make-instance '|CommutativeRingType|)))
```

———–

1.42.4 DifferentialRing

— defclass DifferentialRingType —

```
(defclass |DifferentialRingType| (|RingType|)
  ((parents :initform '(|Ring|))
   (name :initform "DifferentialRing")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'DIFRING)
   (comment :initform (list
     "An ordinary differential ring, that is, a ring with an operation"
     "differentiate."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialRing|
  (progn
    (push '|DifferentialRing| *Categories*)
    (make-instance '|DifferentialRingType|)))
```

———-

1.42.5 EntireRing

— defclass EntireRingType —

```
(defclass |EntireRingType| (|RingType| |BiModuleType|)
  ((parents :initform '(|Ring| |BiModule|))
   (name :initform "EntireRing")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'ENTIRER)
   (comment :initform (list
     "Entire Rings (non-commutative Integral Domains), a ring"
     "not necessarily commutative which has no zero divisors."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EntireRing|
  (progn
    (push '|EntireRing| *Categories*)
    (make-instance '|EntireRingType|)))
```

———-

1.42.6 LeftAlgebra

— defclass LeftAlgebraType —

```
(defclass |LeftAlgebraType| (|RingType|)
  ((parents :initform '(|Ring|))
   (name :initform "LeftAlgebra")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'LALG)
   (comment :initform (list
     "The category of all left algebras over an arbitrary ring."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeftAlgebra|
  (progn
    (push '|LeftAlgebra| *Categories*)
    (make-instance '|LeftAlgebraType|)))
```

———-

1.42.7 LinearlyExplicitRingOver

— defclass LinearlyExplicitRingOverType —

```
(defclass |LinearlyExplicitRingOverType| (|RingType|)
  ((parents :initform '(|Ring|))
   (name :initform "LinearlyExplicitRingOver")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'LINEXP)
   (comment :initform (list
     "An extension ring with an explicit linear dependence test."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearlyExplicitRingOver|
  (progn
    (push '|LinearlyExplicitRingOver| *Categories*)
    (make-instance '|LinearlyExplicitRingOverType|)))
```

———-

1.42.8 Module

— defclass ModuleType —

```
(defclass |ModuleType| (|BiModuleType|)
  ((parents :initform '(|BiModule|))
   (name :initform "Module")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'MODULE)
   (comment :initform (list
     "The category of modules over a commutative ring."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Module|
  (progn
    (push '|Module| *Categories*)
    (make-instance '|ModuleType|)))
```

———-

1.42.9 OrderedRing

— defclass OrderedRingType —

```
(defclass |OrderedRingType| (|OrderedAbelianGroupType| |RingType|)
  ((parents :initform '(|OrderedAbelianGroup| |Ring|))
   (name :initform "OrderedRing")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'ORDRING)
   (comment :initform (list
     "Ordered sets which are also rings, that is, domains where the ring"
     "operations are compatible with the ordering."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedRing|
  (progn
    (push '|OrderedRing| *Categories*)
    (make-instance '|OrderedRingType|)))
```

———-

1.42.10 PartialDifferentialRing

— defclass PartialDifferentialRingType —

```
(defclass |PartialDifferentialRingType| (|RingType|)
  ((parents :initform '(|Ring|))
   (name :initform "PartialDifferentialRing")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'PDRING)
   (comment :initform (list
     "A partial differential ring with differentiations indexed by a"
     "parameter type S."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialDifferentialRing|
  (progn
    (push '|PartialDifferentialRing| *Categories*)
    (make-instance '|PartialDifferentialRingType|)))
```

———-

1.42.11 PointCategory

— defclass PointCategoryType —

```
(defclass |PointCategoryType| (|VectorCategoryType|)
  ((parents :initform '(|VectorCategory|))
   (name :initform "PointCategory")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'PTCAT)
   (comment :initform (list
     "PointCategory is the category of points in space which"
     "may be plotted via the graphics facilities.  Functions are provided for"
     "defining points and handling elements of points."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointCategory|
  (progn
    (push '|PointCategory| *Categories*)
    (make-instance '|PointCategoryType|)))
```

———-

1.42.12 SquareFreeNormalizedTriangularSetCategory

— defclass SquareFreeNormalizedTriangularSetCategoryType —

```
(defclass |SquareFreeNormalizedTriangularSetCategoryType| (|NormalizedTriangularSetCategoryType|
                                                           |SquareFreeRegularTriangularSetCategoryType|)
  ((parents :initform '(|NormalizedTriangularSetCategory| |SquareFreeRegularTriangularSetCategory|))
   (name :initform "SquareFreeNormalizedTriangularSetCategory")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'SNTSCAT)
   (comment :initform (list
     "The category of square-free and normalized triangular sets."
     "Thus, up to the primitivity axiom of [1], these sets are Lazard"
     "triangular sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeNormalizedTriangularSetCategory|
  (progn
    (push '|SquareFreeNormalizedTriangularSetCategory| *Categories*)
    (make-instance '|SquareFreeNormalizedTriangularSetCategoryType|)))
```

———-

1.42.13 StringCategory

— defclass StringCategoryType —

```
(defclass |StringCategoryType| (|OpenMathType| |StringAggregateType|)
  ((parents :initform '(|OpenMath| |StringAggregate|))
   (name :initform "StringCategory")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'STRICAT)
   (comment :initform (list
     "A category for string-like objects"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StringCategory|
  (progn
    (push '|StringCategory| *Categories*)
    (make-instance '|StringCategoryType|)))
```

———-

1.42.14 TableAggregate

— defclass TableAggregateType —

```
(defclass |TableAggregateType| (|KeyedDictionaryType| |IndexedAggregateType|)
  ((parents :initform '(|KeyedDictionary| |IndexedAggregate|))
   (name :initform "TableAggregate")
   (marker :initform 'category)
   (level :initform 9)
   (abbreviation :initform 'TBAGG)
   (comment :initform (list
     "A table aggregate is a model of a table, that is, a discrete many-to-one"
     "mapping from keys to entries."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TableAggregate|
  (progn
    (push '|TableAggregate| *Categories*)
    (make-instance '|TableAggregateType|)))
```

———-

1.43 Level 10

— defvar level10 —

```
(defvar level10
 '(|Algebra| |AssociationListAggregate| |DifferentialExtension| |DivisorCategory|
   |FreeModuleCat| |FullyLinearlyExplicitRingOver| |LeftOreRing| |LieAlgebra|
   |NonAssociativeAlgebra| |RectangularMatrixCategory| |VectorSpace|))
```

———-

1.43.1 Algebra

— defclass AlgebraType —

```
(defclass |AlgebraType| (|RingType| |ModuleType|)
  ((parents :initform '(|Ring| |Module|))
   (name :initform "Algebra")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'ALGEBRA)
   (comment :initform (list
     "The category of associative algebras (modules which are themselves rings)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Algebra|
  (progn
    (push '|Algebra| *Categories*)
    (make-instance '|AlgebraType|)))
```

———-

1.43.2 AssociationListAggregate

— defclass AssociationListAggregateType —

```
(defclass |AssociationListAggregateType| (|ListAggregateType| |TableAggregateType|)
  ((parents :initform '(|ListAggregate| |TableAggregate|))
   (name :initform "AssociationListAggregate")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'ALAGG)
   (comment :initform (list
     "An association list is a list of key entry pairs which may be viewed"
     "as a table. It is a poor mans version of a table:"
     "searching for a key is a linear operation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociationListAggregate|
  (progn
    (push '|AssociationListAggregate| *Categories*)
    (make-instance '|AssociationListAggregateType|)))
```

———-

1.43.3 DifferentialExtension

— defclass DifferentialExtensionType —

```
(defclass |DifferentialExtensionType| (|DifferentialRingType| |PartialDifferentialRingType|)
  ((parents :initform '(|DifferentialRing| |PartialDifferentialRing|))
   (name :initform "DifferentialExtension")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'DIFEXT)
   (comment :initform (list
     "Differential extensions of a ring R."
     "Given a differentiation on R, extend it to a differentiation on %."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialExtension|
  (progn
    (push '|DifferentialExtension| *Categories*)
    (make-instance '|DifferentialExtensionType|)))
```

———-

1.43.4 DivisorCategory

— defclass DivisorCategoryType —

```
(defclass |DivisorCategoryType| (|FreeAbelianMonoidCategoryType| |ModuleType|)
  ((parents :initform '(|FreeAbelianMonoidCategory| |Module|))
   (name :initform "DivisorCategory")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'DIVCAT)
   (comment :initform (list
     "This category exports the function for domains "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DivisorCategory|
  (progn
    (push '|DivisorCategory| *Categories*)
    (make-instance '|DivisorCategoryType|)))
```

———-

1.43.5 FreeModuleCat

— defclass FreeModuleCatType —

```
(defclass |FreeModuleCatType| (|RetractableToType| |ModuleType|)
  ((parents :initform '(|RetractableTo| |Module|))
   (name :initform "FreeModuleCat")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'FMCAT)
   (comment :initform (list
     "A domain of this category"
     "implements formal linear combinations"
     "of elements from a domain Basis with coefficients"
     "in a domain R. The domain Basis needs only"
     "to belong to the category SetCategory and R"
     "to the category Ring. Thus the coefficient ring"
     "may be non-commutative."
     "See the XDistributedPolynomial constructor"
     "for examples of domains built with the FreeModuleCat"
     "category constructor."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeModuleCat|
  (progn
    (push '|FreeModuleCat| *Categories*)
    (make-instance '|FreeModuleCatType|)))
```

——–-

1.43.6 FullyLinearlyExplicitRingOver

— defclass FullyLinearlyExplicitRingOverType —

```
(defclass |FullyLinearlyExplicitRingOverType| (|LinearlyExplicitRingOverType|)
  ((parents :initform '(|LinearlyExplicitRingOver|))
   (name :initform "FullyLinearlyExplicitRingOver")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'FLINEXP)
   (comment :initform (list
     "S is FullyLinearlyExplicitRingOver R means that S is a"
     "LinearlyExplicitRingOver R and, in addition, if R is a"
     "LinearlyExplicitRingOver Integer, then so is S"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullyLinearlyExplicitRingOver|
  (progn
    (push '|FullyLinearlyExplicitRingOver| *Categories*)
    (make-instance '|FullyLinearlyExplicitRingOverType|)))
```

——–-

1.43.7 LeftOreRing

— defclass LeftOreRingType —

```
(defclass |LeftOreRingType| (|EntireRingType|)
  ((parents :initform '(|EntireRing|))
   (name :initform "LeftOreRing")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'LORER)
   (comment :initform (list
     "This is the category of left ore rings, that is noncommutative"
     "rings without zero divisors where we can compute the least left"
     "common multiple."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeftOreRing|
  (progn
    (push '|LeftOreRing| *Categories*)
    (make-instance '|LeftOreRingType|)))
```

———-

1.43.8 LieAlgebra

— defclass LieAlgebraType —

```
(defclass |LieAlgebraType| (|ModuleType|)
  ((parents :initform '(|Module|))
   (name :initform "LieAlgebra")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'LIECAT)
   (comment :initform (list
     "The category of Lie Algebras."
     "It is used by the domains of non-commutative algebra,"
     "LiePolynomial and XPBWPolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LieAlgebra|
  (progn
    (push '|LieAlgebra| *Categories*)
    (make-instance '|LieAlgebraType|)))
```

———-

1.43.9 NonAssociativeAlgebra

— defclass NonAssociativeAlgebraType —

```
(defclass |NonAssociativeAlgebraType| (|NonAssociativeRngType| |ModuleType|)
  ((parents :initform '(|NonAssociativeRng| |Module|))
   (name :initform "NonAssociativeAlgebra")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'NAALG)
   (comment :initform (list
     "NonAssociativeAlgebra is the category of non associative algebras"
     "(modules which are themselves non associative rngs)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonAssociativeAlgebra|
  (progn
    (push '|NonAssociativeAlgebra| *Categories*)
    (make-instance '|NonAssociativeAlgebraType|)))
```

————-

1.43.10 RectangularMatrixCategory

— defclass RectangularMatrixCategoryType —

```
(defclass |RectangularMatrixCategoryType| (|ModuleType| |HomogeneousAggregateType|)
  ((parents :initform '(|Module| |HomogeneousAggregate|))
   (name :initform "RectangularMatrixCategory")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'RMATCAT)
   (comment :initform (list
     "RectangularMatrixCategory is a category of matrices of fixed"
     "dimensions. The dimensions of the matrix will be parameters of the"
     "domain. Domains in this category will be R-modules and will be non-mutable."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RectangularMatrixCategory|
  (progn
    (push '|RectangularMatrixCategory| *Categories*)
    (make-instance '|RectangularMatrixCategoryType|)))
```

————-

1.43.11 VectorSpace

— defclass VectorSpaceType —

```
(defclass |VectorSpaceType| (|ModuleType|)
  ((parents :initform '(|Module|))
   (name :initform "VectorSpace")
   (marker :initform 'category)
   (level :initform 10)
   (abbreviation :initform 'VSPACE)
   (comment :initform (list
     "Vector Spaces (not necessarily finite dimensional) over a field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |VectorSpace|
  (progn
    (push '|VectorSpace| *Categories*)
    (make-instance '|VectorSpaceType|)))
```

——————-

1.44 Level 11

— defvar level11 —

```
(defvar level11
 '(|DirectProductCategory| |DivisionRing| |FiniteRankAlgebra|
   |FiniteRankNonAssociativeAlgebra| |FreeLieAlgebra| |IntegralDomain|
   |MonogenicLinearOperator| |OctonionCategory| |SquareMatrixCategory|
   |UnivariateSkewPolynomialCategory| |XAlgebra|))
```

——————-

1.44.1 DirectProductCategory

— defclass DirectProductCategoryType —

```
(defclass |DirectProductCategoryType| (|FullyRetractableToType|
                                       |FiniteType|
                                       |IndexedAggregateType|
                                       |OrderedAbelianMonoidSupType|
                                       |CommutativeRingType|
                                       |OrderedRingType|
                                       |AlgebraType|
                                       |DifferentialExtensionType|
                                       |FullyLinearlyExplicitRingOverType|
                                       |VectorSpaceType|)
  ((parents :initform '(|FullyRetractableTo|
                        |Finite|
                        |IndexedAggregate|
                        |OrderedAbelianMonoidSup|
                        |CommutativeRing|
                        |OrderedRing|
                        |Algebra|
                        |DifferentialExtension|
                        |FullyLinearlyExplicitRingOver|
                        |VectorSpace|))
```

```
   (name :initform "DirectProductCategory")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'DIRPCAT)
   (comment :initform (list
     "This category represents a finite cartesian product of a given type."
     "Many categorical properties are preserved under this construction."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductCategory|
  (progn
    (push '|DirectProductCategory| *Categories*)
    (make-instance '|DirectProductCategoryType|)))
```

———-

1.44.2 DivisionRing

— defclass DivisionRingType —

```
(defclass |DivisionRingType| (|AlgebraType| |EntireRingType|)
  ((parents :initform '(|Algebra| |EntireRing|))
   (name :initform "DivisionRing")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'DIVRING)
   (comment :initform (list
     "A division ring (sometimes called a skew field),"
     "a not necessarily commutative ring where"
     "all non-zero elements have multiplicative inverses."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DivisionRing|
  (progn
    (push '|DivisionRing| *Categories*)
    (make-instance '|DivisionRingType|)))
```

———-

1.44.3 FiniteRankAlgebra

— defclass FiniteRankAlgebraType —

```
(defclass |FiniteRankAlgebraType| (|CharacteristicNonZeroType| |CharacteristicZeroType|
                                  |AlgebraType|)
  ((parents :initform '(|CharacteristicNonZero| |CharacteristicZero|
                        |Algebra|))
   (name :initform "FiniteRankAlgebra")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'FINRALG)
   (comment :initform (list
     "A FiniteRankAlgebra is an algebra over a commutative ring R which"
     "is a free R-module of finite rank."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteRankAlgebra|
  (progn
    (push '|FiniteRankAlgebra| *Categories*)
    (make-instance '|FiniteRankAlgebraType|)))
```

———-

1.44.4 FiniteRankNonAssociativeAlgebra

— defclass FiniteRankNonAssociativeAlgebraType —

```
(defclass |FiniteRankNonAssociativeAlgebraType| (|NonAssociativeAlgebraType|)
  ((parents :initform '(|NonAssociativeAlgebra|))
   (name :initform "FiniteRankNonAssociativeAlgebra")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'FINAALG)
   (comment :initform (list
     "A FiniteRankNonAssociativeAlgebra is a non associative algebra over"
     "a commutative ring R which is a free R-module of finite rank."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteRankNonAssociativeAlgebra|
  (progn
    (push '|FiniteRankNonAssociativeAlgebra| *Categories*)
    (make-instance '|FiniteRankNonAssociativeAlgebraType|)))
```

———-

1.44.5 FreeLieAlgebra

— defclass FreeLieAlgebraType —

```
(defclass |FreeLieAlgebraType| (|LieAlgebraType|)
  ((parents :initform '(|LieAlgebra|))
   (name :initform "FreeLieAlgebra")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'FLALG)
   (comment :initform (list
     "The category of free Lie algebras."
     "It is used by domains of non-commutative algebra:"
     "LiePolynomial and XPBWPolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeLieAlgebra|
  (progn
    (push '|FreeLieAlgebra| *Categories*)
    (make-instance '|FreeLieAlgebraType|)))
```

———-

1.44.6 IntegralDomain

— defclass IntegralDomainType —

```
(defclass |IntegralDomainType| (|CommutativeRingType| |AlgebraType| |EntireRingType|)
  ((parents :initform '(|CommutativeRing| |Algebra| |EntireRing|))
   (name :initform "IntegralDomain")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'INTDOM)
   (comment :initform (list
     "The category of commutative integral domains, commutative"
     "rings with no zero divisors."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegralDomain|
  (progn
    (push '|IntegralDomain| *Categories*)
    (make-instance '|IntegralDomainType|)))
```

———-

1.44.7 MonogenicLinearOperator

— defclass MonogenicLinearOperatorType —

```
(defclass |MonogenicLinearOperatorType| (|AlgebraType|)
  ((parents :initform '(|Algebra|))
   (name :initform "MonogenicLinearOperator")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'MLO)
   (comment :initform (list
     "This is the category of linear operator rings with one generator."
     "The generator is not named by the category but can always be"
     "constructed as monomial(1,1)."
     " "
     "For convenience, call the generator G."
     "Then each value is equal to"
     "sum(a(i)*G**i, i = 0..n)"
     "for some unique n and a(i) in R."
     " "
     "Note that multiplication is not necessarily commutative."
     "In fact,  if a is in R, it is quite normal"
     "to have a*G ^= G*a."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonogenicLinearOperator|
  (progn
    (push '|MonogenicLinearOperator| *Categories*)
    (make-instance '|MonogenicLinearOperatorType|)))
```

———–

1.44.8 OctonionCategory

— defclass OctonionCategoryType —

```
(defclass |OctonionCategoryType| (|ConvertibleToType|
                                  |FullyRetractableToType|
                                  |FiniteType|
                                  |FullyEvalableOverType|
                                  |OrderedSetType|
                                  |CharacteristicNonZeroType|
                                  |CharacteristicZeroType|
                                  |AlgebraType|)
  ((parents :initform '(|ConvertibleTo|
                        |FullyRetractableTo|
                        |Finite|
                        |FullyEvalableOver|
                        |OrderedSet|
                        |CharacteristicNonZero|
                        |CharacteristicZero|
                        |Algebra|))
   (name :initform "OctonionCategory")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'OC)
   (comment :initform (list
     "OctonionCategory gives the categorial frame for the"
     "octonions, and eight-dimensional non-associative algebra,"
     "doubling the the quaternions in the same way as doubling"
     "the Complex numbers to get the quaternions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OctonionCategory|
  (progn
    (push '|OctonionCategory| *Categories*)
    (make-instance '|OctonionCategoryType|)))
```

———-

1.44.9 SquareMatrixCategory

— defclass SquareMatrixCategoryType —

```
(defclass |SquareMatrixCategoryType| (|FullyRetractableToType| |AlgebraType| |DifferentialExtensionType|
                                      |FullyLinearlyExplicitRingOverType|
                                      |RectangularMatrixCategoryType|)
  ((parents :initform '(|FullyRetractableTo| |Algebra| |DifferentialExtension|
                        |FullyLinearlyExplicitRingOver|
                        |RectangularMatrixCategory|))
   (name :initform "SquareMatrixCategory")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'SMATCAT)
   (comment :initform (list
     "SquareMatrixCategory is a general square matrix category which"
     "allows different representations and indexing schemes.  Rows and"
     "columns may be extracted with rows returned as objects of"
     "type Row and colums returned as objects of type Col."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareMatrixCategory|
  (progn
    (push '|SquareMatrixCategory| *Categories*)
    (make-instance '|SquareMatrixCategoryType|)))
```

———-

1.44.10 UnivariateSkewPolynomialCategory

— defclass UnivariateSkewPolynomialCategoryType —

```
(defclass |UnivariateSkewPolynomialCategoryType| (|FullyRetractableToType| |AlgebraType|)
  ((parents :initform '(|FullyRetractableTo| |Algebra|))
   (name :initform "UnivariateSkewPolynomialCategory")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'OREPCAT)
   (comment :initform (list
     "This is the category of univariate skew polynomials over an Ore"
     "coefficient ring."
     "The multiplication is given by x a = sigma(a) x + delta a."
     "This category is an evolution of the types"
     "MonogenicLinearOperator, OppositeMonogenicLinearOperator, and"
     "NonCommutativeOperatorDivision"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateSkewPolynomialCategory|
  (progn
    (push '|UnivariateSkewPolynomialCategory| *Categories*)
    (make-instance '|UnivariateSkewPolynomialCategoryType|)))
```

———-

1.44.11 XAlgebra

— defclass XAlgebraType —

```
(defclass |XAlgebraType| (|AlgebraType|)
  ((parents :initform '(|Algebra|))
   (name :initform "XAlgebra")
   (marker :initform 'category)
   (level :initform 11)
   (abbreviation :initform 'XALG)
   (comment :initform (list
     "This is the category of algebras over non-commutative rings."
     "It is used by constructors of non-commutative algebras such as"
     "XPolynomialRing and XFreeAlgebra"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XAlgebra|
  (progn
    (push '|XAlgebra| *Categories*)
    (make-instance '|XAlgebraType|)))
```

———-

1.45 Level 12

— defvar level12 —

```
(defvar level12
 '(|AbelianMonoidRing| |FortranMachineTypeCategory| |FramedAlgebra|
   |FramedNonAssociativeAlgebra| |GcdDomain|
   |LinearOrdinaryDifferentialOperatorCategory| |OrderedIntegralDomain|
   |QuaternionCategory| |XFreeAlgebra|))
```

———-

1.45.1 AbelianMonoidRing

— defclass AbelianMonoidRingType —

```
(defclass |AbelianMonoidRingType| (|CharacteristicNonZeroType| |CharacteristicZeroType| |IntegralDomainType|)
  ((parents :initform '(|CharacteristicNonZero| |CharacteristicZero| |IntegralDomain|))
   (name :initform "AbelianMonoidRing")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'AMR)
   (comment :initform (list
     "Abelian monoid ring elements (not necessarily of finite support)"
     "of this ring are of the form formal SUM (r_i * e_i)"
     "where the r_i are coefficents and the e_i, elements of the"
     "ordered abelian monoid, are thought of as exponents or monomials."
     "The monomials commute with each other, and with"
     "the coefficients (which themselves may or may not be commutative)."
     "See FiniteAbelianMonoidRing for the case of finite support"
     "a useful common model for polynomials and power series."
     "Conceptually at least, only the non-zero terms are ever operated on."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AbelianMonoidRing|
  (progn
    (push '|AbelianMonoidRing| *Categories*)
    (make-instance '|AbelianMonoidRingType|)))
```

———-

1.45.2 FortranMachineTypeCategory

— defclass FortranMachineTypeCategoryType —

```
(defclass |FortranMachineTypeCategoryType| (|RetractableToType| |OrderedSetType|
                                            |IntegralDomainType|)
```

```
  ((parents :initform '(|RetractableTo| |OrderedSet|
                        |IntegralDomain|))
   (name :initform "FortranMachineTypeCategory")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'FMTC)
   (comment :initform (list
     "A category of domains which model machine arithmetic"
     "used by machines in the AXIOM-NAG link."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranMachineTypeCategory|
  (progn
    (push '|FortranMachineTypeCategory| *Categories*)
    (make-instance '|FortranMachineTypeCategoryType|)))
```

———-

1.45.3 FramedAlgebra

— defclass FramedAlgebraType —

```
(defclass |FramedAlgebraType| (|FiniteRankAlgebraType|)
  ((parents :initform '(|FiniteRankAlgebra|))
   (name :initform "FramedAlgebra")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'FRAMALG)
   (comment :initform (list
     "A FramedAlgebra is a FiniteRankAlgebra together"
     "with a fixed R-module basis."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedAlgebra|
  (progn
    (push '|FramedAlgebra| *Categories*)
    (make-instance '|FramedAlgebraType|)))
```

———-

1.45.4 FramedNonAssociativeAlgebra

— defclass FramedNonAssociativeAlgebraType —

```
(defclass |FramedNonAssociativeAlgebraType| (|FiniteRankNonAssociativeAlgebraType|)
  ((parents :initform '(|FiniteRankNonAssociativeAlgebra|))
   (name :initform "FramedNonAssociativeAlgebra")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'FRNAALG)
   (comment :initform (list
     "FramedNonAssociativeAlgebra(R) is a"
     "FiniteRankNonAssociativeAlgebra (a non associative"
     "algebra over R which is a free R-module of finite rank)"
     "over a commutative ring R together with a fixed R-module basis."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedNonAssociativeAlgebra|
  (progn
    (push '|FramedNonAssociativeAlgebra| *Categories*)
    (make-instance '|FramedNonAssociativeAlgebraType|)))
```

————-

1.45.5 GcdDomain

— defclass GcdDomainType —

```
(defclass |GcdDomainType| (|IntegralDomainType| |LeftOreRingType|)
  ((parents :initform '(|IntegralDomain| |LeftOreRing|))
   (name :initform "GcdDomain")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'GCDDOM)
   (comment :initform (list
     "This category describes domains where"
     "gcd can be computed but where there is no guarantee"
     "of the existence of factor operation for factorisation"
     "into irreducibles. However, if such a factor operation exist,"
     "factorization will be unique up to order and units."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GcdDomain|
  (progn
    (push '|GcdDomain| *Categories*)
    (make-instance '|GcdDomainType|)))
```

————-

```
( ?~=? : (%,%) -> Boolean
(~= (((|Boolean|) $ $) 7))

zero? : % -> Boolean
(|zero?| (((|Boolean|) $) 15))

unitNormal : % -> Record(unit: %,canonical: %,associate: %)
(|unitNormal| (((|Record| (|:| |unit| $) (|:| |canonical| $) (|:| |associate| $)) $) 40))

unitCanonical : % -> %
(|unitCanonical| (($ $) 39))

unit? : % -> Boolean
(|unit?| (((|Boolean|) $) 37))

subtractIfCan : (%,%) -> Union(%,"failed")
(|subtractIfCan| (((|Union| $ "failed") $ $) 18))

sample : () -> %
(|sample| (($) 16 T CONST))

recip : % -> Union(%,"failed")
(|recip| (((|Union| $ "failed") $) 33))

one? : % -> Boolean
(|one?| (((|Boolean|) $) 30))

lcmCoef : (%,%) -> Record(llcmres: %,coeff1: %,coeff2: %)
(|lcmCoef| (((|Record| (|:| |llcmres| $) (|:| |coeff1| $) (|:| |coeff2| $)) $ $) 48))

lcm : (%,%) -> %
lcm : List(%) -> %
(|lcm| (($ $ $) 45) (($ (|List| $)) 44))

latex : % -> String
(|latex| (((|String|) $) 9))

hash : % -> SingleInteger
(|hash| (((|SingleInteger|) $) 10))

gcdPolynomial : (SparseUnivariatePolynomial(%),SparseUnivariatePolynomial(%)) -> SparseUnivariatePolynomial(%)
(|gcdPolynomial| (((|SparseUnivariatePolynomial| $) (|SparseUnivariatePolynomial| $) (|SparseUnivariatePolynomial|

gcd : (%,%) -> %
gcd : List(%) -> %
(|gcd| (($ $ $) 47) (($ (|List| $)) 46))

exquo : (%,%) -> Union(%,"failed")
(|exquo| (((|Union| $ "failed") $ $) 41))

coerce : % -> OutputForm
coerce : Integer -> %
coerce : % -> %
(|coerce| (((|OutputForm|) $) 11) (($ (|Integer|)) 27) (($ $) 42))

characteristic : () -> NonNegativeInteger
(|characteristic| (((|NonNegativeInteger|)) 28))

associates? : (%,%) -> Boolean
(|associates?| (((|Boolean|) $ $) 38))
?^? : (%,PositiveInteger) -> %
?^? : (%,NonNegativeInteger) -> %
(^ (($ $ (|PositiveInteger|)) 25) (($ $ (|NonNegativeInteger|)) 32))

0 : () -> %
(|Zero| (($) 17 T CONST))

1 : () -> %
(|One| (($) 29 T CONST))

?=? : (%,%) -> Boolean
(= (((|Boolean|) $ $) 6))

-? : % -> %
?-? : (%,%) -> %
(- (($ $) 21) (($ $ $) 20))

?+? : (%,%) -> %
(+ (($ $ $) 13))

?**? : (%,PositiveInteger) -> %
?**? : (%,NonNegativeInteger) -> %
(** (($ $ (|PositiveInteger|)) 24) (($ $ (|NonNegativeInteger|)) 31))

?*? : (PositiveInteger,%) -> %
?*? : (NonNegativeInteger,%) -> %
?*? : (Integer,%) -> %
?*? : (%,%) -> %
(* (($ (|PositiveInteger|) $) 12) (($ (|NonNegativeInteger|) $) 14) (($ (|Integer|) $) 19) (($ $ $) 23)))
```

1.45.6 LinearOrdinaryDifferentialOperatorCategory

— defclass LinearOrdinaryDifferentialOperatorCategoryType —

```
(defclass |LinearOrdinaryDifferentialOperatorCategoryType| (
                  |EltableType| |UnivariateSkewPolynomialCategoryType|)
  ((parents :initform '(|Eltable| |UnivariateSkewPolynomialCategory|))
   (name :initform "LinearOrdinaryDifferentialOperatorCategory")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'LODOCAT)
   (comment :initform (list
     "LinearOrdinaryDifferentialOperatorCategory is the category"
     "of differential operators with coefficients in a ring A with a given"
     "derivation."
     " "
     "Multiplication of operators corresponds to functional composition:"
     "   (L1 * L2).(f) = L1 L2 f"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperatorCategory|
  (progn
    (push '|LinearOrdinaryDifferentialOperatorCategory| *Categories*)
    (make-instance '|LinearOrdinaryDifferentialOperatorCategoryType|)))
```

———-

1.45.7 OrderedIntegralDomain

— defclass OrderedIntegralDomainType —

```
(defclass |OrderedIntegralDomainType| (|OrderedRingType| |IntegralDomainType|)
  ((parents :initform '(|OrderedRing| |IntegralDomain|))
   (name :initform "OrderedIntegralDomain")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'OINTDOM)
   (comment :initform (list
     "The category of ordered commutative integral domains, where ordering"
     "and the arithmetic operations are compatible"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedIntegralDomain|
  (progn
    (push '|OrderedIntegralDomain| *Categories*)
    (make-instance '|OrderedIntegralDomainType|)))
```

———-

1.45.8 QuaternionCategory

— defclass QuaternionCategoryType —

```
(defclass |QuaternionCategoryType| (|ConvertibleToType|
                                    |FullyRetractableToType|
                                    |FullyEvalableOverType|
                                    |OrderedSetType|
                                    |CharacteristicNonZeroType|
                                    |CharacteristicZeroType|
                                    |DifferentialExtensionType|
                                    |FullyLinearlyExplicitRingOverType|
                                    |DivisionRingType|)
  ((parents :initform '(|ConvertibleTo|
                        |FullyRetractableTo|
                        |FullyEvalableOver|
                        |OrderedSet|
                        |CharacteristicNonZero|
                        |CharacteristicZero|
                        |DifferentialExtension|
                        |FullyLinearlyExplicitRingOver|
                        |DivisionRing|))
   (name :initform "QuaternionCategory")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'QUATCAT)
   (comment :initform (list
     "QuaternionCategory describes the category of quaternions"
     "and implements functions that are not representation specific."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuaternionCategory|
  (progn
    (push '|QuaternionCategory| *Categories*)
    (make-instance '|QuaternionCategoryType|)))
```

————-

1.45.9 XFreeAlgebra

— defclass XFreeAlgebraType —

```
(defclass |XFreeAlgebraType| (|RetractableToType| |XAlgebraType|)
  ((parents :initform '(|RetractableTo| |XAlgebra|))
   (name :initform "XFreeAlgebra")
   (marker :initform 'category)
   (level :initform 12)
   (abbreviation :initform 'XFALG)
   (comment :initform (list
     "This category specifies opeations for  polynomials"
     "and formal series with non-commutative variables."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XFreeAlgebra|
  (progn
    (push '|XFreeAlgebra| *Categories*)
    (make-instance '|XFreeAlgebraType|)))
```

————-

1.46 Level 13

— defvar level13 —

```
(defvar level13
 '(|FiniteAbelianMonoidRing| |IntervalCategory| |PowerSeriesCategory|
   |PrincipalIdealDomain| |UniqueFactorizationDomain| |XPolynomialsCat|))
```

———-

1.46.1 FiniteAbelianMonoidRing

— defclass FiniteAbelianMonoidRingType —

```
(defclass |FiniteAbelianMonoidRingType| (|FullyRetractableToType| |AbelianMonoidRingType|)
  ((parents :initform '(|FullyRetractableTo| |AbelianMonoidRing|))
   (name :initform "FiniteAbelianMonoidRing")
   (marker :initform 'category)
   (level :initform 13)
   (abbreviation :initform 'FAMR)
   (comment :initform (list
     "This category is similar to AbelianMonoidRing, except that the sum is"
     "assumed to be finite. It is a useful model for polynomials,"
     "but is somewhat more general."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteAbelianMonoidRing|
  (progn
    (push '|FiniteAbelianMonoidRing| *Categories*)
    (make-instance '|FiniteAbelianMonoidRingType|)))
```

———-

1.46.2 IntervalCategory

— defclass IntervalCategoryType —

```
(defclass |IntervalCategoryType| (|RadicalCategoryType| |RetractableToType|
                                  |TranscendentalFunctionCategoryType|
                                  |OrderedSetType| |GcdDomainType|)
  ((parents :initform '(|RadicalCategory| |RetractableTo|
                         |TranscendentalFunctionCategory|
                         |OrderedSet| |GcdDomain|))
   (name :initform "IntervalCategory")
   (marker :initform 'category)
   (level :initform 13)
   (abbreviation :initform 'INTCAT)
   (comment :initform (list
     "This category implements of interval arithmetic and transcendental"
     "functions over intervals."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntervalCategory|
  (progn
    (push '|IntervalCategory| *Categories*)
    (make-instance '|IntervalCategoryType|)))
```

———-

1.46.3 PowerSeriesCategory

— defclass PowerSeriesCategoryType —

```
(defclass |PowerSeriesCategoryType| (|AbelianMonoidRingType|)
  ((parents :initform '(|AbelianMonoidRing|))
   (name :initform "PowerSeriesCategory")
   (marker :initform 'category)
   (level :initform 13)
   (abbreviation :initform 'PSCAT)
   (comment :initform (list
     "PowerSeriesCategory is the most general power series"
     "category with exponents in an ordered abelian monoid."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PowerSeriesCategory|
  (progn
    (push '|PowerSeriesCategory| *Categories*)
    (make-instance '|PowerSeriesCategoryType|)))
```

———-

1.46.4 PrincipalIdealDomain

— defclass PrincipalIdealDomainType —

```
(defclass |PrincipalIdealDomainType| (|GcdDomainType|)
  ((parents :initform '(|GcdDomain|))
   (name :initform "PrincipalIdealDomain")
   (marker :initform 'category)
   (level :initform 13)
   (abbreviation :initform 'PID)
   (comment :initform (list
     "The category of constructive principal ideal domains, that is,"
     "where a single generator can be constructively found for"
     "any ideal given by a finite set of generators."
     "Note that this constructive definition only implies that"
     "finitely generated ideals are principal. It is not clear"
     "what we would mean by an infinitely generated ideal."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrincipalIdealDomain|
  (progn
    (push '|PrincipalIdealDomain| *Categories*)
    (make-instance '|PrincipalIdealDomainType|)))
```

————-

1.46.5 UniqueFactorizationDomain

— defclass UniqueFactorizationDomainType —

```
(defclass |UniqueFactorizationDomainType| (|GcdDomainType|)
  ((parents :initform '(|GcdDomain|))
   (name :initform "UniqueFactorizationDomain")
   (marker :initform 'category)
   (level :initform 13)
   (abbreviation :initform 'UFD)
   (comment :initform (list
     "A constructive unique factorization domain, where"
     "we can constructively factor members into a product of"
     "a finite number of irreducible elements."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UniqueFactorizationDomain|
  (progn
    (push '|UniqueFactorizationDomain| *Categories*)
    (make-instance '|UniqueFactorizationDomainType|)))
```

————-

1.46.6 XPolynomialsCat

— defclass XPolynomialsCatType —

```
(defclass |XPolynomialsCatType| (|XFreeAlgebraType|)
  ((parents :initform '(|XFreeAlgebra|))
   (name :initform "XPolynomialsCat")
   (marker :initform 'category)
   (level :initform 13)
   (abbreviation :initform 'XPOLYC)
   (comment :initform (list
     "The Category of polynomial rings with non-commutative variables."
     "The coefficient ring may be non-commutative too."
     "However coefficients commute with variables."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPolynomialsCat|
  (progn
    (push '|XPolynomialsCat| *Categories*)
    (make-instance '|XPolynomialsCatType|)))
```

———-

1.47 Level 14

— defvar level14 —

```
(defvar level14
  '(|EuclideanDomain| |MultivariateTaylorSeriesCategory|
    |PolynomialFactorizationExplicit| |UnivariatePowerSeriesCategory|))
```

———-

1.47.1 EuclideanDomain

— defclass EuclideanDomainType —

```
(defclass |EuclideanDomainType| (|PrincipalIdealDomainType|)
  ((parents :initform '(|PrincipalIdealDomain|))
   (name :initform "EuclideanDomain")
   (marker :initform 'category)
   (level :initform 14)
   (abbreviation :initform 'EUCDOM)
   (comment :initform (list
     "A constructive euclidean domain, one can divide producing"
     "a quotient and a remainder where the remainder is either zero"
     "or is smaller (euclideanSize) than the divisor."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EuclideanDomain|
  (progn
    (push '|EuclideanDomain| *Categories*)
    (make-instance '|EuclideanDomainType|)))
```

———-

1.47.2 MultivariateTaylorSeriesCategory

— defclass MultivariateTaylorSeriesCategoryType —

```
(defclass |MultivariateTaylorSeriesCategoryType| (|RadicalCategoryType|
                                                  |EvalableType|
                                                  |TranscendentalFunctionCategoryType|
                                                  |PartialDifferentialRingType|
                                                  |PowerSeriesCategoryType|)
  ((parents :initform '(|RadicalCategory|
                        |Evalable|
                        |TranscendentalFunctionCategory|
                        |PartialDifferentialRing|
                        |PowerSeriesCategory|))
   (name :initform "MultivariateTaylorSeriesCategory")
   (marker :initform 'category)
   (level :initform 14)
   (abbreviation :initform 'MTSCAT)
   (comment :initform (list
     "MultivariateTaylorSeriesCategory is the most general"
     "multivariate Taylor series category."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariateTaylorSeriesCategory|
  (progn
    (push '|MultivariateTaylorSeriesCategory| *Categories*)
    (make-instance '|MultivariateTaylorSeriesCategoryType|)))
```

———–

1.47.3 PolynomialFactorizationExplicit

— defclass PolynomialFactorizationExplicitType —

```
(defclass |PolynomialFactorizationExplicitType| (|UniqueFactorizationDomainType|)
  ((parents :initform '(|UniqueFactorizationDomain|))
   (name :initform "PolynomialFactorizationExplicit")
   (marker :initform 'category)
   (level :initform 14)
   (abbreviation :initform 'PFECAT)
   (comment :initform (list
     "This is the category of domains that know 'enough' about"
     "themselves in order to factor univariate polynomials over themselves."
     "This will be used in future releases for supporting factorization"
     "over finitely generated coefficient fields, it is not yet available"
     "in the current release of axiom."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFactorizationExplicit|
  (progn
    (push '|PolynomialFactorizationExplicit| *Categories*)
    (make-instance '|PolynomialFactorizationExplicitType|)))
```

———-

1.47.4 UnivariatePowerSeriesCategory

— defclass UnivariatePowerSeriesCategoryType —

```
(defclass |UnivariatePowerSeriesCategoryType| (|EltableType|
                                               |PowerSeriesCategoryType|
                                               |DifferentialRingType|
                                               |PartialDifferentialRingType|)
  ((parents :initform '(|Eltable|
                        |PowerSeriesCategory|
                        |DifferentialRing|
                        |PartialDifferentialRing|))
   (name :initform "UnivariatePowerSeriesCategory")
   (marker :initform 'category)
   (level :initform 14)
   (abbreviation :initform 'UPSCAT)
   (comment :initform (list
     "UnivariatePowerSeriesCategory is the most general"
     "univariate power series category with exponents in an ordered"
     "abelian monoid."
     "Note that this category exports a substitution function if it is"
     "possible to multiply exponents."
     "Also note that this category exports a derivative operation if it is"
     "possible to multiply coefficients by exponents."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePowerSeriesCategory|
  (progn
    (push '|UnivariatePowerSeriesCategory| *Categories*)
    (make-instance '|UnivariatePowerSeriesCategoryType|)))
```

———-

1.48 Level 15

— defvar level15 —

```
(defvar level15
  '(|Field| |IntegerNumberSystem| |PAdicIntegerCategory|
    |PolynomialCategory| |UnivariateTaylorSeriesCategory|))
```

———-

1.48.1 Field

— defclass FieldType —

```
(defclass |FieldType| (|DivisionRingType| |UniqueFactorizationDomainType| |EuclideanDomainType|)
  ((parents :initform '(|DivisionRing|
                         |UniqueFactorizationDomain| |EuclideanDomain|))
   (name :initform "Field")
   (marker :initform 'category)
   (level :initform 15)
   (abbreviation :initform 'FIELD)
   (comment :initform (list
     "The category of commutative fields, commutative rings"
     "where all non-zero elements have multiplicative inverses."
     "The factor operation while trivial is useful to have defined."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Field|
  (progn
    (push '|Field| *Categories*)
    (make-instance '|FieldType|)))
```

———-

1.48.2 IntegerNumberSystem

— defclass IntegerNumberSystemType —

```
(defclass |IntegerNumberSystemType| (|CombinatorialFunctionCategoryType|
                                     |RetractableToType|
                                     |RealConstantType|
                                     |PatternMatchableType|
                                     |StepThroughType|
                                     |CharacteristicZeroType|
                                     |DifferentialRingType|
                                     |LinearlyExplicitRingOverType|
                                     |OrderedIntegralDomainType|
                                     |UniqueFactorizationDomainType|
                                     |EuclideanDomainType|)
  ((parents :initform '(|CombinatorialFunctionCategory|
                        |RetractableTo|
                        |RealConstant|
                        |PatternMatchable|
                        |StepThrough|
                        |CharacteristicZero|
                        |DifferentialRing|
                        |LinearlyExplicitRingOver|
                        |OrderedIntegralDomain|
                        |UniqueFactorizationDomain|
                        |EuclideanDomain|))
   (name :initform "IntegerNumberSystem")
   (marker :initform 'category)
   (level :initform 15)
   (abbreviation :initform 'INS)
   (comment :initform (list
     "An IntegerNumberSystem is a model for the integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerNumberSystem|
  (progn
    (push '|IntegerNumberSystem| *Categories*)
    (make-instance '|IntegerNumberSystemType|)))
```

———-

1.48.3 PAdicIntegerCategory

— defclass PAdicIntegerCategoryType —

```
(defclass |PAdicIntegerCategoryType| (|CharacteristicZeroType| |EuclideanDomainType|)
  ((parents :initform '(|CharacteristicZero| |EuclideanDomain|))
   (name :initform "PAdicIntegerCategory")
   (marker :initform 'category)
   (level :initform 15)
   (abbreviation :initform 'PADICCT)
   (comment :initform (list
     "This is the category of stream-based representations of"
     "the p-adic integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicIntegerCategory|
  (progn
    (push '|PAdicIntegerCategory| *Categories*)
    (make-instance '|PAdicIntegerCategoryType|)))
```

———-

1.48.4 PolynomialCategory

— defclass PolynomialCategoryType —

```
(defclass |PolynomialCategoryType| (|ConvertibleToType|
                                    |EvalableType|
                                    |OrderedSetType|
                                    |PatternMatchableType|
                                    |PartialDifferentialRingType|
                                    |FullyLinearlyExplicitRingOverType|
                                    |FiniteAbelianMonoidRingType|
                                    |PolynomialFactorizationExplicitType|)
  ((parents :initform '(|ConvertibleTo|
                        |Evalable|
                        |OrderedSet|
                        |PatternMatchable|
                        |PartialDifferentialRing|
                        |FullyLinearlyExplicitRingOver|
                        |FiniteAbelianMonoidRing|
                        |PolynomialFactorizationExplicit|))
   (name :initform "PolynomialCategory")
   (marker :initform 'category)
   (level :initform 15)
   (abbreviation :initform 'POLYCAT)
   (comment :initform (list
     "The category for general multi-variate polynomials over a ring"
     "R, in variables from VarSet, with exponents from the OrderedAbelianMonoidSup."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialCategory|
  (progn
    (push '|PolynomialCategory| *Categories*)
    (make-instance '|PolynomialCategoryType|)))
```

———-

1.48.5 UnivariateTaylorSeriesCategory

— defclass UnivariateTaylorSeriesCategoryType —

```
(defclass |UnivariateTaylorSeriesCategoryType| (|RadicalCategoryType|
                                                |TranscendentalFunctionCategoryType|
                                                |UnivariatePowerSeriesCategoryType|)
  ((parents :initform '(|RadicalCategory|
                        |TranscendentalFunctionCategory|
                        |UnivariatePowerSeriesCategory|))
   (name :initform "UnivariateTaylorSeriesCategory")
   (marker :initform 'category)
   (level :initform 15)
   (abbreviation :initform 'UTSCAT)
   (comment :initform (list
     "UnivariateTaylorSeriesCategory is the category of Taylor"
     "series in one variable."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesCategory|
  (progn
    (push '|UnivariateTaylorSeriesCategory| *Categories*)
    (make-instance '|UnivariateTaylorSeriesCategoryType|)))
```

———-

1.49 Level 16

— defvar level16 —

```
(defvar level16
 '(|AlgebraicallyClosedField| |DifferentialPolynomialCategory|
   |FieldOfPrimeCharacteristic| |FunctionSpace| |LocalPowerSeriesCategory|
   |PseudoAlgebraicClosureOfPerfectFieldCategory| |QuotientFieldCategory|
   |RealClosedField| |RealNumberSystem| |RecursivePolynomialCategory|
   |UnivariateLaurentSeriesCategory| |UnivariatePolynomialCategory|
   |UnivariatePuiseuxSeriesCategory|))
```

———-

1.49.1 AlgebraicallyClosedField

— defclass AlgebraicallyClosedFieldType —

```
(defclass |AlgebraicallyClosedFieldType| (|RadicalCategoryType| |FieldType|)
  ((parents :initform '(|RadicalCategory| |Field|))
   (name :initform "AlgebraicallyClosedField")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'ACF)
   (comment :initform (list
     "Model for algebraically closed fields."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicallyClosedField|
  (progn
    (push '|AlgebraicallyClosedField| *Categories*)
    (make-instance '|AlgebraicallyClosedFieldType|)))
```

———-

1.49.2 DifferentialPolynomialCategory

— defclass DifferentialPolynomialCategoryType —

```
(defclass |DifferentialPolynomialCategoryType| (|DifferentialExtensionType| |PolynomialCategoryType|)
  ((parents :initform '(|DifferentialExtension|
                        |PolynomialCategory|))
   (name :initform "DifferentialPolynomialCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'DPOLCAT)
   (comment :initform (list
     "DifferentialPolynomialCategory is a category constructor"
     "specifying basic functions in an ordinary differential polynomial"
     "ring with a given ordered set of differential indeterminates."
     "In addition, it implements defaults for the basic functions."
     "The functions order and weight are extended"
     "from the set of derivatives of differential indeterminates"
     "to the set of differential polynomials.  Other operations"
     "provided on differential polynomials are"
     "leader, initial,"
     "separant, differentialVariables, and"
     "isobaric?.   Furthermore, if the ground ring is"
     "a differential ring, then evaluation (substitution"
     "of differential indeterminates by elements of the ground ring"
     "or by differential polynomials) is"
     "provided by eval."
     "A convenient way of referencing derivatives is provided by"
     "the functions makeVariable."
     " "
     "To construct a domain using this constructor, one needs"
     "to provide a ground ring R, an ordered set S of differential"
     "indeterminates, a ranking V on the set of derivatives"
     "of the differential indeterminates, and a set E of"
     "exponents in bijection with the set of differential monomials"
     "in the given differential indeterminates."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialPolynomialCategory|
  (progn
    (push '|DifferentialPolynomialCategory| *Categories*)
    (make-instance '|DifferentialPolynomialCategoryType|)))
```

———-

1.49.3 FieldOfPrimeCharacteristic

— defclass FieldOfPrimeCharacteristicType —

```
(defclass |FieldOfPrimeCharacteristicType| (|CharacteristicNonZeroType| |FieldType|)
  ((parents :initform '(|CharacteristicNonZero| |Field|))
   (name :initform "FieldOfPrimeCharacteristic")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'FPC)
   (comment :initform (list
     "FieldOfPrimeCharacteristic is the category of fields of prime"
     "characteristic, for example, finite fields, algebraic closures of"
     "fields of prime characteristic, transcendental extensions of"
     "of fields of prime characteristic."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FieldOfPrimeCharacteristic|
  (progn
    (push '|FieldOfPrimeCharacteristic| *Categories*)
    (make-instance '|FieldOfPrimeCharacteristicType|)))
```

————-

1.49.4 FunctionSpace

— defclass FunctionSpaceType —

```
(defclass |FunctionSpaceType| (|PatternableType|
                               |FullyRetractableToType|
                               |ExpressionSpaceType|
                               |FullyPatternMatchableType|
                               |GroupType|
                               |CharacteristicNonZeroType|
                               |CharacteristicZeroType|
                               |PartialDifferentialRingType|
                               |FullyLinearlyExplicitRingOverType|
                               |FieldType|)
  ((parents :initform '(|Patternable|
                        |FullyRetractableTo|
                        |ExpressionSpace|
                        |FullyPatternMatchable|
                        |Group|
                        |CharacteristicNonZero|
                        |CharacteristicZero|
                        |PartialDifferentialRing|
                        |FullyLinearlyExplicitRingOver|
                        |Field|))
   (name :initform "FunctionSpace")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'FS)
   (comment :initform (list
     "A space of formal functions with arguments in an arbitrary ordered set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpace|
  (progn
    (push '|FunctionSpace| *Categories*)
    (make-instance '|FunctionSpaceType|)))
```

———-

1.49.5 LocalPowerSeriesCategory

— defclass LocalPowerSeriesCategoryType —

```
(defclass |LocalPowerSeriesCategoryType| (|UnivariatePowerSeriesCategoryType| |FieldType|)
  ((parents :initform '(|UnivariatePowerSeriesCategory| |Field|))
   (name :initform "LocalPowerSeriesCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'LOCPOWC)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LocalPowerSeriesCategory|
  (progn
    (push '|LocalPowerSeriesCategory| *Categories*)
    (make-instance '|LocalPowerSeriesCategoryType|)))
```

———-

1.49.6 PseudoAlgebraicClosureOfPerfectFieldCategory

— defclass PseudoAlgebraicClosureOfPerfectFieldCategoryType —

```
(defclass |PseudoAlgebraicClosureOfPerfectFieldCategoryType| (|FieldType|)
  ((parents :initform '(|Field|))
   (name :initform "PseudoAlgebraicClosureOfPerfectFieldCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'PACPERC)
   (comment :initform (list
     "This category exports the function for domains"
     "which implement dynamic extension using the simple notion of tower"
     "extensions."
     "A tower extension T of the ground"
     "field K is any sequence of field extension "
     "(T : K_0, K_1, ..., K_i...,K_n) where K_0 = K"
     "and for i =1,2,...,n, K_i is an extension of K_{i-1} of degree > 1"
     "and defined by an irreducible polynomial p(Z) in K_{i-1}."
     "Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1)"
     "and (T_2: K_02, K_12,...,K_i2,...,K_n2)"
     "are said to be related if T_1 <= T_2 (or T_1 >= T_2),"
     "that is if K_i1 = K_i2 for i=1,2,...,n1 (or i=1,2,...,n2)."
     "Any algebraic operations defined for several elements"
     "are only defined if all of the concerned elements are coming from"
     "a set of related tower extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfPerfectFieldCategory|
  (progn
    (push '|PseudoAlgebraicClosureOfPerfectFieldCategory| *Categories*)
    (make-instance '|PseudoAlgebraicClosureOfPerfectFieldCategoryType|)))
```

———-

1.49.7 QuotientFieldCategory

— defclass QuotientFieldCategoryType —

```
(defclass |QuotientFieldCategoryType| (|PatternableType|
                                        |RetractableToType|
                                        |RealConstantType|
                                        |FullyEvalableOverType|
                                        |StepThroughType|
                                        |FullyPatternMatchableType|
                                        |CharacteristicNonZeroType|
                                        |CharacteristicZeroType|
                                        |DifferentialExtensionType|
                                        |FullyLinearlyExplicitRingOverType|
                                        |OrderedIntegralDomainType|
                                        |PolynomialFactorizationExplicitType|
                                        |FieldType|)
  ((parents :initform '(|Patternable|
                        |RetractableTo|
                        |RealConstant|
                        |FullyEvalableOver|
                        |StepThrough|
                        |FullyPatternMatchable|
                        |CharacteristicNonZero|
                        |CharacteristicZero|
                        |DifferentialExtension|
                        |FullyLinearlyExplicitRingOver|
                        |OrderedIntegralDomain|
                       |PolynomialFactorizationExplicit|
                       |Field|))
   (name :initform "QuotientFieldCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'QFCAT)
   (comment :initform (list
     "QuotientField(S) is the category of fractions of an Integral Domain S."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuotientFieldCategory|
  (progn
    (push '|QuotientFieldCategory| *Categories*)
    (make-instance '|QuotientFieldCategoryType|)))
```

———-

1.49.8 RealClosedField

— defclass RealClosedFieldType —

```
(defclass |RealClosedFieldType| (|RadicalCategoryType| |FullyRetractableToType|
                                 |CharacteristicZeroType| |OrderedRingType|
                                 |FieldType|)
  ((parents :initform '(|RadicalCategory| |FullyRetractableTo|
                        |CharacteristicZero| |OrderedRing|
                        |Field|))
   (name :initform "RealClosedField")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'RCFIELD)
   (comment :initform (list
     "RealClosedField provides common access"
     "functions for all real closed fields."
     "provides computations with generic real roots of polynomials"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealClosedField|
  (progn
    (push '|RealClosedField| *Categories*)
    (make-instance '|RealClosedFieldType|)))
```

———-

1.49.9 RealNumberSystem

— defclass RealNumberSystemType —

```
(defclass |RealNumberSystemType| (|RadicalCategoryType| |RetractableToType|
                                  |RealConstantType| |PatternMatchableType|
                                  |CharacteristicZeroType| |OrderedRingType| |FieldType|)
  ((parents :initform '(|RadicalCategory| |RetractableTo|
                        |RealConstant| |PatternMatchable|
                        |CharacteristicZero| |OrderedRing| |Field|))
   (name :initform "RealNumberSystem")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'RNS)
   (comment :initform (list
     "The real number system category is intended as a model for the real"
     "numbers.  The real numbers form an ordered normed field.  Note that"
     "we have purposely not included DifferentialRing or"
     "the elementary functions (see TranscendentalFunctionCategory)"
     "in the definition."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealNumberSystem|
  (progn
    (push '|RealNumberSystem| *Categories*)
    (make-instance '|RealNumberSystemType|)))
```

———-

1.49.10 RecursivePolynomialCategory

— defclass RecursivePolynomialCategoryType —

```
(defclass |RecursivePolynomialCategoryType| (|PolynomialCategoryType|)
  ((parents :initform '(|PolynomialCategory|))
   (name :initform "RecursivePolynomialCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'RPOLCAT)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RecursivePolynomialCategory|
  (progn
    (push '|RecursivePolynomialCategory| *Categories*)
    (make-instance '|RecursivePolynomialCategoryType|)))
```

———-

1.49.11 UnivariateLaurentSeriesCategory

— defclass UnivariateLaurentSeriesCategoryType —

```
(defclass |UnivariateLaurentSeriesCategoryType| (|RadicalCategoryType|
                                                 |TranscendentalFunctionCategoryType|
                                                 |UnivariatePowerSeriesCategoryType|
                                                 |FieldType|)
  ((parents :initform '(|RadicalCategory|
                        |TranscendentalFunctionCategory|
                        |UnivariatePowerSeriesCategory|
                        |Field|))
   (name :initform "UnivariateLaurentSeriesCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'ULSCAT)
   (comment :initform (list
     "UnivariateLaurentSeriesCategory is the category of"
     "Laurent series in one variable."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeriesCategory|
  (progn
    (push '|UnivariateLaurentSeriesCategory| *Categories*)
    (make-instance '|UnivariateLaurentSeriesCategoryType|)))
```

———-

1.49.12 UnivariatePolynomialCategory

— defclass UnivariatePolynomialCategoryType —

```
(defclass |UnivariatePolynomialCategoryType| (|EltableType| |StepThroughType|
                                              |DifferentialExtensionType|
                                              |EuclideanDomainType|
                                              |PolynomialCategoryType|)
  ((parents :initform '(|Eltable| |StepThrough|
                        |DifferentialExtension|
                        |EuclideanDomain|
                        |PolynomialCategory|))
   (name :initform "UnivariatePolynomialCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'UPOLYC)
   (comment :initform (list
     "The category of univariate polynomials over a ring R."
     "No particular model is assumed - implementations can be either"
```

```
    "sparse or dense."))
  (argslist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |UnivariatePolynomialCategory|
  (progn
    (push '|UnivariatePolynomialCategory| *Categories*)
    (make-instance '|UnivariatePolynomialCategoryType|)))
```

———-

1.49.13 UnivariatePuiseuxSeriesCategory

— defclass UnivariatePuiseuxSeriesCategoryType —

```
(defclass |UnivariatePuiseuxSeriesCategoryType| (|RadicalCategoryType|
                                                 |TranscendentalFunctionCategoryType|
                                                 |UnivariatePowerSeriesCategoryType|
                                                 |FieldType|)
  ((parents :initform '(|RadicalCategory|
                        |TranscendentalFunctionCategory|
                        |UnivariatePowerSeriesCategory|
                        |Field|))
   (name :initform "UnivariatePuiseuxSeriesCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'UPXSCAT)
   (comment :initform (list
     "UnivariatePuiseuxSeriesCategory is the category of Puiseux"
     "series in one variable."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesCategory|
  (progn
    (push '|UnivariatePuiseuxSeriesCategory| *Categories*)
    (make-instance '|UnivariatePuiseuxSeriesCategoryType|)))
```

———-

1.50 Level 17

— defvar level17 —

```
(defvar level17
  '(|AlgebraicallyClosedFunctionSpace| |ExtensionField| |FiniteFieldCategory|
    |FloatingPointSystem| |UnivariateLaurentSeriesConstructorCategory|
    |UnivariatePuiseuxSeriesConstructorCategory|))
```

———-

1.50.1 AlgebraicallyClosedFunctionSpace

— defclass AlgebraicallyClosedFunctionSpaceType —

```
(defclass |AlgebraicallyClosedFunctionSpaceType| (|AlgebraicallyClosedFieldType| |FunctionSpaceType|)
  ((parents :initform '(|AlgebraicallyClosedField| |FunctionSpace|))
   (name :initform "AlgebraicallyClosedFunctionSpace")
   (marker :initform 'category)
   (level :initform 17)
   (abbreviation :initform 'ACFS)
   (comment :initform (list
     "Model for algebraically closed function spaces."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicallyClosedFunctionSpace|
  (progn
    (push '|AlgebraicallyClosedFunctionSpace| *Categories*)
    (make-instance '|AlgebraicallyClosedFunctionSpaceType|)))
```

———-

1.50.2 ExtensionField

— defclass ExtensionFieldType —

```
(defclass |ExtensionFieldType| (|RetractableToType|
                                |CharacteristicZeroType|
                                |VectorSpaceType|
                                |FieldOfPrimeCharacteristicType|)
  ((parents :initform '(|RetractableTo|
                        |CharacteristicZero|
                        |VectorSpace|
                        |FieldOfPrimeCharacteristic|))
   (name :initform "ExtensionField")
   (marker :initform 'category)
   (level :initform 17)
   (abbreviation :initform 'XF)
   (comment :initform (list
     "ExtensionField F is the category of fields which extend the field F"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExtensionField|
  (progn
    (push '|ExtensionField| *Categories*)
    (make-instance '|ExtensionFieldType|)))
```

———-

1.50.3 FiniteFieldCategory

— defclass FiniteFieldCategoryType —

```
(defclass |FiniteFieldCategoryType| (|FiniteType| |StepThroughType|
                                     |DifferentialRingType|
                                     |FieldOfPrimeCharacteristicType|)
  ((parents :initform '(|Finite| |StepThrough|
                        |DifferentialRing|
                        |FieldOfPrimeCharacteristic|))
   (name :initform "FiniteFieldCategory")
   (marker :initform 'category)
   (level :initform 17)
   (abbreviation :initform 'FFIELDC)
   (comment :initform (list
     "FiniteFieldCategory is the category of finite fields"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCategory|
  (progn
    (push '|FiniteFieldCategory| *Categories*)
    (make-instance '|FiniteFieldCategoryType|)))
```

———-

1.50.4 FloatingPointSystem

— defclass FloatingPointSystemType —

```
(defclass |FloatingPointSystemType| (|RealNumberSystemType|)
  ((parents :initform '(|RealNumberSystem|))
   (name :initform "FloatingPointSystem")
   (marker :initform 'category)
   (level :initform 17)
   (abbreviation :initform 'FPS)
   (comment :initform (list
     "This category is intended as a model for floating point systems."
     "A floating point system is a model for the real numbers.  In fact,"
     "it is an approximation in the sense that not all real numbers are"
     "exactly representable by floating point numbers."
     "A floating point system is characterized by the following:"
     " "
     "1: base of the exponent where the actual implemenations are"
     "usually binary or decimal)"
     "2: precision of the mantissa (arbitrary or fixed)"
     "3: rounding error for operations"
      " "
     "Because a Float is an approximation to the real numbers, even though"
     "it is defined to be a join of a Field and OrderedRing, some of"
     "the attributes do not hold.  In particular associative('+')"
     "does not hold.  Algorithms defined over a field need special"
     "considerations when the field is a floating point system."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatingPointSystem|
  (progn
    (push '|FloatingPointSystem| *Categories*)
    (make-instance '|FloatingPointSystemType|)))
```

————-

1.50.5 UnivariateLaurentSeriesConstructorCategory

— defclass UnivariateLaurentSeriesConstructorCategoryType —

```
(defclass |UnivariateLaurentSeriesConstructorCategoryType| (
     |QuotientFieldCategoryType| |UnivariateLaurentSeriesCategoryType|)
  ((parents :initform '(
     |QuotientFieldCategory| |UnivariateLaurentSeriesCategory|))
   (name :initform "UnivariateLaurentSeriesConstructorCategory")
   (marker :initform 'category)
   (level :initform 17)
   (abbreviation :initform 'ULSCCAT)
   (comment :initform (list
     "This is a category of univariate Laurent series constructed from"
     "univariate Taylor series.  A Laurent series is represented by a pair"
     "[n,f(x)], where n is an arbitrary integer and f(x)"
     "is a Taylor series.  This pair represents the Laurent series"
     "x**n * f(x)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeriesConstructorCategory|
  (progn
    (push '|UnivariateLaurentSeriesConstructorCategory| *Categories*)
    (make-instance '|UnivariateLaurentSeriesConstructorCategoryType|)))
```

———-

1.50.6 UnivariatePuiseuxSeriesConstructorCategory

— defclass UnivariatePuiseuxSeriesConstructorCategoryType —

```
(defclass |UnivariatePuiseuxSeriesConstructorCategoryType| (|RetractableToType|
                                                            |UnivariatePuiseuxSeriesCategoryType|)
  ((parents :initform '(|RetractableTo| |UnivariatePuiseuxSeriesCategory|))
   (name :initform "UnivariatePuiseuxSeriesConstructorCategory")
   (marker :initform 'category)
   (level :initform 17)
   (abbreviation :initform 'UPXSCCA)
   (comment :initform (list
     "This is a category of univariate Puiseux series constructed"
     "from univariate Laurent series.  A Puiseux series is represented"
     "by a pair [r,f(x)], where r is a positive rational number and"
     "f(x) is a Laurent series.  This pair represents the Puiseux"
     "series f(x^r)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesConstructorCategory|
  (progn
    (push '|UnivariatePuiseuxSeriesConstructorCategory| *Categories*)
    (make-instance '|UnivariatePuiseuxSeriesConstructorCategoryType|)))
```

———-

1.51 Level 18

— defvar level18 —

```
(defvar level18
  '(|FiniteAlgebraicExtensionField| |MonogenicAlgebra|
    |PseudoAlgebraicClosureOfFiniteFieldCategory|
    |PseudoAlgebraicClosureOfRationalNumberCategory|))
```

———-

1.51.1 FiniteAlgebraicExtensionField

— defclass FiniteAlgebraicExtensionFieldType —

```
(defclass |FiniteAlgebraicExtensionFieldType| (|FiniteFieldCategoryType| |ExtensionFieldType|)
  ((parents :initform '(|FiniteFieldCategory| |ExtensionField|))
   (name :initform "FiniteAlgebraicExtensionField")
   (marker :initform 'category)
   (level :initform 18)
   (abbreviation :initform 'FAXF)
   (comment :initform (list
     "FiniteAlgebraicExtensionField F is the category of fields"
     "which are finite algebraic extensions of the field F."
     "If F is finite then any finite algebraic extension of F"
     "is finite, too. Let K be a finite algebraic extension of the"
     "finite field F. The exponentiation of elements of K"
     "defines a Z-module structure on the multiplicative group of K."
     "The additive group of K becomes a module over the ring of"
     "polynomials over F via the operation"
     "linearAssociatedExp(a:K,f:SparseUnivariatePolynomial F)"
     "which is linear over F, that is, for elements a from K,"
     "c,d from F and f,g univariate polynomials over F"
     "we have linearAssociatedExp(a,cf+dg) equals c times"
     "linearAssociatedExp(a,f) plus d times"
     "linearAssociatedExp(a,g)."
     "Therefore linearAssociatedExp is defined completely by"
     "its action on  monomials from F[X]:"
     "linearAssociatedExp(a,monomial(1,k)$SUP(F)) is defined to be"
     "Frobenius(a,k) which is a**(q**k) where q=size()$F."
     "The operations order and discreteLog associated with the multiplicative"
     "exponentiation have additive analogues associated to the operation"
     "linearAssociatedExp. These are the functions"
     "linearAssociatedOrder and linearAssociatedLog,"
     "respectively."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteAlgebraicExtensionField|
  (progn
    (push '|FiniteAlgebraicExtensionField| *Categories*)
    (make-instance '|FiniteAlgebraicExtensionFieldType|)))
```

————-

1.51.2 MonogenicAlgebra

— defclass MonogenicAlgebraType —

```
(defclass |MonogenicAlgebraType| (|ConvertibleToType| |FullyRetractableToType|
                                  |DifferentialExtensionType|
                                  |FullyLinearlyExplicitRingOverType|
                                  |FramedAlgebraType| |FiniteFieldCategoryType|)
  ((parents :initform '(|ConvertibleTo| |FullyRetractableTo|
                        |DifferentialExtension|
                        |FullyLinearlyExplicitRingOver|
                        |FramedAlgebra| |FiniteFieldCategory|))
   (name :initform "MonogenicAlgebra")
   (marker :initform 'category)
   (level :initform 18)
   (abbreviation :initform 'MONOGEN)
   (comment :initform (list
     "A MonogenicAlgebra is an algebra of finite rank which"
     "can be generated by a single element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonogenicAlgebra|
  (progn
    (push '|MonogenicAlgebra| *Categories*)
    (make-instance '|MonogenicAlgebraType|)))
```

———–

1.51.3 PseudoAlgebraicClosureOfFiniteFieldCategory

— defclass PseudoAlgebraicClosureOfFiniteFieldCategoryType —

```
(defclass |PseudoAlgebraicClosureOfFiniteFieldCategoryType| (
    |PseudoAlgebraicClosureOfPerfectFieldCategoryType| |FiniteFieldCategoryType|)
  ((parents :initform '(
     |PseudoAlgebraicClosureOfPerfectFieldCategory| |FiniteFieldCategory|))
   (name :initform "PseudoAlgebraicClosureOfFiniteFieldCategory")
   (marker :initform 'category)
   (level :initform 18)
   (abbreviation :initform 'PACFFC)
   (comment :initform (list
     "This category exports the function for the domain"
     "PseudoAlgebraicClosureOfFiniteField which implement dynamic extension"
     "using the simple notion of tower extensions."
     "A tower extension T of the ground"
     "field K is any sequence of field extension (T : K_0, K_1, ..., K_i...,K_n)"
     "where K_0 = K and for i =1,2,...,n, K_i is an extension"
     "of K_{i-1} of degree > 1 and defined by an irreducible polynomial"
     "p(Z) in K_{i-1}."
     "Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1)"
     "and (T_2: K_02, K_12,...,K_i2,...,K_n2)"
     "are said to be related if T_1 <= T_2 (or T_1 >= T_2),"
     "that is if K_i1 = K_i2 for i=1,2,...,n1"
     "(or i=1,2,...,n2). Any algebraic operations defined for several elements"
     "are only defined if all of the concerned elements are comming from"
     "a set of related tour extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfFiniteFieldCategory|
  (progn
    (push '|PseudoAlgebraicClosureOfFiniteFieldCategory| *Categories*)
    (make-instance '|PseudoAlgebraicClosureOfFiniteFieldCategoryType|)))
```

———-

1.51.4 PseudoAlgebraicClosureOfRationalNumberCategory

— defclass PseudoAlgebraicClosureOfRationalNumberCategoryType —

```
(defclass |PseudoAlgebraicClosureOfRationalNumberCategoryType| (
                |ExtensionFieldType| |PseudoAlgebraicClosureOfPerfectFieldCategoryType|)
  ((parents :initform '(|ExtensionField| |PseudoAlgebraicClosureOfPerfectFieldCategory|))
   (name :initform "PseudoAlgebraicClosureOfRationalNumberCategory")
   (marker :initform 'category)
   (level :initform 18)
   (abbreviation :initform 'PACRATC)
   (comment :initform (list
     "This category exports the function for the domain"
     "PseudoAlgebraicClosureOfRationalNumber"
     "which implement dynamic extension using the simple notion of tower"
     "extensions. A tower extension T of the ground"
     "field K is any sequence of field extension (T : K_0, K_1, ..., K_i...,K_n)"
     "where K_0 = K and for i =1,2,...,n, K_i is an extension"
     "of K_{i-1} of degree > 1 and defined by an irreducible polynomial"
     "p(Z) in K_{i-1}."
     "Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1)"
     "and (T_2: K_02, K_12,...,K_i2,...,K_n2)"
     "are said to be related if T_1 <= T_2 (or T_1 >= T_2),"
     "that is if K_i1 = K_i2 for i=1,2,...,n1"
     "(or i=1,2,...,n2). Any algebraic operations defined for several elements"
     "are only defined if all of the concerned elements are comming from"
     "a set of related tour extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfRationalNumberCategory|
  (progn
    (push '|PseudoAlgebraicClosureOfRationalNumberCategory| *Categories*)
    (make-instance '|PseudoAlgebraicClosureOfRationalNumberCategoryType|)))
```

———-

1.52 Level 19

— defvar level19 —

```
(defvar level19
  '(|ComplexCategory| |FunctionFieldCategory|
    |PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory|))
```

———-

1.52.1 ComplexCategory

— defclass ComplexCategoryType —

```
(defclass |ComplexCategoryType| (|PatternableType| |RadicalCategoryType|
                                 |TranscendentalFunctionCategoryType|
                                 |FullyEvalableOverType| |OrderedSetType|
                                 |FullyPatternMatchableType|
                                 |PolynomialFactorizationExplicitType|
                                 |MonogenicAlgebraType|)
  ((parents :initform '(|Patternable| |RadicalCategory|
                        |TranscendentalFunctionCategory|
                        |FullyEvalableOver| |OrderedSet|
                        |FullyPatternMatchable|
                        |PolynomialFactorizationExplicit|
                        |MonogenicAlgebra|))
   (name :initform "ComplexCategory")
   (marker :initform 'category)
   (level :initform 19)
   (abbreviation :initform 'COMPCAT)
   (comment :initform (list
     "This category represents the extension of a ring by a square root of -1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexCategory|
  (progn
    (push '|ComplexCategory| *Categories*)
    (make-instance '|ComplexCategoryType|)))
```

————-

1.52.2 FunctionFieldCategory

— defclass FunctionFieldCategoryType —

```
(defclass |FunctionFieldCategoryType| (|MonogenicAlgebraType|)
  ((parents :initform '(|MonogenicAlgebra|))
   (name :initform "FunctionFieldCategory")
   (marker :initform 'category)
   (level :initform 19)
   (abbreviation :initform 'FFCAT)
   (comment :initform (list
     "This category is a model for the function field of a"
     "plane algebraic curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |FunctionFieldCategory|
  (progn
    (push '|FunctionFieldCategory| *Categories*)
    (make-instance '|FunctionFieldCategoryType|)))
```

———-

1.52.3 PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory

— defclass PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategoryType —

```
(defclass |PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategoryType| (
            |PseudoAlgebraicClosureOfRationalNumberCategoryType|)
  ((parents :initform '(|PseudoAlgebraicClosureOfRationalNumberCategory|))
   (name :initform "PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory")
   (marker :initform 'category)
   (level :initform 19)
   (abbreviation :initform 'PACEXTC)
   (comment :initform (list
     "This category exports the function for the domain"
     "PseudoAlgebraicClosureOfAlgExtOfRationalNumber which implement dynamic"
     "extension using the simple notion of tower extensions. A tower extension"
     "T of the ground field K is any sequence of field extension"
     "(T : K0, K1, ..., Ki...,Kn) where K0 = K and for i =1,2,...,n,"
     "Ki is an extension of K{i-1} of degree > 1 and defined by an"
     "irreducible polynomial p(Z) in K{i-1}."
     "Two towers (T1: K01, K11,...,Ki1,...,Kn1) and"
     "(T2: K02, K12,...,Ki2,...,Kn2)"
     "are said to be related if T1 <= T2 (or T1 >= T2),"
     "that is if Ki1 = Ki2 for i=1,2,...,n1 (or i=1,2,...,n2)."
     "Any algebraic operations defined for several elements"
     "are only defined if all of the concerned elements are comming from"
     "a set of related tour extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory|
  (progn
    (push '|PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory| *Categories*)
    (make-instance '|PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategoryType|)))
```

———-

The NonNegativeInteger Domain

```
)show NNI
 NonNegativeInteger is a domain constructor
 Abbreviation for NonNegativeInteger is NNI
 This constructor is exposed in this frame.
 Issue )edit bookvol10.3.pamphlet to see algebra source code for NNI

------------------------------- Operations --------------------------------
 ?*? : (%,%) -> %                      ?*? : (NonNegativeInteger,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?**? : (%,PositiveInteger) -> %
 ?**? : (%,NonNegativeInteger) -> %    ?+? : (%,%) -> %
 ?<? : (%,%) -> Boolean                ?<=? : (%,%) -> Boolean
 ?=? : (%,%) -> Boolean                ?>? : (%,%) -> Boolean
 ?>=? : (%,%) -> Boolean               1 : () -> %
 0 : () -> %                           ?^? : (%,PositiveInteger) -> %
 ?^? : (%,NonNegativeInteger) -> %     coerce : % -> OutputForm
 gcd : (%,%) -> %                      hash : % -> SingleInteger
 latex : % -> String                   max : (%,%) -> %
 min : (%,%) -> %                      one? : % -> Boolean
 qcoerce : Integer -> %                ?quo? : (%,%) -> %
 random : % -> %                       recip : % -> Union(%,"failed")
 ?rem? : (%,%) -> %                    sample : () -> %
 shift : (%,Integer) -> %              sup : (%,%) -> %
 zero? : % -> Boolean                  ?~=? : (%,%) -> Boolean
 divide : (%,%) -> Record(quotient: %,remainder: %)
 exquo : (%,%) -> Union(%,"failed")
 subtractIfCan : (%,%) -> Union(%,"failed")
```

1.52.4 Database Files

Database files are very similar to kaf files except that there is an optimization (currently broken) which makes the first item a pair of two numbers. The first number in the pair is the offset of the key-value table, the second is a time stamp. If the time stamp in the database matches the time stamp in the image the database is not needed (since the internal hash tables already contain all of the information). When the database is built the time stamp is saved in both the gcl image and the database.

Regarding the 'ancestors field in a category: At database build time there exists a *ancestors-hash* hash table that gets filled with CATEGORY (not domain) ancestor information. This later provides the information that goes into interp.daase This *ancestors-hash* does not exist at normal runtime (it can be made by a call to genCategoryTable). Note that the ancestor information in *ancestors-hash* (and hence interp.daase) involves `#1`, `#2`, etc instead of R, Coef, etc. The latter thingies appear in all .nrlib/index.kaf files. So we need to be careful when we)lib categories and update the ancestor info.

This file contains the code to build, open and access the .daase files. This file contains the code to)library nrlibs and asy files

There is a major issue about the data that resides in these databases. the fundamental problem is that the system requires more information to build the databases than it needs to run the interpreter. in particular, modemap.daase is constructed using properties like "modemaps" but the interpreter will never ask for this information.

So, the design is as follows:

- the modemap.daase needs to be built. this is done by doing a)library on ALL of the nrlib files that are going into the system. this will bring in "modemap" information and add it to the *modemaps-hash* hashtable.
- database build proceeds, accessing the "modemap" property from the hashtables. once this completes this information is never used again.
- the interp.daase database is built. this contains only the information necessary to run the interpreter. note that during the running of the interpreter users can extend the system by do a)library on a new nrlib file. this will cause fields such as "modemap" to be read and hashed.

Each constructor (e.g. LIST) had one library directory (e.g. LIST.nrlib). This directory contained a random access file called the index.kaf file. These files contain runtime information such as the operationAlist and the ConstructorModemap. At system build time we merge all of these .nrlib/index.kaf files into one database, INTERP.daase. Requests to get information from this database are cached so that multiple references do not cause additional disk i/o.

This database is left open at all times as it is used frequently by the interpreter. one minor complication is that newly compiled files need to override information that exists in this database.

The design calls for constructing a random read (kaf format) file that is accessed by functions that cache their results. when the database is opened the list of constructor-index pairs is hashed by constructor name. a request for information about a constructor causes the information to replace the index in the hash table. since the index is a number and the data is a non-numeric sexpr there is no source of confusion about when the data needs to be read.

The format of this new database is as follows:

```
first entry:
  an integer giving the byte offset to the constructor alist
  at the bottom of the file
second and subsequent entries (one per constructor)
  (operationAlist)
  (constructorModemap)
  ....
last entry: (pointed at by the first entry)
  an alist of (constructor . index) e.g.
     ( (PI offset-of-operationAlist offset-of-constructorModemap)
     (NNI offset-of-operationAlist offset-of-constructorModemap)
      ....)
 This list is read at open time and hashed by the car of each item.
```

The system has been changed to use the property list of the symbols rather than hash tables. since we already hashed once to get the symbol we need only an offset to get the property list. this also has the advantage that eq hash tables no longer need to be moved during garbage collection.

There are 3 potential speedups that could be done.

- the best would be to use the value cell of the symbol rather than the property list but i'm unable to determine all uses of the value cell at the present time.
- a second speedup is to guarantee that the property list is a single item, namely the database structure. this removes an assoc but leaves one open to breaking the system if someone adds something to the property list. this was not done because of the danger mentioned.
- a third speedup is to make the getdatabase call go away, either by making it a macro or eliding it entirely. this was not done because we want to keep the flexibility of changing the database forms.

The new design does not use hash tables. the database structure contains an entry for each item that used to be in a hash table. initially the structure contains file-position pointers and these are replaced by real data when they are first looked up. the database structure is kept on the property list of the constructor, thus, (get '—DenavitHartenbergMatrix— 'database) will return the database structure object.

Each operation has a property on its symbol name called 'operation which is a list of all of the signatures of operations with that name.

— initvars —

```
(defstruct database
 abbreviation              ; interp.
 ancestors                 ; interp.
 constructor               ; interp.
 constructorcategory       ; interp.
 constructorkind           ; interp.
 constructormodemap        ; interp.
 cosig                     ; interp.
 defaultdomain             ; interp.
 modemaps                  ; interp.
 niladic                   ; interp.
 object                    ; interp.
 operationalist            ; interp.
 documentation             ; browse.
 constructorform           ; browse.
 attributes                ; browse.
 predicates                ; browse.
 sourcefile                ; browse.
 parents                   ; browse.
 users                     ; browse.
 dependents                ; browse.
 spare                     ; superstition
 ) ; database structure
```

———-

There are only a small number of domains that have default domains. rather than keep this slot in every domain we maintain a list here.

— initvars —

```
(defvar *defaultdomain-list* '(
  (|MultisetAggregate| |Multiset|)
  (|FunctionSpace| |Expression|)
  (|AlgebraicallyClosedFunctionSpace| |Expression|)
  (|ThreeSpaceCategory| |ThreeSpace|)
  (|DequeueAggregate| |Dequeue|)
  (|ComplexCategory| |Complex|)
  (|LazyStreamAggregate| |Stream|)
  (|AssociationListAggregate| |AssociationList|)
  (|QuaternionCategory| |Quaternion|)
  (|PriorityQueueAggregate| |Heap|)
  (|PointCategory| |Point|)
  (|PlottableSpaceCurveCategory| |Plot3D|)
  (|PermutationCategory| |Permutation|)
  (|StringCategory| |String|)
  (|FileNameCategory| |FileName|)
  (|OctonionCategory| |Octonion|)))
```

———-

— initvars —

```
(defvar *operation-hash* nil "given an operation name, what are its modemaps?")
```

———-

This hash table is used to answer the question "does domain x have category y?". this is answered by constructing a pair of (x . y) and doing an equal hash into this table.

— initvars —

```
(defvar *hasCategory-hash* nil "answers x has y category questions")
```

———-

This variable is used for debugging. If a hash table lookup fails and this variable is non-nil then a message is printed.

— initvars —

```
(defvar *miss* nil "print out cache misses on getdatabase calls")
```

———-

Note that constructorcategory information need only be kept for items of type category. this will be fixed in the next iteration when the need for the various caches are reviewed

Note that the *modemaps-hash* information does not need to be kept for system files. these are precomputed and kept in modemap.daase however, for user-defined files these are needed. Currently these are added to the database for 2 reasons; there is a still-unresolved issue of user database extensions and this information is used during database build time

```
)lisp (showdatabase '|NonNegativeInteger|)
getdatabase call: NonNegativeInteger   CONSTRUCTORKIND
CONSTRUCTORKIND: domain
getdatabase call: NonNegativeInteger   COSIG
COSIG: (NIL)
getdatabase call: NonNegativeInteger   OPERATION
OPERATION: NIL
CONSTRUCTORMODEMAP:
```

```
getdatabase call: NonNegativeInteger   CONSTRUCTORMODEMAP

(((|NonNegativeInteger|)
  (|Join| (|OrderedAbelianMonoidSup|) (|Monoid|)
          (CATEGORY |domain| (SIGNATURE |quo| ($ $ $))
              (SIGNATURE |rem| ($ $ $)) (SIGNATURE |gcd| ($ $ $))
              (SIGNATURE |divide|
                  ((|Record| (|:| |quotient| $) (|:| |remainder| $)) $
                   $))
              (SIGNATURE |exquo| ((|Union| $ "failed") $ $))
              (SIGNATURE |shift| ($ $ (|Integer|)))
              (SIGNATURE |random| ($ $))
              (SIGNATURE |qcoerce| ($ (|Integer|))))))
 (T |NonNegativeInteger|))
CONSTRUCTORCATEGORY:
getdatabase call: NonNegativeInteger   CONSTRUCTORCATEGORY
getdatabase miss: NonNegativeInteger   CONSTRUCTORCATEGORY

(|Join| (|OrderedAbelianMonoidSup|) (|Monoid|)
        (CATEGORY |domain| (SIGNATURE |quo| ($ $ $))
            (SIGNATURE |rem| ($ $ $)) (SIGNATURE |gcd| ($ $ $))
            (SIGNATURE |divide|
                ((|Record| (|:| |quotient| $) (|:| |remainder| $)) $ $))
            (SIGNATURE |exquo| ((|Union| $ "failed") $ $))
            (SIGNATURE |shift| ($ $ (|Integer|)))
            (SIGNATURE |random| ($ $))
            (SIGNATURE |qcoerce| ($ (|Integer|)))))
OPERATIONALIST:
getdatabase call: NonNegativeInteger   OPERATIONALIST

((~= (((|Boolean|) $ $) NIL)) (|zero?| (((|Boolean|) $) NIL))
 (|sup| (($ $ $) 6)) (|subtractIfCan| (((|Union| $ "failed") $ $) 10))
 (|shift| (($ $ (|Integer|)) 7)) (|sample| (($) NIL T CONST))
 (|rem| (($ $ $) NIL)) (|recip| (((|Union| $ "failed") $) NIL))
 (|random| (($ $) NIL)) (|quo| (($ $ $) NIL))
 (|qcoerce| (($ (|Integer|)) 8)) (|one?| (((|Boolean|) $) NIL))
 (|min| (($ $ $) NIL)) (|max| (($ $ $) NIL))
 (|latex| (((|String|) $) NIL)) (|hash| (((|SingleInteger|) $) NIL))
 (|gcd| (($ $ $) 11)) (|exquo| (((|Union| $ "failed") $ $) NIL))
 (|divide|
     (((|Record| (|:| |quotient| $) (|:| |remainder| $)) $ $) NIL))
 (|coerce| (((|OutputForm|) $) NIL))
 (^ (($ $ (|NonNegativeInteger|)) NIL) (($ $ (|PositiveInteger|)) NIL))
 (|Zero| (($) NIL T CONST)) (|One| (($) NIL T CONST))
 (>= (((|Boolean|) $ $) NIL)) (> (((|Boolean|) $ $) NIL))
 (= (((|Boolean|) $ $) NIL)) (<= (((|Boolean|) $ $) NIL))
 (< (((|Boolean|) $ $) NIL)) (+ (($ $ $) NIL))
 (** (($ $ (|NonNegativeInteger|)) NIL)
     (($ $ (|PositiveInteger|)) NIL))
 (* (($ (|PositiveInteger|) $) NIL) (($ (|NonNegativeInteger|) $) NIL)
    (($ $ $) NIL)))
MODEMAPS:
getdatabase call: NonNegativeInteger   MODEMAPS
getdatabase miss: NonNegativeInteger   MODEMAPS

((|quo| (*1 *1 *1 *1) (|isDomain| *1 (|NonNegativeInteger|)))
 (|rem| (*1 *1 *1 *1) (|isDomain| *1 (|NonNegativeInteger|)))
 (|gcd| (*1 *1 *1 *1) (|isDomain| *1 (|NonNegativeInteger|)))
 (|divide| (*1 *2 *1 *1)
     (AND (|isDomain| *2
              (|Record| (|:| |quotient| (|NonNegativeInteger|))
                  (|:| |remainder| (|NonNegativeInteger|))))
          (|isDomain| *1 (|NonNegativeInteger|))))
 (|exquo| (*1 *1 *1 *1)
          (|partial| |isDomain| *1 (|NonNegativeInteger|)))
 (|shift| (*1 *1 *1 *2)
          (AND (|isDomain| *2 (|Integer|))
               (|isDomain| *1 (|NonNegativeInteger|))))
 (|random| (*1 *1 *1) (|isDomain| *1 (|NonNegativeInteger|)))
 (|qcoerce| (*1 *1 *2)
     (AND (|isDomain| *2 (|Integer|))
          (|isDomain| *1 (|NonNegativeInteger|)))))getdatabase call: NonNegativeInteger   HASCATEGORY
HASCATEGORY: NIL
getdatabase call: NonNegativeInteger   OBJECT
OBJECT: /mnt/c/Users/markb/EXE/axiom/mnt/ubuntu/algebra/NNI.o
getdatabase call: NonNegativeInteger   NILADIC
NILADIC: T
getdatabase call: NonNegativeInteger   ABBREVIATION
ABBREVIATION: NNI
getdatabase call: NonNegativeInteger   CONSTRUCTOR?
CONSTRUCTOR?: T
getdatabase call: NonNegativeInteger   CONSTRUCTOR
CONSTRUCTOR: NIL
getdatabase call: NonNegativeInteger   DEFAULTDOMAIN
DEFAULTDOMAIN: NIL
getdatabase call: NonNegativeInteger   ANCESTORS
ANCESTORS: NIL
getdatabase call: NonNegativeInteger   SOURCEFILE
SOURCEFILE: bookvol10.3.pamphlet
getdatabase call: NonNegativeInteger   CONSTRUCTORFORM
CONSTRUCTORFORM: (NonNegativeInteger)
getdatabase call: NonNegativeInteger   CONSTRUCTORARGS
getdatabase call: NonNegativeInteger   CONSTRUCTORFORM
CONSTRUCTORARGS: NIL
getdatabase call: NonNegativeInteger   ATTRIBUTES
getdatabase miss: NonNegativeInteger   ATTRIBUTES
ATTRIBUTES: NIL
PREDICATES:
getdatabase call: NonNegativeInteger   PREDICATES
getdatabase miss: NonNegativeInteger   PREDICATES

NILgetdatabase call: NonNegativeInteger   DOCUMENTATION
getdatabase miss: NonNegativeInteger   DOCUMENTATION
DOCUMENTATION: ((constructor (NIL \spadtype{NonNegativeInteger} provides functions for non negative integers.)) (qc
getdatabase call: NonNegativeInteger   PARENTS
PARENTS: NIL
Value = NIL

#S(DATABASE
   ABBREVIATION NNI
   ANCESTORS NIL
   CONSTRUCTOR NIL
   CONSTRUCTORCATEGORY 3449807
   CONSTRUCTORKIND |domain|
   CONSTRUCTORMODEMAP
    (((|NonNegativeInteger|)
      (|Join|
        (|OrderedAbelianMonoidSup|)
        (|Monoid|)
        (CATEGORY
          |domain|
          (SIGNATURE |quo| ($ $ $))
          (SIGNATURE |rem| ($ $ $))
          (SIGNATURE |gcd| ($ $ $))
          (SIGNATURE |divide| ((|Record| (|:| |quotient| $) (|:| |remainder| $)) $ $))
          (SIGNATURE |exquo| ((|Union| $ "failed") $ $))
          (SIGNATURE |shift| ($ $ (|Integer|)))
          (SIGNATURE |random| ($ $))
          (SIGNATURE |qcoerce| ($ (|Integer|))))))
      (T |NonNegativeInteger|))
     COSIG (NIL)
     DEFAULTDOMAIN NIL
     MODEMAPS 3449116
     NILADIC T
     OBJECT "NNI"
     OPERATIONALIST (
       (~= (((|Boolean|) $ $) NIL))
       (|zero?| (((|Boolean|) $) NIL))
       (|sup| (($ $ $) 6))
       (|subtractIfCan| (((|Union| $ "failed") $ $) 10))
       (|shift| (($ $ (|Integer|)) 7))
       (|sample| (($) NIL T CONST))
       (|rem| (($ $ $) NIL))
       (|recip| (((|Union| $ "failed") $) NIL))
       (|random| (($ $) NIL))
       (|quo| (($ $ $) NIL))
       (|qcoerce| (($ (|Integer|)) 8))
       (|one?| (((|Boolean|) $) NIL))
       (|min| (($ $ $) NIL))
       (|max| (($ $ $) NIL))
       (|latex| (((|String|) $) NIL))
       (|hash| (((|SingleInteger|) $) NIL))
       (|gcd| (($ $ $) 11))
       (|exquo| (((|Union| $ "failed") $ $) NIL))
       (|divide| (((|Record| (|:| |quotient| $) (|:| |remainder| $)) $ $) NIL))
       (|coerce| (((|OutputForm|) $) NIL))
       (^ (($ $ (|NonNegativeInteger|)) NIL) (($ $ (|PositiveInteger|)) NIL))
       (|Zero| (($) NIL T CONST))
       (|One| (($) NIL T CONST))
       (>= (((|Boolean|) $ $) NIL))
       (> (((|Boolean|) $ $) NIL))
       (= (((|Boolean|) $ $) NIL))
       (<= (((|Boolean|) $ $) NIL))
       (< (((|Boolean|) $ $) NIL))
       (+ (($ $ $) NIL))
       (** (($ $ (|NonNegativeInteger|)) NIL) (($ $ (|PositiveInteger|)) NIL))
       (* (($ (|PositiveInteger|) $) NIL) (($ (|NonNegativeInteger|) $) NIL) (($ $ $) NIL)))
   DOCUMENTATION 1437684
   CONSTRUCTORFORM (|NonNegativeInteger|)
   ATTRIBUTES 1438930
   PREDICATES 1438935
   SOURCEFILE "bookvol10.3.pamphlet"
   PARENTS NIL
   USERS NIL
   DEPENDENTS NIL
   SPARE NIL)
```

The Domains

1.53 A

1.53.1 AffinePlane

— defclass AffinePlaneType —

```
(defclass |AffinePlaneType| (|AffineSpaceCategoryType|)
  ((parents :initform '(|AffineSpaceCategory|))
   (name :initform "AffinePlane")
   (marker :initform 'domain)
   (abbreviation :initform 'AFFPL)
   (comment :initform (list
     "The following is all the categories and domains related to projective"
     "space and part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffinePlane|
  (progn
    (push '|AffinePlane| *Domains*)
    (make-instance '|AffinePlaneType|)))
```

———-

1.53.2 AffinePlaneOverPseudoAlgebraicClosureOfFiniteField

— defclass AffinePlaneOverPseudoAlgebraicClosureOfFiniteFieldType —

```
(defclass |AffinePlaneOverPseudoAlgebraicClosureOfFiniteFieldType| (|AffineSpaceCategoryType|)
  ((parents :initform '(|AffineSpaceCategory|))
   (name :initform "AffinePlaneOverPseudoAlgebraicClosureOfFiniteField")
   (marker :initform 'domain)
   (abbreviation :initform 'AFFPLPS)
   (comment :initform (list
     "The following is all the categories and domains related to projective"
     "space and part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffinePlaneOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|AffinePlaneOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|AffinePlaneOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.53.3 AffineSpace

— defclass AffineSpaceType —

```
(defclass |AffineSpaceType| (|AffineSpaceCategoryType|)
  ((parents :initform '(|AffineSpaceCategory|))
   (name :initform "AffineSpace")
   (marker :initform 'domain)
   (abbreviation :initform 'AFFSP)
   (comment :initform (list
     "The following is all the categories and domains related to projective"
     "space and part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffineSpace|
  (progn
    (push '|AffineSpace| *Domains*)
    (make-instance '|AffineSpaceType|)))
```

———-

1.53.4 AlgebraGivenByStructuralConstants

— defclass AlgebraGivenByStructuralConstantsType —

```
(defclass |AlgebraGivenByStructuralConstantsType| (|FramedNonAssociativeAlgebraType|)
  ((parents :initform '(|FramedNonAssociativeAlgebra|))
   (name :initform "AlgebraGivenByStructuralConstants")
   (marker :initform 'domain)
   (abbreviation :initform 'ALGSC)
   (comment :initform (list
     "The following is all the categories and domains related to projective"
     "space and part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraGivenByStructuralConstants|
  (progn
    (push '|AlgebraGivenByStructuralConstants| *Domains*)
    (make-instance '|AlgebraGivenByStructuralConstantsType|)))
```

———-

1.53.5 AlgebraicFunctionField

— defclass AlgebraicFunctionFieldType —

```
(defclass |AlgebraicFunctionFieldType| (|FunctionFieldCategoryType|)
  ((parents :initform '(|FunctionFieldCategory|))
   (name :initform "AlgebraicFunctionField")
   (marker :initform 'domain)
   (abbreviation :initform 'ALGFF)
   (comment :initform (list
     "Function field defined by f(x, y) = 0."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicFunctionField|
  (progn
    (push '|AlgebraicFunctionField| *Domains*)
    (make-instance '|AlgebraicFunctionFieldType|)))
```

———-

1.53.6 AlgebraicNumber

— defclass AlgebraicNumberType —

```
(defclass |AlgebraicNumberType| (|AlgebraicallyClosedFieldType|
                                 |CharacteristicZeroType|
                                 |DifferentialRingType|
                                 |ExpressionSpaceType|
                                 |LinearlyExplicitRingOverType|
                                 |RealConstantType|)
  ((parents :initform '(|AlgebraicallyClosedField|
                        |CharacteristicZero|
                        |DifferentialRing|
                        |ExpressionSpace|
                        |LinearlyExplicitRingOver|
                        |RealConstant|))
   (name :initform "AlgebraicNumber")
   (marker :initform 'domain)
   (abbreviation :initform 'AN)
   (comment :initform (list
     "Algebraic closure of the rational numbers, with mathematical ="))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicNumber|
  (progn
    (push '|AlgebraicNumber| *Domains*)
    (make-instance '|AlgebraicNumberType|)))
```

————-

1.53.7 AnonymousFunction

— defclass AnonymousFunctionType —

```
(defclass |AnonymousFunctionType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "AnonymousFunction")
   (marker :initform 'domain)
   (abbreviation :initform 'ANON)
   (comment :initform (list
     "This domain implements anonymous functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnonymousFunction|
  (progn
    (push '|AnonymousFunction| *Domains*)
    (make-instance '|AnonymousFunctionType|)))
```

————-

1.53.8 AntiSymm

— defclass AntiSymmType —

```
(defclass |AntiSymmType| (|RetractableToType| |LeftAlgebraType|)
  ((parents :initform '(|RetractableTo| |LeftAlgebra|))
   (name :initform "AntiSymm")
   (marker :initform 'domain)
   (abbreviation :initform 'ANTISYM)
   (comment :initform (list
     "The domain of antisymmetric polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AntiSymm|
  (progn
    (push '|AntiSymm| *Domains*)
    (make-instance '|AntiSymmType|)))
```

———-

1.53.9 Any

— defclass AnyType —

```
(defclass |AnyType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Any")
   (marker :initform 'domain)
   (abbreviation :initform 'ANY)
   (comment :initform (list
     "Any implements a type that packages up objects and their"
     "types in objects of Any. Roughly speaking that means"
     "that if s : S then when converted to Any, the new"
     "object will include both the original object and its type. This is"
     "a way of converting arbitrary objects into a single type without"
     "losing any of the original information. Any object can be converted"
     "to one of Any."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Any|
  (progn
    (push '|Any| *Domains*)
    (make-instance '|AnyType|)))
```

———-

1.53.10 ArrayStack

— defclass ArrayStackType —

```
(defclass |ArrayStackType| (|StackAggregateType|)
  ((parents :initform '(|StackAggregate|))
   (name :initform "ArrayStack")
   (marker :initform 'domain)
   (abbreviation :initform 'ASTACK)
   (comment :initform (list
     "A stack represented as a flexible array."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ArrayStack|
  (progn
    (push '|ArrayStack| *Domains*)
    (make-instance '|ArrayStackType|)))
```

———-

1.53.11 Asp1

— defclass Asp1Type —

```
(defclass |Asp1Type| (|FortranFunctionCategoryType|)
  ((parents :initform '(|FortranFunctionCategory|))
   (name :initform "Asp1")
   (marker :initform 'domain)
   (abbreviation :initform 'Asp1)
   (comment :initform (list
     "Asp1 produces Fortran for Type 1 ASPs, needed for various"
     "NAG routines. Type 1 ASPs take a univariate expression (in the symbol x)"
     "and turn it into a Fortran Function like the following:"
     " "
     "     DOUBLE PRECISION FUNCTION F(X)"
     "     DOUBLE PRECISION X"
     "     F=DSIN(X)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp1|
  (progn
    (push '|Asp1| *Domains*)
    (make-instance '|Asp1Type|)))
```

———-

1.53.12 Asp10

— defclass Asp10Type —

```
(defclass |Asp10Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp10")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP10)
   (comment :initform (list
     "ASP10 produces Fortran for Type 10 ASPs, needed for NAG routine"
     "d02kef. This ASP computes the values of a set of functions, for example:"
     " "
     "     SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)"
     "     DOUBLE PRECISION ELAM,P,Q,X,DQDL"
     "     INTEGER JINT"
     "     P=1.0D0"
     "     Q=((-1.0D0*X**3)+ELAM*X*X-2.0D0)/(X*X)"
     "     DQDL=1.0D0"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp10|
  (progn
    (push '|Asp10| *Domains*)
    (make-instance '|Asp10Type|)))
```

———-

1.53.13 Asp12

— defclass —Asp12Type —

```
(defclass |Asp12Type| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "Asp12")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP12)
   (comment :initform (list
     "Asp12 produces Fortran for Type 12 ASPs, needed for NAG routine"
     "d02kef etc., for example:"
     " "
     "     SUBROUTINE MONIT (MAXIT,IFLAG,ELAM,FINFO)"
     "     DOUBLE PRECISION ELAM,FINFO(15)"
     "     INTEGER MAXIT,IFLAG"
     "     IF(MAXIT.EQ.-1)THEN"
     "     PRINT*,\"Output from Monit\""
     "     ENDIF"
     "     PRINT*,MAXIT,IFLAG,ELAM,(FINFO(I),I=1,4)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Asp12|
  (progn
    (push '|Asp12| *Domains*)
    (make-instance '|Asp12Type|)))
```

———-

1.53.14 Asp19

— defclass Asp19Type —

```
(defclass |Asp19Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp19")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP19)
   (comment :initform (list
     "Asp19 produces Fortran for Type 19 ASPs, evaluating a set of"
     "functions and their jacobian at a given point, for example:"
     " "
     "      SUBROUTINE LSFUN2(M,N,XC,FVECC,FJACC,LJC)"
     "      DOUBLE PRECISION FVECC(M),FJACC(LJC,N),XC(N)"
     "      INTEGER M,N,LJC"
     "      INTEGER I,J"
     "      DO 25003 I=1,LJC"
     "        DO 25004 J=1,N"
     "          FJACC(I,J)=0.0D0"
     "25004   CONTINUE"
     "25003 CONTINUE"
     "      FVECC(1)=((XC(1)-0.14D0)*XC(3)+(15.0D0*XC(1)-2.1D0)*XC(2)+1.0D0)/("
     "     &XC(3)+15.0D0*XC(2))"
     "      FVECC(2)=((XC(1)-0.18D0)*XC(3)+(7.0D0*XC(1)-1.26D0)*XC(2)+1.0D0)/("
     "     &XC(3)+7.0D0*XC(2))"
     "      FVECC(3)=((XC(1)-0.22D0)*XC(3)+(4.333333333333333D0*XC(1)-0.953333"
     "     &3333333333D0)*XC(2)+1.0D0)/(XC(3)+4.333333333333333D0*XC(2))"
     "      FVECC(4)=((XC(1)-0.25D0)*XC(3)+(3.0D0*XC(1)-0.75D0)*XC(2)+1.0D0)/("
     "     &XC(3)+3.0D0*XC(2))"
     "      FVECC(5)=((XC(1)-0.29D0)*XC(3)+(2.2D0*XC(1)-0.6379999999999999D0)*"
     "     &XC(2)+1.0D0)/(XC(3)+2.2D0*XC(2))"
     "      FVECC(6)=((XC(1)-0.32D0)*XC(3)+(1.666666666666667D0*XC(1)-0.533333"
     "     &3333333333D0)*XC(2)+1.0D0)/(XC(3)+1.666666666666667D0*XC(2))"
     "      FVECC(7)=((XC(1)-0.35D0)*XC(3)+(1.285714285714286D0*XC(1)-0.45D0)*"
     "     &XC(2)+1.0D0)/(XC(3)+1.285714285714286D0*XC(2))"
     "      FVECC(8)=((XC(1)-0.39D0)*XC(3)+(XC(1)-0.39D0)*XC(2)+1.0D0)/(XC(3)+"
     "     &XC(2))"
     "      FVECC(9)=((XC(1)-0.37D0)*XC(3)+(XC(1)-0.37D0)*XC(2)+1.285714285714"
     "     &286D0)/(XC(3)+XC(2))"
     "      FVECC(10)=((XC(1)-0.58D0)*XC(3)+(XC(1)-0.58D0)*XC(2)+1.66666666666"
     "     &6667D0)/(XC(3)+XC(2))"
     "      FVECC(11)=((XC(1)-0.73D0)*XC(3)+(XC(1)-0.73D0)*XC(2)+2.2D0)/(XC(3)"
     "     &+XC(2))"
     "      FVECC(12)=((XC(1)-0.96D0)*XC(3)+(XC(1)-0.96D0)*XC(2)+3.0D0)/(XC(3)"
     "     &+XC(2))"
     "      FVECC(13)=((XC(1)-1.34D0)*XC(3)+(XC(1)-1.34D0)*XC(2)+4.33333333333"
     "     &3333D0)/(XC(3)+XC(2))"
     "      FVECC(14)=((XC(1)-2.1D0)*XC(3)+(XC(1)-2.1D0)*XC(2)+7.0D0)/(XC(3)+X"
     "     &C(2))"
     "      FVECC(15)=((XC(1)-4.39D0)*XC(3)+(XC(1)-4.39D0)*XC(2)+15.0D0)/(XC(3"
     "     &)+XC(2))"
     "      FJACC(1,1)=1.0D0"
     "      FJACC(1,2)=-15.0D0/(XC(3)**2+30.0D0*XC(2)*XC(3)+225.0D0*XC(2)**2)"
     "      FJACC(1,3)=-1.0D0/(XC(3)**2+30.0D0*XC(2)*XC(3)+225.0D0*XC(2)**2)"
     "      FJACC(2,1)=1.0D0"
     "      FJACC(2,2)=-7.0D0/(XC(3)**2+14.0D0*XC(2)*XC(3)+49.0D0*XC(2)**2)"
     "      FJACC(2,3)=-1.0D0/(XC(3)**2+14.0D0*XC(2)*XC(3)+49.0D0*XC(2)**2)"
     "      FJACC(3,1)=1.0D0"
     "      FJACC(3,2)=((-0.1110223024625157D-15*XC(3))-4.333333333333333D0)/("
     "     &XC(3)**2+8.666666666666666D0*XC(2)*XC(3)+18.77777777777778D0*XC(2)"
     "     &**2)"
     "      FJACC(3,3)=(0.1110223024625157D-15*XC(2)-1.0D0)/(XC(3)**2+8.666666"
     "     &666666666D0*XC(2)*XC(3)+18.77777777777778D0*XC(2)**2)"
     "      FJACC(4,1)=1.0D0"
     "      FJACC(4,2)=-3.0D0/(XC(3)**2+6.0D0*XC(2)*XC(3)+9.0D0*XC(2)**2)"
     "      FJACC(4,3)=-1.0D0/(XC(3)**2+6.0D0*XC(2)*XC(3)+9.0D0*XC(2)**2)"
     "      FJACC(5,1)=1.0D0"
     "      FJACC(5,2)=((-0.1110223024625157D-15*XC(3))-2.2D0)/(XC(3)**2+4.399"
     "     &999999999999D0*XC(2)*XC(3)+4.839999999999998D0*XC(2)**2)"
     "      FJACC(5,3)=(0.1110223024625157D-15*XC(2)-1.0D0)/(XC(3)**2+4.399999"
     "     &999999999D0*XC(2)*XC(3)+4.839999999999998D0*XC(2)**2)"
     "      FJACC(6,1)=1.0D0"
     "      FJACC(6,2)=((-0.2220446049250313D-15*XC(3))-1.666666666666667D0)/("
     "     &XC(3)**2+3.333333333333333D0*XC(2)*XC(3)+2.777777777777777D0*XC(2)"
     "     &**2)"
     "      FJACC(6,3)=(0.2220446049250313D-15*XC(2)-1.0D0)/(XC(3)**2+3.333333"
     "     &333333333D0*XC(2)*XC(3)+2.777777777777777D0*XC(2)**2)"
     "      FJACC(7,1)=1.0D0"
     "      FJACC(7,2)=((-0.5551115123125783D-16*XC(3))-1.285714285714286D0)/("
     "     &XC(3)**2+2.571428571428571D0*XC(2)*XC(3)+1.653061224489796D0*XC(2)"
     "     &**2)"
     "      FJACC(7,3)=(0.5551115123125783D-16*XC(2)-1.0D0)/(XC(3)**2+2.571428"
     "     &571428571D0*XC(2)*XC(3)+1.653061224489796D0*XC(2)**2)"
     "      FJACC(8,1)=1.0D0"
     "      FJACC(8,2)=-1.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "      FJACC(8,3)=-1.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "      FJACC(9,1)=1.0D0"
     "      FJACC(9,2)=-1.285714285714286D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)*"
     "     &*2)"
     "      FJACC(9,3)=-1.285714285714286D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)*"
     "     &*2)"
     "      FJACC(10,1)=1.0D0"
     "      FJACC(10,2)=-1.666666666666667D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)"
     "     &**2)"
     "      FJACC(10,3)=-1.666666666666667D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)"
     "     &**2)"
     "      FJACC(11,1)=1.0D0"
     "      FJACC(11,2)=-2.2D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "      FJACC(11,3)=-2.2D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "      FJACC(12,1)=1.0D0"
     "      FJACC(12,2)=-3.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "      FJACC(12,3)=-3.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "      FJACC(13,1)=1.0D0"
     "      FJACC(13,2)=-4.333333333333333D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)"
     "     &**2)"
     "      FJACC(13,3)=-4.333333333333333D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)"
     "     &**2)"
```

```
     "     FJACC(14,1)=1.0D0"
     "     FJACC(14,2)=-7.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "     FJACC(14,3)=-7.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "     FJACC(15,1)=1.0D0"
     "     FJACC(15,2)=-15.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "     FJACC(15,3)=-15.0D0/(XC(3)**2+2.0D0*XC(2)*XC(3)+XC(2)**2)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp19|
  (progn
    (push '|Asp19| *Domains*)
    (make-instance '|Asp19Type|)))
```

———-

1.53.15 Asp20

— defclass Asp20Type —

```
(defclass |Asp20Type| (|FortranMatrixFunctionCategoryType|)
  ((parents :initform '(|FortranMatrixFunctionCategory|))
   (name :initform "Asp20")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP20)
   (comment :initform (list
     "Asp20 produces Fortran for Type 20 ASPs, for example:"
     " "
     "     SUBROUTINE QPHESS(N,NROWH,NCOLH,JTHCOL,HESS,X,HX)"
     "     DOUBLE PRECISION HX(N),X(N),HESS(NROWH,NCOLH)"
     "     INTEGER JTHCOL,N,NROWH,NCOLH"
     "     HX(1)=2.0D0*X(1)"
     "     HX(2)=2.0D0*X(2)"
     "     HX(3)=2.0D0*X(4)+2.0D0*X(3)"
     "     HX(4)=2.0D0*X(4)+2.0D0*X(3)"
     "     HX(5)=2.0D0*X(5)"
     "     HX(6)=(-2.0D0*X(7))+(-2.0D0*X(6))"
     "     HX(7)=(-2.0D0*X(7))+(-2.0D0*X(6))"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp20|
  (progn
    (push '|Asp20| *Domains*)
    (make-instance '|Asp20Type|)))
```

———-

1.53.16 Asp24

— defclass Asp24Type —

```
(defclass |Asp24Type| (|FortranFunctionCategoryType|)
  ((parents :initform '(|FortranFunctionCategory|))
   (name :initform "Asp24")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP24)
   (comment :initform (list
     "Asp24 produces Fortran for Type 24 ASPs which evaluate a"
     "multivariate function at a point (needed for NAG routine e04jaf),"
     "for example:"
     " "
     "     SUBROUTINE FUNCT1(N,XC,FC)"
     "     DOUBLE PRECISION FC,XC(N)"
     "     INTEGER N"
     "     FC=10.0D0*XC(4)**4+(-40.0D0*XC(1)*XC(4)**3)+(60.0D0*XC(1)**2+5"
     "    &.0D0)*XC(4)**2+((-10.0D0*XC(3))+(-40.0D0*XC(1)**3))*XC(4)+16.0D0*X"
     "    &C(3)**4+(-32.0D0*XC(2)*XC(3)**3)+(24.0D0*XC(2)**2+5.0D0)*XC(3)**2+"
     "    &(-8.0D0*XC(2)**3*XC(3))+XC(2)**4+100.0D0*XC(2)**2+20.0D0*XC(1)*XC("
     "    &2)+10.0D0*XC(1)**4+XC(1)**2"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp24|
  (progn
    (push '|Asp24| *Domains*)
    (make-instance '|Asp24Type|)))
```

———-

1.53.17 Asp27

— defclass Asp27Type —

```
(defclass |Asp27Type| (|FortranMatrixCategoryType|)
  ((parents :initform '(|FortranMatrixCategory|))
   (name :initform "Asp27")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP27)
   (comment :initform (list
     "Asp27 produces Fortran for Type 27 ASPs, needed for NAG routine"
     "f02fjf ,for example:"
     " "
     "     FUNCTION DOT(IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK)"
     "     DOUBLE PRECISION W(N),Z(N),RWORK(LRWORK)"
     "     INTEGER N,LIWORK,IFLAG,LRWORK,IWORK(LIWORK)"
     "     DOT=(W(16)+(-0.5D0*W(15)))*Z(16)+((-0.5D0*W(16))+W(15)+(-0.5D0*W(1"
     "    &4)))*Z(15)+((-0.5D0*W(15))+W(14)+(-0.5D0*W(13)))*Z(14)+((-0.5D0*W("
     "    &14))+W(13)+(-0.5D0*W(12)))*Z(13)+((-0.5D0*W(13))+W(12)+(-0.5D0*W(1"
     "    &1)))*Z(12)+((-0.5D0*W(12))+W(11)+(-0.5D0*W(10)))*Z(11)+((-0.5D0*W("
     "    &11))+W(10)+(-0.5D0*W(9)))*Z(10)+((-0.5D0*W(10))+W(9)+(-0.5D0*W(8))"
     "    &)*Z(9)+((-0.5D0*W(9))+W(8)+(-0.5D0*W(7)))*Z(8)+((-0.5D0*W(8))+W(7)"
     "    &+(-0.5D0*W(6)))*Z(7)+((-0.5D0*W(7))+W(6)+(-0.5D0*W(5)))*Z(6)+((-0."
     "    &5D0*W(6))+W(5)+(-0.5D0*W(4)))*Z(5)+((-0.5D0*W(5))+W(4)+(-0.5D0*W(3"
     "    &)))*Z(4)+((-0.5D0*W(4))+W(3)+(-0.5D0*W(2)))*Z(3)+((-0.5D0*W(3))+W("
     "    &2)+(-0.5D0*W(1)))*Z(2)+((-0.5D0*W(2))+W(1))*Z(1)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp27|
  (progn
    (push '|Asp27| *Domains*)
    (make-instance '|Asp27Type|)))
```

———-

1.53.18 Asp28

— defclass Asp28Type —

```
(defclass |Asp28Type| (|FortranMatrixCategoryType|)
  ((parents :initform '(|FortranMatrixCategory|))
   (name :initform "Asp28")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP28)
   (comment :initform (list
     "Asp28 produces Fortran for Type 28 ASPs, used in NAG routine"
     "f02fjf, for example:"
     " "
     "     SUBROUTINE IMAGE(IFLAG,N,Z,W,RWORK,LRWORK,IWORK,LIWORK)"
     "     DOUBLE PRECISION Z(N),W(N),IWORK(LRWORK),RWORK(LRWORK)"
     "     INTEGER N,LIWORK,IFLAG,LRWORK"
     "     W(1)=0.01707454969713436D0*Z(16)+0.001747395874954051D0*Z(15)+0.00"
     "    &2106973900813502D0*Z(14)+0.002957434991769087D0*Z(13)+(-0.00700554"
     "    &0882865317D0*Z(12))+(-0.01219194009813166D0*Z(11))+0.0037230647365"
     "    &3087D0*Z(10)+0.04932374658377151D0*Z(9)+(-0.03586220812223305D0*Z("
     "    &8))+(-0.04723268012114625D0*Z(7))+(-0.02434652144032987D0*Z(6))+0."
     "    &2264766947290192D0*Z(5)+(-0.1385343580686922D0*Z(4))+(-0.116530050"
     "    &8238904D0*Z(3))+(-0.2803531651057233D0*Z(2))+1.019463911841327D0*Z"
     "    &(1)"
     "     W(2)=0.0227345011107737D0*Z(16)+0.008812321197398072D0*Z(15)+0.010"
     "    &94012210519586D0*Z(14)+(-0.01764072463999744D0*Z(13))+(-0.01357136"
     "    &72105995D0*Z(12))+0.00157466157362272D0*Z(11)+0.05258889186338282D"
     "    &0*Z(10)+(-0.01981532388243379D0*Z(9))+(-0.06095390688679697D0*Z(8)"
     "    &)+(-0.04153119955569051D0*Z(7))+0.2176561076571465D0*Z(6)+(-0.0532"
     "     &5555586632358D0*Z(5))+(-0.1688977368984641D0*Z(4))+(-0.32440166056"
     "     &67343D0*Z(3))+0.9128222941872173D0*Z(2)+(-0.2419652703415429D0*Z(1"
     "     &))"
     "      W(3)=0.03371198197190302D0*Z(16)+0.02021603150122265D0*Z(15)+(-0.0"
     "     &06607305534689702D0*Z(14))+(-0.03032392238968179D0*Z(13))+0.002033"
     "     &305231024948D0*Z(12)+0.05375944956767728D0*Z(11)+(-0.0163213312502"
     "     &9967D0*Z(10))+(-0.05483186562035512D0*Z(9))+(-0.04901428822579872D"
     "     &0*Z(8))+0.2091097927887612D0*Z(7)+(-0.05760560341383113D0*Z(6))+(-"
     "     &0.1236679206156403D0*Z(5))+(-0.3523683853026259D0*Z(4))+0.88929961"
     "     &32269974D0*Z(3)+(-0.2995429545781457D0*Z(2))+(-0.02986582812574917"
     "     &D0*Z(1))"
     "      W(4)=0.05141563713660119D0*Z(16)+0.005239165960779299D0*Z(15)+(-0."
     "     &01623427735779699D0*Z(14))+(-0.01965809746040371D0*Z(13))+0.054688"
     "     &97337339577D0*Z(12)+(-0.01422469593568 7D0*Z(11))+(-0.0505181779315"
     "     &6355D0*Z(10))+(-0.04353074206076491D0*Z(9))+0.2012230497530726D0*Z"
     "     &(8)+(-0.06630874514535952D0*Z(7))+(-0.1280829963720053D0*Z(6))+(-0"
     "     &.305169742604165D0*Z(5))+0.8600427128450191D0*Z(4)+(-0.32415033802"
     "     &68184D0*Z(3))+(-0.09033531980693314D0*Z(2))+0.09089205517109111D0*"
     "     &Z(1)"
     "      W(5)=0.04556369767776375D0*Z(16)+(-0.001822737697581869D0*Z(15))+("
     "     &-0.002512226501941856D0*Z(14))+0.02947046460707379D0*Z(13)+(-0.014"
     "     &45079632086177D0*Z(12))+(-0.05034242196614937D0*Z(11))+(-0.0376966"
     "     &3291725935D0*Z(10))+0.2171103102175198D0*Z(9)+(-0.08249492560213 52"
     "     &4D0*Z(8))+(-0.1473995209288945D0*Z(7))+(-0.315042193418466D0*Z(6))"
     "     &+0.9591623347824002D0*Z(5)+(-0.3852396953763045D0*Z(4))+(-0.141718"
     "     &5427288274D0*Z(3))+(-0.03423495461011043D0*Z(2))+0.319820917706851"
     "     &6D0*Z(1)"
     "      W(6)=0.04015147277405744D0*Z(16)+0.01328585741341559D0*Z(15)+0.048"
     "     &2608200546 5965D0*Z(14)+(-0.04319641116207706D0*Z(13))+(-0.04931323"
     "     &319055762D0*Z(12))+(-0.03526886317505474D0*Z(11))+0.22295383396730"
     "     &01D0*Z(10)+(-0.07375317649315155D0*Z(9))+(-0.1589391311991561D0*Z("
     "     &8))+(-0.328001910890377D0*Z(7))+0.952576555482747D0*Z(6)+(-0.31583"
     "     &0997578673 1D0*Z(5))+(-0.1846882042225383D0*Z(4))+(-0.0703762046700"
     "     &4427D0*Z(3))+0.2311852964327382D0*Z(2)+0.04254083491825025D0*Z(1)"
     "      W(7)=0.06069778964023718D0*Z(16)+0.06681263884671322D0*Z(15)+(-0.0"
     "     &2113506688615768D0*Z(14))+(-0.083996867458326D0*Z(13))+(-0.0329843"
     "     &8523869648D0*Z(12))+0.2276878326327734D0*Z(11)+(-0.067356038933017"
     "     &95D0*Z(10))+(-0.1559813965382218D0*Z(9))+(-0.3363262957694705D0*Z("
     "     &8))+0.9442791158560948D0*Z(7)+(-0.3199955249404657D0*Z(6))+(-0.136"
     "     &2463839920727D0*Z(5))+(-0.1006185171570586D0*Z(4))+0.2057504515015"
     "     &423D0*Z(3)+(-0.02065879269286707D0*Z(2))+0.03160990266745513D0*Z(1"
     "     &)"
     "      W(8)=0.126386868896738D0*Z(16)+0.002563370039476418D0*Z(15)+(-0.05"
     "     &581757739455641D0*Z(14))+(-0.07777893205900685D0*Z(13))+0.23117338"
     "     &45834199D0*Z(12)+(-0.06031581134427592D0*Z(11))+(-0.14805474755869"
     "     &52D0*Z(10))+(-0.3364014128402243D0*Z(9))+0.9364014128402244D0*Z(8)"
     "     &+(-0.3269452524413048D0*Z(7))+(-0.1396841886557241D0*Z(6))+(-0.056"
     "     &1733845834199D0*Z(5))+0.1777789320590069D0*Z(4)+(-0.04418242260544"
     "     &359D0*Z(3))+(-0.02756337003947642D0*Z(2))+0.07361313110326199D0*Z("
     "     &1)"
     "      W(9)=0.07361313110326199D0*Z(16)+(-0.02756337003947642D0*Z(15))+(-"
     "     &0.04418242260544359D0*Z(14))+0.1777789320590069D0*Z(13)+(-0.056173"
     "     &3845834199D0*Z(12))+(-0.1396841886557241D0*Z(11))+(-0.326945252441"
     "     &3048D0*Z(10))+0.9364014128402244D0*Z(9)+(-0.3364014128402243D0*Z(8"
     "     &))+(-0.1480547475586952D0*Z(7))+(-0.06031581134427592D0*Z(6))+0.23"
     "     &11733845834199D0*Z(5)+(-0.07777893205900685D0*Z(4))+(-0.0558175773"
     "     &9455641D0*Z(3))+0.002563370039476418D0*Z(2)+0.126386868896738D0*Z("
     "     &1)"
     "      W(10)=0.03160990266745513D0*Z(16)+(-0.02065879269286707D0*Z(15))+0"
```

```
  "    &.2057504515015423D0*Z(14)+(-0.1006185171570586D0*Z(13))+(-0.136246"
  "    &3839920727D0*Z(12))+(-0.3199955249404657D0*Z(11))+0.94427911585609"
  "    &48D0*Z(10)+(-0.3363262957694705D0*Z(9))+(-0.1559813965382218D0*Z(8"
  "    &))+(-0.06735603893301795D0*Z(7))+0.2276878326327734D0*Z(6)+(-0.032"
  "    &98438523869648D0*Z(5))+(-0.083996867458326D0*Z(4))+(-0.02113506688"
  "    &615768D0*Z(3))+0.06681263884671322D0*Z(2)+0.06069778964023718D0*Z("
  "    &1)"
  "     W(11)=0.04254083491825025D0*Z(16)+0.2311852964327382D0*Z(15)+(-0.0"
  "    &7037620467004427D0*Z(14))+(-0.1846882042225383D0*Z(13))+(-0.315830"
  "    &9975786731D0*Z(12))+0.952576555482747D0*Z(11)+(-0.328001910890377D"
  "    &0*Z(10))+(-0.1589391311991561D0*Z(9))+(-0.07375317649315155D0*Z(8)"
  "    &)+0.2229538339673001D0*Z(7)+(-0.03526886317505474D0*Z(6))+(-0.0493"
  "    &1323319055762D0*Z(5))+(-0.04319641116207706D0*Z(4))+0.048260820054"
  "    &65965D0*Z(3)+0.01328585741341559D0*Z(2)+0.04015147277405744D0*Z(1)"
  "     W(12)=0.3198209177068516D0*Z(16)+(-0.03423495461011043D0*Z(15))+(-"
  "    &0.1417185427288274D0*Z(14))+(-0.3852396953763045D0*Z(13))+0.959162"
  "    &3347824002D0*Z(12)+(-0.315042193418466D0*Z(11))+(-0.14739952092889"
  "    &45D0*Z(10))+(-0.08249492560213524D0*Z(9))+0.2171103102175198D0*Z(8"
  "    &)+(-0.03769663291725935D0*Z(7))+(-0.05034242196614937D0*Z(6))+(-0."
  "    &01445079632086177D0*Z(5))+0.02947046460707379D0*Z(4)+(-0.002512226"
  "    &501941856D0*Z(3))+(-0.001822737697581869D0*Z(2))+0.045563697677763"
  "    &75D0*Z(1)"
  "     W(13)=0.09089205517109111D0*Z(16)+(-0.09033531980693314D0*Z(15))+("
  "    &-0.3241503380268184D0*Z(14))+0.8600427128450191D0*Z(13)+(-0.305169"
  "    &742604165D0*Z(12))+(-0.1280829963720053D0*Z(11))+(-0.0663087451453"
  "    &5952D0*Z(10))+0.2012230497530726D0*Z(9)+(-0.04353074206076491D0*Z("
  "    &8))+(-0.05051817793156355D0*Z(7))+(-0.014224695935687D0*Z(6))+0.05"
  "    &468897337339577D0*Z(5)+(-0.01965809746040371D0*Z(4))+(-0.016234277"
  "    &35779699D0*Z(3))+0.005239165960779299D0*Z(2)+0.05141563713660119D0"
  "    &*Z(1)"
  "     W(14)=(-0.02986582812574917D0*Z(16))+(-0.2995429545781457D0*Z(15))"
  "    &+0.8892996132269974D0*Z(14)+(-0.3523683853026259D0*Z(13))+(-0.1236"
  "    &679206156403D0*Z(12))+(-0.05760560341383113D0*Z(11))+0.20910979278"
  "    &87612D0*Z(10)+(-0.04901428822579872D0*Z(9))+(-0.05483186562035512D"
  "    &0*Z(8))+(-0.01632133125029967D0*Z(7))+0.05375944956767728D0*Z(6)+0"
  "    &.002033305231024948D0*Z(5)+(-0.03032392238968179D0*Z(4))+(-0.00660"
  "    &7305534689702D0*Z(3))+0.02021603150122265D0*Z(2)+0.033711981971903"
  "    &02D0*Z(1)"
  "     W(15)=(-0.2419652703415429D0*Z(16))+0.9128222941872173D0*Z(15)+(-0"
  "    &.3244016605667343D0*Z(14))+(-0.1688977368984641D0*Z(13))+(-0.05325"
  "    &555586632358D0*Z(12))+0.2176561076571465D0*Z(11)+(-0.0415311995556"
  "    &9051D0*Z(10))+(-0.06095390688679697D0*Z(9))+(-0.01981532388243379D"
  "    &0*Z(8))+0.05258889186338282D0*Z(7)+0.00157466157362272D0*Z(6)+(-0."
  "    &0135713672105995D0*Z(5))+(-0.01764072463999744D0*Z(4))+0.010940122"
  "    &10519586D0*Z(3)+0.008812321197398072D0*Z(2)+0.0227345011107737D0*Z"
  "    &(1)"
  "     W(16)=1.019463911841327D0*Z(16)+(-0.2803531651057233D0*Z(15))+(-0."
  "    &1165300508238904D0*Z(14))+(-0.1385343580686922D0*Z(13))+0.22647669"
  "    &47290192D0*Z(12)+(-0.02434652144032987D0*Z(11))+(-0.04723268012114"
  "    &625D0*Z(10))+(-0.03586220812223305D0*Z(9))+0.04932374658377151D0*Z"
  "    &(8)+0.00372306473653087D0*Z(7)+(-0.01219194009813166D0*Z(6))+(-0.0"
  "    &07005540882865317D0*Z(5))+0.002957434991769087D0*Z(4)+0.0021069739"
  "    &00813502D0*Z(3)+0.001747395874954051D0*Z(2)+0.01707454969713436D0*"
  "    &Z(1)"
  "     RETURN"
  "     END"))
(argslist :initform nil)
(macros :initform nil)
(withlist :initform nil)
```

```
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp28|
  (progn
    (push '|Asp28| *Domains*)
    (make-instance '|Asp28Type|)))
```

———-

1.53.19 Asp29

— defclass Asp29Type —

```
(defclass |Asp29Type| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "Asp29")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP29)
   (comment :initform (list
     "Asp29 produces Fortran for Type 29 ASPs, needed for NAG routine"
     "f02fjf, for example:"
     " "
     "     SUBROUTINE MONIT(ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D)"
     "     DOUBLE PRECISION D(K),F(K)"
     "     INTEGER K,NEXTIT,NEVALS,NVECS,ISTATE"
     "     CALL F02FJZ(ISTATE,NEXTIT,NEVALS,NEVECS,K,F,D)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp29|
  (progn
    (push '|Asp29| *Domains*)
    (make-instance '|Asp29Type|)))
```

———-

1.53.20 Asp30

— defclass Asp30Type —

```
(defclass |Asp30Type| (|FortranMatrixCategoryType|)
  ((parents :initform '(|FortranMatrixCategory|))
   (name :initform "Asp30")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP30)
   (comment :initform (list
     "Asp30 produces Fortran for Type 30 ASPs, needed for NAG routine"
     "f04qaf, for example:"
     " "
     "     SUBROUTINE APROD(MODE,M,N,X,Y,RWORK,LRWORK,IWORK,LIWORK)"
     "     DOUBLE PRECISION X(N),Y(M),RWORK(LRWORK)"
     "     INTEGER M,N,LIWORK,IFAIL,LRWORK,IWORK(LIWORK),MODE"
     "     DOUBLE PRECISION A(5,5)"
     "     EXTERNAL F06PAF"
     "     A(1,1)=1.0D0"
     "     A(1,2)=0.0D0"
     "     A(1,3)=0.0D0"
     "     A(1,4)=-1.0D0"
     "     A(1,5)=0.0D0"
     "     A(2,1)=0.0D0"
     "     A(2,2)=1.0D0"
     "     A(2,3)=0.0D0"
     "     A(2,4)=0.0D0"
     "     A(2,5)=-1.0D0"
     "     A(3,1)=0.0D0"
     "     A(3,2)=0.0D0"
     "     A(3,3)=1.0D0"
     "     A(3,4)=-1.0D0"
     "     A(3,5)=0.0D0"
     "     A(4,1)=-1.0D0"
     "     A(4,2)=0.0D0"
     "     A(4,3)=-1.0D0"
     "     A(4,4)=4.0D0"
     "     A(4,5)=-1.0D0"
     "     A(5,1)=0.0D0"
     "     A(5,2)=-1.0D0"
     "     A(5,3)=0.0D0"
     "     A(5,4)=-1.0D0"
     "     A(5,5)=4.0D0"
     "     IF(MODE.EQ.1)THEN"
     "       CALL F06PAF('N',M,N,1.0D0,A,M,X,1,1.0D0,Y,1)"
     "     ELSEIF(MODE.EQ.2)THEN"
     "       CALL F06PAF('T',M,N,1.0D0,A,M,Y,1,1.0D0,X,1)"
     "     ENDIF"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp30|
  (progn
    (push '|Asp30| *Domains*)
    (make-instance '|Asp30Type|)))
```

————-

1.53.21 Asp31

— defclass Asp31Type —

```
(defclass |Asp31Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp31")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP31)
   (comment :initform (list
     "Asp31 produces Fortran for Type 31 ASPs, needed for NAG routine"
     "d02ejf, for example:"
     " "
     "     SUBROUTINE PEDERV(X,Y,PW)"
     "     DOUBLE PRECISION X,Y(*)"
     "     DOUBLE PRECISION PW(3,3)"
     "     PW(1,1)=-0.03999999999999999D0"
     "     PW(1,2)=10000.0D0*Y(3)"
     "     PW(1,3)=10000.0D0*Y(2)"
     "     PW(2,1)=0.03999999999999999D0"
     "     PW(2,2)=(-10000.0D0*Y(3))+(-60000000.0D0*Y(2))"
     "     PW(2,3)=-10000.0D0*Y(2)"
     "     PW(3,1)=0.0D0"
     "     PW(3,2)=60000000.0D0*Y(2)"
     "     PW(3,3)=0.0D0"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp31|
  (progn
    (push '|Asp31| *Domains*)
    (make-instance '|Asp31Type|)))
```

———-

1.53.22 Asp33

— defclass Asp33Type —

```
(defclass |Asp33Type| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "Asp33")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP33)
   (comment :initform (list
     "Asp33 produces Fortran for Type 33 ASPs, needed for NAG routine"
     "d02kef.  The code is a dummy ASP:"
     " "
     "     SUBROUTINE REPORT(X,V,JINT)"
     "     DOUBLE PRECISION V(3),X"
     "     INTEGER JINT"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp33|
  (progn
    (push '|Asp33| *Domains*)
    (make-instance '|Asp33Type|)))
```

———-

1.53.23 Asp34

— defclass Asp34Type —

```
(defclass |Asp34Type| (|FortranMatrixCategoryType|)
  ((parents :initform '(|FortranMatrixCategory|))
   (name :initform "Asp34")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP34)
   (comment :initform (list
     "Asp34 produces Fortran for Type 34 ASPs, needed for NAG routine"
     "f04mbf, for example:"
     " "
     "      SUBROUTINE MSOLVE(IFLAG,N,X,Y,RWORK,LRWORK,IWORK,LIWORK)"
     "      DOUBLE PRECISION RWORK(LRWORK),X(N),Y(N)"
     "      INTEGER I,J,N,LIWORK,IFLAG,LRWORK,IWORK(LIWORK)"
     "      DOUBLE PRECISION W1(3),W2(3),MS(3,3)"
     "      IFLAG=-1"
     "      MS(1,1)=2.0D0"
     "      MS(1,2)=1.0D0"
     "      MS(1,3)=0.0D0"
     "      MS(2,1)=1.0D0"
     "      MS(2,2)=2.0D0"
     "      MS(2,3)=1.0D0"
     "      MS(3,1)=0.0D0"
     "      MS(3,2)=1.0D0"
     "      MS(3,3)=2.0D0"
     "      CALL F04ASF(MS,N,X,N,Y,W1,W2,IFLAG)"
     "      IFLAG=-IFLAG"
     "      RETURN"
     "      END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp34|
  (progn
    (push '|Asp34| *Domains*)
    (make-instance '|Asp34Type|)))
```

———-

1.53.24 Asp35

— defclass Asp35Type —

```
(defclass |Asp35Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp35")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP35)
   (comment :initform (list
     "Asp35 produces Fortran for Type 35 ASPs, needed for NAG routines"
     "c05pbf, c05pcf, for example:"
     " "
     "     SUBROUTINE FCN(N,X,FVEC,FJAC,LDFJAC,IFLAG)"
     "     DOUBLE PRECISION X(N),FVEC(N),FJAC(LDFJAC,N)"
     "     INTEGER LDFJAC,N,IFLAG"
     "     IF(IFLAG.EQ.1)THEN"
     "       FVEC(1)=(-1.0D0*X(2))+X(1)"
     "       FVEC(2)=(-1.0D0*X(3))+2.0D0*X(2)"
     "       FVEC(3)=3.0D0*X(3)"
     "     ELSEIF(IFLAG.EQ.2)THEN"
     "       FJAC(1,1)=1.0D0"
     "       FJAC(1,2)=-1.0D0"
     "       FJAC(1,3)=0.0D0"
     "       FJAC(2,1)=0.0D0"
     "       FJAC(2,2)=2.0D0"
     "       FJAC(2,3)=-1.0D0"
     "       FJAC(3,1)=0.0D0"
     "       FJAC(3,2)=0.0D0"
     "       FJAC(3,3)=3.0D0"
     "     ENDIF"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp35|
  (progn
    (push '|Asp35| *Domains*)
    (make-instance '|Asp35Type|)))
```

———-

1.53.25 Asp4

— defclass Asp4Type —

```
(defclass |Asp4Type| (|FortranFunctionCategoryType|)
  ((parents :initform '(|FortranFunctionCategory|))
   (name :initform "Asp4")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP4)
   (comment :initform (list
     "Asp4 produces Fortran for Type 4 ASPs, which take an expression"
     "in X(1) .. X(NDIM) and produce a real function of the form:"
     " "
     "     DOUBLE PRECISION FUNCTION FUNCTN(NDIM,X)"
     "     DOUBLE PRECISION X(NDIM)"
     "     INTEGER NDIM"
     "     FUNCTN=(4.0D0*X(1)*X(3)**2*DEXP(2.0D0*X(1)*X(3)))/(X(4)**2+(2.0D0*"
     "    &X(2)+2.0D0)*X(4)+X(2)**2+2.0D0*X(2)+1.0D0)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp4|
  (progn
    (push '|Asp4| *Domains*)
    (make-instance '|Asp4Type|)))
```

———-

1.53.26 Asp41

— defclass Asp41Type —

```
(defclass |Asp41Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp41")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP41)
   (comment :initform (list
     "Asp41 produces Fortran for Type 41 ASPs, needed for NAG"
     "routines d02raf and d02saf in particular.  These ASPs are in fact"
     "three Fortran routines which return a vector of functions, and their"
     "derivatives wrt Y(i) and also a continuation parameter EPS, for example:"
     " "
     "     SUBROUTINE FCN(X,EPS,Y,F,N)"
     "     DOUBLE PRECISION EPS,F(N),X,Y(N)"
     "     INTEGER N"
     "     F(1)=Y(2)"
     "     F(2)=Y(3)"
     "     F(3)=(-1.0D0*Y(1)*Y(3))+2.0D0*EPS*Y(2)**2+(-2.0D0*EPS)"
     "     RETURN"
     "     END"
     "     SUBROUTINE JACOBF(X,EPS,Y,F,N)"
     "     DOUBLE PRECISION EPS,F(N,N),X,Y(N)"
     "     INTEGER N"
     "     F(1,1)=0.0D0"
     "     F(1,2)=1.0D0"
     "     F(1,3)=0.0D0"
     "     F(2,1)=0.0D0"
     "     F(2,2)=0.0D0"
     "     F(2,3)=1.0D0"
     "     F(3,1)=-1.0D0*Y(3)"
     "     F(3,2)=4.0D0*EPS*Y(2)"
     "     F(3,3)=-1.0D0*Y(1)"
     "     RETURN"
     "     END"
     "     SUBROUTINE JACEPS(X,EPS,Y,F,N)"
     "     DOUBLE PRECISION EPS,F(N),X,Y(N)"
     "     INTEGER N"
     "     F(1)=0.0D0"
     "     F(2)=0.0D0"
     "     F(3)=2.0D0*Y(2)**2-2.0D0"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp41|
  (progn
    (push '|Asp41| *Domains*)
    (make-instance '|Asp41Type|)))
```

———-

1.53.27 Asp42

— defclass Asp42Type —

```
(defclass |Asp42Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp42")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP42)
   (comment :initform (list
     "Asp42 produces Fortran for Type 42 ASPs, needed for NAG"
     "routines d02raf and d02saf"
     "in particular.  These ASPs are in fact"
     "three Fortran routines which return a vector of functions, and their"
     "derivatives wrt Y(i) and also a continuation parameter EPS, for example:"
     " "
     "     SUBROUTINE G(EPS,YA,YB,BC,N)"
     "     DOUBLE PRECISION EPS,YA(N),YB(N),BC(N)"
     "     INTEGER N"
     "     BC(1)=YA(1)"
     "     BC(2)=YA(2)"
     "     BC(3)=YB(2)-1.0D0"
     "     RETURN"
     "     END"
     "     SUBROUTINE JACOBG(EPS,YA,YB,AJ,BJ,N)"
     "     DOUBLE PRECISION EPS,YA(N),AJ(N,N),BJ(N,N),YB(N)"
     "     INTEGER N"
     "     AJ(1,1)=1.0D0"
     "     AJ(1,2)=0.0D0"
     "     AJ(1,3)=0.0D0"
     "     AJ(2,1)=0.0D0"
     "     AJ(2,2)=1.0D0"
     "     AJ(2,3)=0.0D0"
     "     AJ(3,1)=0.0D0"
     "     AJ(3,2)=0.0D0"
     "     AJ(3,3)=0.0D0"
     "     BJ(1,1)=0.0D0"
     "     BJ(1,2)=0.0D0"
     "     BJ(1,3)=0.0D0"
     "     BJ(2,1)=0.0D0"
     "     BJ(2,2)=0.0D0"
     "     BJ(2,3)=0.0D0"
     "     BJ(3,1)=0.0D0"
     "     BJ(3,2)=1.0D0"
     "     BJ(3,3)=0.0D0"
     "     RETURN"
     "     END"
     "     SUBROUTINE JACGEP(EPS,YA,YB,BCEP,N)"
     "     DOUBLE PRECISION EPS,YA(N),YB(N),BCEP(N)"
     "     INTEGER N"
     "     BCEP(1)=0.0D0"
     "     BCEP(2)=0.0D0"
     "     BCEP(3)=0.0D0"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp42|
  (progn
    (push '|Asp42| *Domains*)
    (make-instance '|Asp42Type|)))
```

———-

1.53.28 Asp49

— defclass Asp49Type —

```
(defclass |Asp49Type| (|FortranFunctionCategoryType|)
  ((parents :initform '(|FortranFunctionCategory|))
   (name :initform "Asp49")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP49)
   (comment :initform (list
     "Asp49 produces Fortran for Type 49 ASPs, needed for NAG routines"
     "e04dgf, e04ucf, for example:"
     " "
     "     SUBROUTINE OBJFUN(MODE,N,X,OBJF,OBJGRD,NSTATE,IUSER,USER)"
     "     DOUBLE PRECISION X(N),OBJF,OBJGRD(N),USER(*)"
     "     INTEGER N,IUSER(*),MODE,NSTATE"
     "     OBJF=X(4)*X(9)+((-1.0D0*X(5))+X(3))*X(8)+((-1.0D0*X(3))+X(1))*X(7)"
     "    &+(-1.0D0*X(2)*X(6))"
     "     OBJGRD(1)=X(7)"
     "     OBJGRD(2)=-1.0D0*X(6)"
     "     OBJGRD(3)=X(8)+(-1.0D0*X(7))"
     "     OBJGRD(4)=X(9)"
     "     OBJGRD(5)=-1.0D0*X(8)"
     "     OBJGRD(6)=-1.0D0*X(2)"
     "     OBJGRD(7)=(-1.0D0*X(3))+X(1)"
     "     OBJGRD(8)=(-1.0D0*X(5))+X(3)"
     "     OBJGRD(9)=X(4)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp49|
  (progn
    (push '|Asp49| *Domains*)
    (make-instance '|Asp49Type|)))
```

———-

1.53.29 Asp50

— defclass Asp50Type —

```
(defclass |Asp50Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp50")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP50)
   (comment :initform (list
     "Asp50 produces Fortran for Type 50 ASPs, needed for NAG routine"
     "e04fdf, for example:"
     " "
     "     SUBROUTINE LSFUN1(M,N,XC,FVECC)"
     "     DOUBLE PRECISION FVECC(M),XC(N)"
     "     INTEGER I,M,N"
     "     FVECC(1)=((XC(1)-2.4D0)*XC(3)+(15.0D0*XC(1)-36.0D0)*XC(2)+1.0D0)/("
     "    &XC(3)+15.0D0*XC(2))"
     "     FVECC(2)=((XC(1)-2.8D0)*XC(3)+(7.0D0*XC(1)-19.6D0)*XC(2)+1.0D0)/(X"
     "    &C(3)+7.0D0*XC(2))"
     "     FVECC(3)=((XC(1)-3.2D0)*XC(3)+(4.333333333333333D0*XC(1)-13.866666"
     "    &66666667D0)*XC(2)+1.0D0)/(XC(3)+4.333333333333333D0*XC(2))"
     "     FVECC(4)=((XC(1)-3.5D0)*XC(3)+(3.0D0*XC(1)-10.5D0)*XC(2)+1.0D0)/(X"
     "    &C(3)+3.0D0*XC(2))"
     "     FVECC(5)=((XC(1)-3.9D0)*XC(3)+(2.2D0*XC(1)-8.579999999999998D0)*XC"
     "    &(2)+1.0D0)/(XC(3)+2.2D0*XC(2))"
     "     FVECC(6)=((XC(1)-4.199999999999999D0)*XC(3)+(1.666666666666667D0*X"
     "    &C(1)-7.0D0)*XC(2)+1.0D0)/(XC(3)+1.666666666666667D0*XC(2))"
     "     FVECC(7)=((XC(1)-4.5D0)*XC(3)+(1.285714285714286D0*XC(1)-5.7857142"
     "    &85714286D0)*XC(2)+1.0D0)/(XC(3)+1.285714285714286D0*XC(2))"
     "     FVECC(8)=((XC(1)-4.899999999999999D0)*XC(3)+(XC(1)-4.8999999999999"
     "    &99D0)*XC(2)+1.0D0)/(XC(3)+XC(2))"
```

```
    "     FVECC(9)=((XC(1)-4.699999999999999D0)*XC(3)+(XC(1)-4.6999999999999"
    "    &99D0)*XC(2)+1.285714285714286D0)/(XC(3)+XC(2))"
    "     FVECC(10)=((XC(1)-6.8D0)*XC(3)+(XC(1)-6.8D0)*XC(2)+1.6666666666666"
    "    &67D0)/(XC(3)+XC(2))"
    "     FVECC(11)=((XC(1)-8.299999999999999D0)*XC(3)+(XC(1)-8.299999999999"
    "    &999D0)*XC(2)+2.2D0)/(XC(3)+XC(2))"
    "     FVECC(12)=((XC(1)-10.6D0)*XC(3)+(XC(1)-10.6D0)*XC(2)+3.0D0)/(XC(3)"
    "    &+XC(2))"
    "     FVECC(13)=((XC(1)-1.34D0)*XC(3)+(XC(1)-1.34D0)*XC(2)+4.33333333333"
    "    &3333D0)/(XC(3)+XC(2))"
    "     FVECC(14)=((XC(1)-2.1D0)*XC(3)+(XC(1)-2.1D0)*XC(2)+7.0D0)/(XC(3)+X"
    "    &C(2))"
    "     FVECC(15)=((XC(1)-4.39D0)*XC(3)+(XC(1)-4.39D0)*XC(2)+15.0D0)/(XC(3"
    "    &)+XC(2))"
    "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp50|
  (progn
    (push '|Asp50| *Domains*)
    (make-instance '|Asp50Type|)))
```

————-

1.53.30 Asp55

— defclass Asp55Type —

```
(defclass |Asp55Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp55")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP55)
   (comment :initform (list
    "Asp55 produces Fortran for Type 55 ASPs, needed for NAG routines"
    "e04dgf and e04ucf, for example:"
    " "
    "     SUBROUTINE CONFUN(MODE,NCNLN,N,NROWJ,NEEDC,X,C,CJAC,NSTATE,IUSER"
    "    &,USER)"
    "     DOUBLE PRECISION C(NCNLN),X(N),CJAC(NROWJ,N),USER(*)"
    "     INTEGER N,IUSER(*),NEEDC(NCNLN),NROWJ,MODE,NCNLN,NSTATE"
    "     IF(NEEDC(1).GT.0)THEN"
    "       C(1)=X(6)**2+X(1)**2"
    "       CJAC(1,1)=2.0D0*X(1)"
    "       CJAC(1,2)=0.0D0"
    "       CJAC(1,3)=0.0D0"
    "       CJAC(1,4)=0.0D0"
    "       CJAC(1,5)=0.0D0"
    "       CJAC(1,6)=2.0D0*X(6)"
    "     ENDIF"
    "     IF(NEEDC(2).GT.0)THEN"
    "       C(2)=X(2)**2+(-2.0D0*X(1)*X(2))+X(1)**2"
```

```
     "       CJAC(2,1)=(-2.0D0*X(2))+2.0D0*X(1)"
     "       CJAC(2,2)=2.0D0*X(2)+(-2.0D0*X(1))"
     "       CJAC(2,3)=0.0D0"
     "       CJAC(2,4)=0.0D0"
     "       CJAC(2,5)=0.0D0"
     "       CJAC(2,6)=0.0D0"
     "     ENDIF"
     "     IF(NEEDC(3).GT.0)THEN"
     "       C(3)=X(3)**2+(-2.0D0*X(1)*X(3))+X(2)**2+X(1)**2"
     "       CJAC(3,1)=(-2.0D0*X(3))+2.0D0*X(1)"
     "       CJAC(3,2)=2.0D0*X(2)"
     "       CJAC(3,3)=2.0D0*X(3)+(-2.0D0*X(1))"
     "       CJAC(3,4)=0.0D0"
     "       CJAC(3,5)=0.0D0"
     "       CJAC(3,6)=0.0D0"
     "     ENDIF"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp55|
  (progn
    (push '|Asp55| *Domains*)
    (make-instance '|Asp55Type|)))
```

————-

1.53.31 Asp6

— defclass Asp6Type —

```
(defclass |Asp6Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp6")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP6)
   (comment :initform (list
     "Asp6 produces Fortran for Type 6 ASPs, needed for NAG routines"
     "c05nbf, c05ncf. These represent vectors of functions of X(i) and look like:"
     " "
     "     SUBROUTINE FCN(N,X,FVEC,IFLAG)"
     "     DOUBLE PRECISION X(N),FVEC(N)"
     "     INTEGER N,IFLAG"
     "     FVEC(1)=(-2.0D0*X(2))+(-2.0D0*X(1)**2)+3.0D0*X(1)+1.0D0"
     "     FVEC(2)=(-2.0D0*X(3))+(-2.0D0*X(2)**2)+3.0D0*X(2)+(-1.0D0*X(1))+1."
     "    &0D0"
     "     FVEC(3)=(-2.0D0*X(4))+(-2.0D0*X(3)**2)+3.0D0*X(3)+(-1.0D0*X(2))+1."
     "    &0D0"
     "     FVEC(4)=(-2.0D0*X(5))+(-2.0D0*X(4)**2)+3.0D0*X(4)+(-1.0D0*X(3))+1."
     "    &0D0"
     "     FVEC(5)=(-2.0D0*X(6))+(-2.0D0*X(5)**2)+3.0D0*X(5)+(-1.0D0*X(4))+1."
     "    &0D0"
     "     FVEC(6)=(-2.0D0*X(7))+(-2.0D0*X(6)**2)+3.0D0*X(6)+(-1.0D0*X(5))+1."
     "    &0D0"
     "     FVEC(7)=(-2.0D0*X(8))+(-2.0D0*X(7)**2)+3.0D0*X(7)+(-1.0D0*X(6))+1."
     "    &0D0"
     "     FVEC(8)=(-2.0D0*X(9))+(-2.0D0*X(8)**2)+3.0D0*X(8)+(-1.0D0*X(7))+1."
     "    &0D0"
     "     FVEC(9)=(-2.0D0*X(9)**2)+3.0D0*X(9)+(-1.0D0*X(8))+1.0D0"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp6|
  (progn
    (push '|Asp6| *Domains*)
    (make-instance '|Asp6Type|)))
```

———-

1.53.32 Asp7

— defclass Asp7Type —

```
(defclass |Asp7Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp7")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP7)
   (comment :initform (list
     "Asp7 produces Fortran for Type 7 ASPs, needed for NAG routines"
     "d02bbf, d02gaf. These represent a vector of functions of the scalar X and"
     "the array Z, and look like:"
     " "
     "     SUBROUTINE FCN(X,Z,F)"
     "     DOUBLE PRECISION F(*),X,Z(*)"
     "     F(1)=DTAN(Z(3))"
     "     F(2)=((-0.031999999999999999D0*DCOS(Z(3))*DTAN(Z(3)))+(-0.02D0*Z(2)"
     "    &**2))/(Z(2)*DCOS(Z(3)))"
     "     F(3)=-0.031999999999999999D0/(X*Z(2)**2)"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp7|
  (progn
    (push '|Asp7| *Domains*)
    (make-instance '|Asp7Type|)))
```

———-

1.53.33 Asp73

— defclass Asp73Type —

```
(defclass |Asp73Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp73")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP73)
   (comment :initform (list
     "Asp73 produces Fortran for Type 73 ASPs, needed for NAG routine"
     "d03eef, for example:"
     " "
     "     SUBROUTINE PDEF(X,Y,ALPHA,BETA,GAMMA,DELTA,EPSOLN,PHI,PSI)"
     "     DOUBLE PRECISION ALPHA,EPSOLN,PHI,X,Y,BETA,DELTA,GAMMA,PSI"
     "     ALPHA=DSIN(X)"
     "     BETA=Y"
     "     GAMMA=X*Y"
     "     DELTA=DCOS(X)*DSIN(Y)"
     "     EPSOLN=Y+X"
     "     PHI=X"
     "     PSI=Y"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp73|
  (progn
    (push '|Asp73| *Domains*)
    (make-instance '|Asp73Type|)))
```

———-

1.53.34 Asp74

— defclass Asp74Type —

```
(defclass |Asp74Type| (|FortranMatrixFunctionCategoryType|)
  ((parents :initform '(|FortranMatrixFunctionCategory|))
   (name :initform "Asp74")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP74)
   (comment :initform (list
     "Asp74 produces Fortran for Type 74 ASPs, needed for NAG routine"
     "d03eef, for example:"
     " "
     "     SUBROUTINE BNDY(X,Y,A,B,C,IBND)"
     "     DOUBLE PRECISION A,B,C,X,Y"
     "     INTEGER IBND"
     "     IF(IBND.EQ.0)THEN"
     "       A=0.0D0"
     "       B=1.0D0"
     "       C=-1.0D0*DSIN(X)"
     "     ELSEIF(IBND.EQ.1)THEN"
     "       A=1.0D0"
     "       B=0.0D0"
     "       C=DSIN(X)*DSIN(Y)"
     "     ELSEIF(IBND.EQ.2)THEN"
     "       A=1.0D0"
     "       B=0.0D0"
     "       C=DSIN(X)*DSIN(Y)"
     "     ELSEIF(IBND.EQ.3)THEN"
     "       A=0.0D0"
     "       B=1.0D0"
     "       C=-1.0D0*DSIN(Y)"
     "     ENDIF"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp74|
  (progn
    (push '|Asp74| *Domains*)
    (make-instance '|Asp74Type|)))
```

——————-

1.53.35 Asp77

— defclass Asp77Type —

```
(defclass |Asp77Type| (|FortranMatrixFunctionCategoryType|)
  ((parents :initform '(|FortranMatrixFunctionCategory|))
   (name :initform "Asp77")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP77)
   (comment :initform (list
     "Asp77 produces Fortran for Type 77 ASPs, needed for NAG routine"
     "d02gbf, for example:"
     " "
     "     SUBROUTINE FCNF(X,F)"
     "     DOUBLE PRECISION X"
     "     DOUBLE PRECISION F(2,2)"
     "     F(1,1)=0.0D0"
     "     F(1,2)=1.0D0"
     "     F(2,1)=0.0D0"
     "     F(2,2)=-10.0D0"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp77|
  (progn
    (push '|Asp77| *Domains*)
    (make-instance '|Asp77Type|)))
```

————-

1.53.36 Asp78

— defclass Asp78Type —

```
(defclass |Asp78Type| (|FortranVectorFunctionCategoryType|)
  ((parents :initform '(|FortranVectorFunctionCategory|))
   (name :initform "Asp78")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP78)
   (comment :initform (list
     "Asp78 produces Fortran for Type 78 ASPs, needed for NAG routine"
     "d02gbf, for example:"
     " "
     "     SUBROUTINE FCNG(X,G)"
     "     DOUBLE PRECISION G(*),X"
     "     G(1)=0.0D0"
     "     G(2)=0.0D0"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp78|
  (progn
    (push '|Asp78| *Domains*)
    (make-instance '|Asp78Type|)))
```

————-

1.53.37 Asp8

— defclass Asp8Type —

```
(defclass |Asp8Type| (|FortranVectorCategoryType|)
  ((parents :initform '(|FortranVectorCategory|))
   (name :initform "Asp8")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP8)
   (comment :initform (list
     "Asp8 produces Fortran for Type 8 ASPs, needed for NAG routine"
     "d02bbf.  This ASP prints intermediate values of the computed solution of"
     "an ODE and might look like:"
     " "
     "     SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT,FORWRD)"
     "     DOUBLE PRECISION Y(N),RESULT(M,N),XSOL"
     "     INTEGER M,N,COUNT"
     "     LOGICAL FORWRD"
     "     DOUBLE PRECISION X02ALF,POINTS(8)"
     "     EXTERNAL X02ALF"
     "     INTEGER I"
     "     POINTS(1)=1.0D0"
     "     POINTS(2)=2.0D0"
     "     POINTS(3)=3.0D0"
     "     POINTS(4)=4.0D0"
     "     POINTS(5)=5.0D0"
     "     POINTS(6)=6.0D0"
     "     POINTS(7)=7.0D0"
     "     POINTS(8)=8.0D0"
     "     COUNT=COUNT+1"
     "     DO 25001 I=1,N"
     "        RESULT(COUNT,I)=Y(I)"
     "25001 CONTINUE"
     "     IF(COUNT.EQ.M)THEN"
     "       IF(FORWRD)THEN"
     "         XSOL=X02ALF()"
     "       ELSE"
     "         XSOL=-X02ALF()"
     "       ENDIF"
     "     ELSE"
     "        XSOL=POINTS(COUNT)"
     "     ENDIF"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp8|
  (progn
    (push '|Asp8| *Domains*)
    (make-instance '|Asp8Type|)))
```

———-

1.53.38 Asp80

— defclass Asp80Type —

```
(defclass |Asp80Type| (|FortranMatrixFunctionCategoryType|)
  ((parents :initform '(|FortranMatrixFunctionCategory|))
   (name :initform "Asp80")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP80)
   (comment :initform (list
     "Asp80 produces Fortran for Type 80 ASPs, needed for NAG routine"
     "d02kef, for example:"
     " "
     "     SUBROUTINE BDYVAL(XL,XR,ELAM,YL,YR)"
     "     DOUBLE PRECISION ELAM,XL,YL(3),XR,YR(3)"
     "     YL(1)=XL"
     "     YL(2)=2.0D0"
     "     YR(1)=1.0D0"
     "     YR(2)=-1.0D0*DSQRT(XR+(-1.0D0*ELAM))"
     "     RETURN"
     "     END"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp80|
  (progn
    (push '|Asp80| *Domains*)
    (make-instance '|Asp80Type|)))
```

———-

1.53.39 Asp9

— defclass Asp9Type —

```
(defclass |Asp9Type| (|FortranFunctionCategoryType|)
  ((parents :initform '(|FortranFunctionCategory|))
   (name :initform "Asp9")
   (marker :initform 'domain)
   (abbreviation :initform 'ASP9)
   (comment :initform (list
     "Asp9 produces Fortran for Type 9 ASPs, needed for NAG routines"
     "d02bhf, d02cjf, d02ejf."
     "These ASPs represent a function of a scalar X and a vector Y, for example:"
     " "
     "     DOUBLE PRECISION FUNCTION G(X,Y)"
     "     DOUBLE PRECISION X,Y(*)"
     "     G=X+Y(1)"
     "     RETURN"
     "     END"
     " "
     "If the user provides a constant value for G, then extra information is added"
     "via COMMON blocks used by certain routines.  This specifies that the value"
     "returned by G in this case is to be ignored."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp9|
  (progn
    (push '|Asp9| *Domains*)
    (make-instance '|Asp9Type|)))
```

———-

1.53.40 AssociatedJordanAlgebra

— defclass AssociatedJordanAlgebraType —

```
(defclass |AssociatedJordanAlgebraType| (|FramedNonAssociativeAlgebraType|)
  ((parents :initform '(|FramedNonAssociativeAlgebra|))
   (name :initform "AssociatedJordanAlgebra")
   (marker :initform 'domain)
   (abbreviation :initform 'JORDAN)
   (comment :initform (list
     "AssociatedJordanAlgebra takes an algebra A and uses *$A"
     "to define the new multiplications a*b := (a *$A b + b *$A a)/2"
     "(anticommutator)."
     "The usual notation {a,b}_+ cannot be used due to"
     "restrictions in the current language."
     "This domain only gives a Jordan algebra if the"
     "Jordan-identity (a*b)*c + (b*c)*a + (c*a)*b = 0 holds"
     "for all a,b,c in A."
     "This relation can be checked by"
     "jordanAdmissible?()$A."
     " "
     "If the underlying algebra is of type"
     "FramedNonAssociativeAlgebra(R) (a non"
     "associative algebra over R which is a free R-module of finite"
     "rank, together with a fixed R-module basis), then the same"
     "is true for the associated Jordan algebra."
     "Moreover, if the underlying algebra is of type"
     "FiniteRankNonAssociativeAlgebra(R) (a non"
     "associative algebra over R which is a free R-module of finite"
     "rank), then the same true for the associated Jordan algebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociatedJordanAlgebra|
  (progn
    (push '|AssociatedJordanAlgebra| *Domains*)
    (make-instance '|AssociatedJordanAlgebraType|)))
```

———-

1.53.41 AssociatedLieAlgebra

— defclass AssociatedLieAlgebraType —

```
(defclass |AssociatedLieAlgebraType| (|FramedNonAssociativeAlgebraType|)
  ((parents :initform '(|FramedNonAssociativeAlgebra|))
   (name :initform "AssociatedLieAlgebra")
   (marker :initform 'domain)
   (abbreviation :initform 'LIE)
   (comment :initform (list
     "AssociatedLieAlgebra takes an algebra A"
     "and uses *$A to define the"
     "Lie bracket a*b := (a *$A b - b *$A a) (commutator). Note that"
     "the notation [a,b] cannot be used due to"
     "restrictions of the current compiler."
     "This domain only gives a Lie algebra if the"
     "Jacobi-identity (a*b)*c + (b*c)*a + (c*a)*b = 0 holds"
     "for all a,b,c in A."
     "This relation can be checked by"
     "lieAdmissible?()$A."
     " "
     "If the underlying algebra is of type"
     "FramedNonAssociativeAlgebra(R) (a non"
     "associative algebra over R which is a free R-module of finite"
     "rank, together with a fixed R-module basis), then the same"
     "is true for the associated Lie algebra."
     "Also, if the underlying algebra is of type"
     "FiniteRankNonAssociativeAlgebra(R) (a non"
     "associative algebra over R which is a free R-module of finite"
     "rank), then the same is true for the associated Lie algebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociatedLieAlgebra|
  (progn
    (push '|AssociatedLieAlgebra| *Domains*)
    (make-instance '|AssociatedLieAlgebraType|)))
```

———-

1.53.42 AssociationList

— defclass AssociationListType —

```
(defclass |AssociationListType| (|AssociationListAggregateType|)
  ((parents :initform '(|AssociationListAggregate|))
   (name :initform "AssociationList")
   (marker :initform 'domain)
   (abbreviation :initform 'ALIST)
   (comment :initform (list
     "AssociationList implements association lists. These"
     "may be viewed as lists of pairs where the first part is a key"
     "and the second is the stored value. For example, the key might"
     "be a string with a persons employee identification number and"
     "the value might be a record with personnel data."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociationList|
  (progn
    (push '|AssociationList| *Domains*)
    (make-instance '|AssociationListType|)))
```

———-

1.53.43 AttributeButtons

— defclass AttributeButtonsType —

```
(defclass |AttributeButtonsType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "AttributeButtons")
   (marker :initform 'domain)
   (abbreviation :initform 'ATTRBUT)
   (comment :initform (list
     "AttributeButtons implements a database and associated"
     "adjustment mechanisms for a set of attributes."
     " "
     "For ODEs these attributes are 'stiffness', 'stability' (how much"
     "affect the cosine or sine component of the solution has on the stability of"
     "the result), 'accuracy' and 'expense' (how expensive is the evaluation"
     "of the ODE).  All these have bearing on the cost of calculating the"
     "solution given that reducing the step-length to achieve greater accuracy"
     "requires considerable number of evaluations and calculations."
     " "
     "The effect of each of these attributes can be altered by increasing or"
     "decreasing the button value."
     " "
     "For Integration there is a button for increasing and decreasing the preset"
     "number of function evaluations for each method.  This is automatically used"
     "by ANNA when a method fails due to insufficient workspace or where the"
     "limit of function evaluations has been reached before the required"
     "accuracy is achieved."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AttributeButtons|
  (progn
    (push '|AttributeButtons| *Domains*)
    (make-instance '|AttributeButtonsType|)))
```

———-

1.53.44 Automorphism

— defclass AutomorphismType —

```
(defclass |AutomorphismType| (|EltableType| |GroupType|)
  ((parents :initform '(|Eltable| |Group|))
   (name :initform "Automorphism")
   (marker :initform 'domain)
   (abbreviation :initform 'AUTOMOR)
   (comment :initform (list
     "Automorphism R is the multiplicative group of automorphisms of R."
     "In fact, non-invertible endomorphism are allowed as partial functions."
     "This domain is noncanonical in that f*f^{-1} will be the identity"
     "function but won't be equal to 1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Automorphism|
  (progn
    (push '|Automorphism| *Domains*)
    (make-instance '|AutomorphismType|)))
```

————-

1.54 B

1.54.1 BalancedBinaryTree

— defclass BalancedBinaryTreeType —

```
(defclass |BalancedBinaryTreeType| (|BinaryTreeCategoryType|)
  ((parents :initform '(|BinaryTreeCategory|))
   (name :initform "BalancedBinaryTree")
   (marker :initform 'domain)
   (abbreviation :initform 'BBTREE)
   (comment :initform (list
     "BalancedBinaryTree(S) is the domain of balanced"
     "binary trees (bbtree). A balanced binary tree of 2**k leaves,"
     "for some k > 0, is symmetric, that is, the left and right"
     "subtree of each interior node have identical shape."
     "In general, the left and right subtree of a given node can differ"
     "by at most leaf node."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BalancedBinaryTree|
  (progn
    (push '|BalancedBinaryTree| *Domains*)
    (make-instance '|BalancedBinaryTreeType|)))
```

———-

1.54.2 BalancedPAdicInteger

— defclass BalancedPAdicIntegerType —

```
(defclass |BalancedPAdicIntegerType| (|PAdicIntegerCategoryType|)
  ((parents :initform '(|PAdicIntegerCategory|))
   (name :initform "BalancedPAdicInteger")
   (marker :initform 'domain)
   (abbreviation :initform 'BPADIC)
   (comment :initform (list
     "Stream-based implementation of Zp: p-adic numbers are represented as"
     "sum(i = 0.., a[i] * p^i), where the a[i] lie in -(p - 1)/2,...,(p - 1)/2."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BalancedPAdicInteger|
  (progn
    (push '|BalancedPAdicInteger| *Domains*)
    (make-instance '|BalancedPAdicIntegerType|)))
```

———-

1.54.3 BalancedPAdicRational

— defclass BalancedPAdicRationalType —

```
(defclass |BalancedPAdicRationalType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "BalancedPAdicRational")
   (marker :initform 'domain)
   (abbreviation :initform 'BPADICRT)
   (comment :initform (list
     "Stream-based implementation of Qp: numbers are represented as"
     "sum(i = k.., a[i] * p^i), where the a[i] lie in -(p - 1)/2,...,(p - 1)/2."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BalancedPAdicRational|
  (progn
    (push '|BalancedPAdicRational| *Domains*)
    (make-instance '|BalancedPAdicRationalType|)))
```

———-

1.54.4 BasicFunctions

— defclass BasicFunctionsType —

```
(defclass |BasicFunctionsType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "BasicFunctions")
   (marker :initform 'domain)
   (abbreviation :initform 'BFUNCT)
   (comment :initform (list
     "A Domain which implements a table containing details of"
     "points at which particular functions have evaluation problems."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicFunctions|
  (progn
    (push '|BasicFunctions| *Domains*)
    (make-instance '|BasicFunctionsType|)))
```

———-

1.54.5 BasicOperator

— defclass BasicOperatorType —

```
(defclass |BasicOperatorType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "BasicOperator")
   (marker :initform 'domain)
   (abbreviation :initform 'BOP)
   (comment :initform (list
     "A basic operator is an object that can be applied to a list of"
     "arguments from a set, the result being a kernel over that set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicOperator|
  (progn
    (push '|BasicOperator| *Domains*)
    (make-instance '|BasicOperatorType|)))
```

———-

1.54.6 BasicStochasticDifferential

— defclass BasicStochasticDifferentialType —

```
(defclass |BasicStochasticDifferentialType| (|ConvertibleToType| |OrderedSetType|)
  ((parents :initform '(|ConvertibleTo| |OrderedSet|))
   (name :initform "BasicStochasticDifferential")
   (marker :initform 'domain)
   (abbreviation :initform 'BSD)
   (comment :initform (list
     "Based on Symbol: a domain of symbols representing basic stochastic"
     "differentials, used in StochasticDifferential(R) in the underlying"
     "sparse multivariate polynomial representation."
     " "
     "We create new BSD only by coercion from Symbol using a special"
     "function introduce! first of all to add to a private set SDset."
     "We allow a separate function convertIfCan which will check whether the"
     "argument has previously been declared as a BSD."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicStochasticDifferential|
  (progn
    (push '|BasicStochasticDifferential| *Domains*)
    (make-instance '|BasicStochasticDifferentialType|)))
```

———-

1.54.7 BinaryExpansion

— defclass BinaryExpansionType —

```
(defclass |BinaryExpansionType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "BinaryExpansion")
   (marker :initform 'domain)
   (abbreviation :initform 'BINARY)
   (comment :initform (list
     "This domain allows rational numbers to be presented as repeating"
     "binary expansions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryExpansion|
  (progn
    (push '|BinaryExpansion| *Domains*)
    (make-instance '|BinaryExpansionType|)))
```

———-

1.54.8 BinaryFile

— defclass BinaryFileType —

```
(defclass |BinaryFileType| (|FileCategoryType|)
  ((parents :initform '(|FileCategory|))
   (name :initform "BinaryFile")
   (marker :initform 'domain)
   (abbreviation :initform 'BINFILE)
   (comment :initform (list
     "This domain provides an implementation of binary files. Data is"
     "accessed one byte at a time as a small integer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryFile|
  (progn
    (push '|BinaryFile| *Domains*)
    (make-instance '|BinaryFileType|)))
```

———-

1.54.9 BinarySearchTree

— defclass BinarySearchTreeType —

```
(defclass |BinarySearchTreeType| (|BinaryTreeCategoryType|)
  ((parents :initform '(|BinaryTreeCategory|))
   (name :initform "BinarySearchTree")
   (marker :initform 'domain)
   (abbreviation :initform 'BSTREE)
   (comment :initform (list
     "BinarySearchTree(S) is the domain of"
     "a binary trees where elements are ordered across the tree."
     "A binary search tree is either empty or has"
     "a value which is an S, and a"
     "right and left which are both BinaryTree(S)"
     "Elements are ordered across the tree."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinarySearchTree|
  (progn
    (push '|BinarySearchTree| *Domains*)
    (make-instance '|BinarySearchTreeType|)))
```

———-

1.54.10 BinaryTournament

— defclass BinaryTournamentType —

```
(defclass |BinaryTournamentType| (|BinaryTreeCategoryType|)
  ((parents :initform '(|BinaryTreeCategory|))
   (name :initform "BinaryTournament")
   (marker :initform 'domain)
   (abbreviation :initform 'BTOURN)
   (comment :initform (list
     "BinaryTournament creates a binary tournament with the"
     "elements of ls as values at the nodes."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryTournament|
  (progn
    (push '|BinaryTournament| *Domains*)
    (make-instance '|BinaryTournamentType|)))
```

———-

1.54.11 BinaryTree

— defclass BinaryTreeType —

```
(defclass |BinaryTreeType| (|BinaryTreeCategoryType|)
  ((parents :initform '(|BinaryTreeCategory|))
   (name :initform "BinaryTree")
   (marker :initform 'domain)
   (abbreviation :initform 'BTREE)
   (comment :initform (list
     "BinaryTree(S) is the domain of all"
     "binary trees. A binary tree over S is either empty or has"
     "a value which is an S and a right"
     "and left which are both binary trees."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryTree|
  (progn
    (push '|BinaryTree| *Domains*)
    (make-instance '|BinaryTreeType|)))
```

———-

1.54.12 Bits

— defclass BitsType —

```
(defclass |BitsType| (|BitAggregateType|)
  ((parents :initform '(|BitAggregate|))
   (name :initform "Bits")
   (marker :initform 'domain)
   (abbreviation :initform 'BITS)
   (comment :initform (list
     "Bits provides logical functions for Indexed Bits."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Bits|
  (progn
    (push '|Bits| *Domains*)
    (make-instance '|BitsType|)))
```

———-

1.54.13 BlowUpWithHamburgerNoether

— defclass BlowUpWithHamburgerNoetherType —

```
(defclass |BlowUpWithHamburgerNoetherType| (|BlowUpMethodCategoryType|)
  ((parents :initform '(|BlowUpMethodCategory|))
   (name :initform "BlowUpWithHamburgerNoether")
   (marker :initform 'domain)
   (abbreviation :initform 'BLHN)
   (comment :initform (list
     "This domain is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BlowUpWithHamburgerNoether|
  (progn
    (push '|BlowUpWithHamburgerNoether| *Domains*)
    (make-instance '|BlowUpWithHamburgerNoetherType|)))
```

———-

1.54.14 BlowUpWithQuadTrans

— defclass BlowUpWithQuadTransType —

```
(defclass |BlowUpWithQuadTransType| (|BlowUpMethodCategoryType|)
  ((parents :initform '(|BlowUpMethodCategory|))
   (name :initform "BlowUpWithQuadTrans")
   (marker :initform 'domain)
   (abbreviation :initform 'BLQT)
   (comment :initform (list
     "This domain is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BlowUpWithQuadTrans|
  (progn
    (push '|BlowUpWithQuadTrans| *Domains*)
    (make-instance '|BlowUpWithQuadTransType|)))
```

———-

1.54.15 Boolean

— defclass BooleanType —

```
(defclass |BooleanType| (|ConvertibleToType| |FiniteType| |LogicType| |OrderedSetType|)
  ((parents :initform '(|ConvertibleTo| |Finite| |Logic| |OrderedSet|))
   (name :initform "Boolean")
   (marker :initform 'domain)
   (abbreviation :initform 'BOOLEAN)
   (comment :initform (list
     "Boolean is the elementary logic with 2 values:"
     "true and false"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Boolean|
  (progn
    (push '|Boolean| *Domains*)
    (make-instance '|BooleanType|)))
```

———-

1.55 C

1.55.1 CardinalNumber

— defclass CardinalNumberType —

```
(defclass |CardinalNumberType| (|AbelianMonoidType| |MonoidType| |OrderedSetType| |RetractableToType|)
  ((parents :initform '(|AbelianMonoid| |Monoid| |OrderedSet| |RetractableTo|))
   (name :initform "CardinalNumber")
   (marker :initform 'domain)
   (abbreviation :initform 'CARD)
   (comment :initform (list
     "Members of the domain CardinalNumber are values indicating the"
     "cardinality of sets, both finite and infinite.  Arithmetic operations"
     "are defined on cardinal numbers as follows."
     " "
     "If x = #X and y = #Y then"
     "   x+y  = #(X+Y) disjoint union"
     "   x-y  = #(X-Y) relative complement"
     "   x*y  = #(X*Y) cartesian product"
     "   x**y = #(X**Y) X**Y = g \| g:Y->X"
     " "
     "The non-negative integers have a natural construction as cardinals"
     "  0 = #{}, 1 = {0},"
     " 2 = {0, 1}}, ..., n = {i\| 0 <= i < n}."
     " "
     "That 0 acts as a zero for the multiplication of cardinals is"
     "equivalent to the axiom of choice."
     " "
     "The generalized continuum hypothesis asserts"
     "2**Aleph i = Aleph(i+1)"
     "and is independent of the axioms of set theory [Goedel 1940]"
     " "
     "Three commonly encountered cardinal numbers are"
     "  a = #Z countable infinity"
     "  c = #R the continuum"
     "  f = # g \| g:[0,1]->R"
     " "
     "In this domain, these values are obtained using"
     "  a := Aleph 0, c := 2**a, f := 2**c."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CardinalNumber|
  (progn
    (push '|CardinalNumber| *Domains*)
    (make-instance '|CardinalNumberType|)))
```

———-

1.55.2 CartesianTensor

— defclass CartesianTensorType —

```
(defclass |CartesianTensorType| (|GradedAlgebraType|)
  ((parents :initform '(|GradedAlgebra|))
   (name :initform "CartesianTensor")
   (marker :initform 'domain)
   (abbreviation :initform 'CARTEN)
   (comment :initform (list
     "CartesianTensor(minix,dim,R) provides Cartesian tensors with"
     "components belonging to a commutative ring R.  These tensors"
     "can have any number of indices.  Each index takes values from"
     "minix to minix + dim - 1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CartesianTensor|
  (progn
    (push '|CartesianTensor| *Domains*)
    (make-instance '|CartesianTensorType|)))
```

———-

1.55.3 Cell

— defclass CellType —

```
(defclass |CellType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "Cell")
   (marker :initform 'domain)
   (abbreviation :initform 'CELL)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Cell|
  (progn
    (push '|Cell| *Domains*)
    (make-instance '|CellType|)))
```

———-

1.55.4 Character

— defclass CharacterType —

```
(defclass |CharacterType| (|OrderedFiniteType|)
  ((parents :initform '(|OrderedFinite|))
   (name :initform "Character")
   (marker :initform 'domain)
   (abbreviation :initform 'CHAR)))

(defvar |Character|
  (progn
    (push '|Character| *Domains*)
    (make-instance '|CharacterType|)))
```

————-

1.55.5 CharacterClass

— defclass CharacterClassType —

```
(defclass |CharacterClassType| (|FiniteSetAggregateType|)
  ((parents :initform '(|FiniteSetAggregate|))
   (name :initform "CharacterClass")
   (marker :initform 'domain)
   (abbreviation :initform 'CCLASS)
   (comment :initform (list
     "This domain allows classes of characters to be defined and manipulated"
     "efficiently."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacterClass|
  (progn
    (push '|CharacterClass| *Domains*)
    (make-instance '|CharacterClassType|)))
```

————-

1.55.6 CliffordAlgebra

— defclass CliffordAlgebraType —

```
(defclass |CliffordAlgebraType| (|VectorSpaceType| |AlgebraType|)
  ((parents :initform '(|VectorSpace| |Algebra|))
   (name :initform "CliffordAlgebra")
   (marker :initform 'domain)
   (abbreviation :initform 'CLIF)
   (comment :initform (list
     "CliffordAlgebra(n, K, Q) defines a vector space of dimension 2**n"
     "over K, given a quadratic form Q on K**n."
     " "
     "If e[i], 1<=i<=n is a basis for K**n then"
     "   1, e[i] (1<=i<=n), e[i1]*e[i2]"
     "   (1<=i1<i2<=n},...,e[1]*e[2]*..*e[n]"
     "is a basis for the Clifford Algebra."
     " "
     "The algebra is defined by the relations"
     "   e[i]*e[j] = -e[j]*e[i]}  (i \~~= j),"
     "   e[i]*e[i] = Q(e[i])"
     " "
     "Examples of Clifford Algebras are: gaussians, quaternions, exterior"
     "algebras and spin algebras."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CliffordAlgebra|
  (progn
    (push '|CliffordAlgebra| *Domains*)
    (make-instance '|CliffordAlgebraType|)))
```

————-

1.55.7 Color

— defclass ColorType —

```
(defclass |ColorType| (|AbelianSemiGroupType|)
  ((parents :initform '(|AbelianSemiGroup|))
   (name :initform "Color")
   (marker :initform 'domain)
   (abbreviation :initform 'COLOR)
   (comment :initform (list
     "Color() specifies a domain of 27 colors provided in the"
     "Axiom system (the colors mix additively)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Color|
  (progn
    (push '|Color| *Domains*)
    (make-instance '|ColorType|)))
```

————-

1.55.8 Commutator

— defclass CommutatorType —

```
(defclass |CommutatorType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Commutator")
   (marker :initform 'domain)
   (abbreviation :initform 'COMM)
   (comment :initform (list
     "A type for basic commutators"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Commutator|
  (progn
    (push '|Commutator| *Domains*)
    (make-instance '|CommutatorType|)))
```

———-

1.55.9 Complex

— defclass ComplexType —

```
(defclass |ComplexType| (|ComplexCategoryType| |OpenMathType|)
  ((parents :initform '(|ComplexCategory| |OpenMath|))
   (name :initform "Complex")
   (marker :initform 'domain)
   (abbreviation :initform 'COMPLEX)
   (comment :initform (list
     "Complex(R) creates the domain of elements of the form"
     "a + b * i where a and b come from the ring R,"
     "and i is a new element such that i**2 = -1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Complex|
  (progn
    (push '|Complex| *Domains*)
    (make-instance '|ComplexType|)))
```

———-

1.55.10 ComplexDoubleFloatMatrix

— defclass ComplexDoubleFloatMatrixType —

```
(defclass |ComplexDoubleFloatMatrixType| (|MatrixCategoryType|)
  ((parents :initform '(|MatrixCategory|))
   (name :initform "ComplexDoubleFloatMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'CDFMAT)
   (comment :initform (list
     "This is a low-level domain which implements matrices"
     "(two dimensional arrays) of complex double precision floating point"
     "numbers.  Indexing is 0 based, there is no bound checking (unless"
    "provided by lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexDoubleFloatMatrix|
  (progn
    (push '|ComplexDoubleFloatMatrix| *Domains*)
    (make-instance '|ComplexDoubleFloatMatrixType|)))
```

———-

1.55.11 ComplexDoubleFloatVector

— defclass ComplexDoubleFloatVectorType —

```
(defclass |ComplexDoubleFloatVectorType| (|VectorCategoryType|)
  ((parents :initform '(|VectorCategory|))
   (name :initform "ComplexDoubleFloatVector")
   (marker :initform 'domain)
   (abbreviation :initform 'CDFVEC)
   (comment :initform (list
     "This is a low-level domain which implements vectors"
     "(one dimensional arrays) of complex double precision floating point"
     "numbers.  Indexing is 0 based, there is no bound checking (unless"
     "provided by lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexDoubleFloatVector|
  (progn
    (push '|ComplexDoubleFloatVector| *Domains*)
    (make-instance '|ComplexDoubleFloatVectorType|)))
```

———-

1.55.12 ContinuedFraction

— defclass ContinuedFractionType —

```
(defclass |ContinuedFractionType| (|FieldType|)
  ((parents :initform '(|Field|))
   (name :initform "ContinuedFraction")
   (marker :initform 'domain)
   (abbreviation :initform 'CONTFRAC)
   (comment :initform (list
     "ContinuedFraction implements general"
     "continued fractions.  This version is not restricted to simple,"
     "finite fractions and uses the Stream as a"
     "representation.  The arithmetic functions assume that the"
     "approximants alternate below/above the convergence point."
     "This is enforced by ensuring the partial numerators and partial"
     "denominators are greater than 0 in the Euclidean domain view of R"
     "(sizeLess?(0, x)).")
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ContinuedFraction|
  (progn
    (push '|ContinuedFraction| *Domains*)
    (make-instance '|ContinuedFractionType|)))
```

———-

1.56 D

1.56.1 Database

— defclass DatabaseType —

```
(defclass |DatabaseType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Database")
   (marker :initform 'domain)
   (abbreviation :initform 'DBASE)
   (comment :initform (list
     "This domain implements a simple view of a database whose fields are"
     "indexed by symbols"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Database|
  (progn
    (push '|Database| *Domains*)
    (make-instance '|DatabaseType|)))
```

———-

1.56.2 DataList

— defclass DataListType —

```
(defclass |DataListType| (|ListAggregateType|)
  ((parents :initform '(|ListAggregate|))
   (name :initform "DataList")
   (marker :initform 'domain)
   (abbreviation :initform 'DLIST)
   (comment :initform (list
     "This domain provides some nice functions on lists"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DataList|
  (progn
    (push '|DataList| *Domains*)
    (make-instance '|DataListType|)))
```

————-

1.56.3 DecimalExpansion

— defclass DecimalExpansionType —

```
(defclass |DecimalExpansionType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "DecimalExpansion")
   (marker :initform 'domain)
   (abbreviation :initform 'DECIMAL)
   (comment :initform (list
     "This domain allows rational numbers to be presented as repeating"
     "decimal expansions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DecimalExpansion|
  (progn
    (push '|DecimalExpansion| *Domains*)
    (make-instance '|DecimalExpansionType|)))
```

————-

1.56.4 DenavitHartenbergMatrix

— defclass DenavitHartenbergMatrixType —

```
(defclass |DenavitHartenbergMatrixType| (|MatrixCategoryType|)
  ((parents :initform '(|MatrixCategory|))
   (name :initform "DenavitHartenbergMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'DHMATRIX)
   (comment :initform (list
     "4x4 Matrices for coordinate transformations"
     "This package contains functions to create 4x4 matrices"
     "useful for rotating and transforming coordinate systems."
     "These matrices are useful for graphics and robotics."
     "(Reference: Robot Manipulators Richard Paul MIT Press 1981)"
     " "
     "A Denavit-Hartenberg Matrix is a 4x4 Matrix of the form:"
     "     nx ox ax px"
     "     ny oy ay py"
     "     nz oz az pz"
     "     0  0  0  1"
     "(n, o, and a are the direction cosines)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DenavitHartenbergMatrix|
  (progn
    (push '|DenavitHartenbergMatrix| *Domains*)
    (make-instance '|DenavitHartenbergMatrixType|)))
```

———-

1.56.5 Dequeue

— defclass DequeueType —

```
(defclass |DequeueType| (|DequeueAggregateType|)
  ((parents :initform '(|DequeueAggregate|))
   (name :initform "Dequeue")
   (marker :initform 'domain)
   (abbreviation :initform 'DEQUEUE)
   (comment :initform (list
     "Linked list implementation of a Dequeue"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Dequeue|
  (progn
    (push '|Dequeue| *Domains*)
    (make-instance '|DequeueType|)))
```

———-

1.56.6 DeRhamComplex

— defclass DeRhamComplexType —

```
(defclass |DeRhamComplexType| (|RetractableToType| |LeftAlgebraType|)
  ((parents :initform '(|RetractableTo| |LeftAlgebra|))
   (name :initform "DeRhamComplex")
   (marker :initform 'domain)
   (abbreviation :initform 'DERHAM)
   (comment :initform (list
     "The deRham complex of Euclidean space, that is, the"
     "class of differential forms of arbitary degree over a coefficient ring."
     "See Flanders, Harley, Differential Forms, With Applications to the Physical"
     "Sciences, New York, Academic Press, 1963."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DeRhamComplex|
  (progn
    (push '|DeRhamComplex| *Domains*)
    (make-instance '|DeRhamComplexType|)))
```

———-

1.56.7 DesingTree

— defclass DesingTreeType —

```
(defclass |DesingTreeType| (|DesingTreeCategoryType|)
  ((parents :initform '(|DesingTreeCategory|))
   (name :initform "DesingTree")
   (marker :initform 'domain)
   (abbreviation :initform 'DSTREE)
   (comment :initform (list
     "This category is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DesingTree|
  (progn
    (push '|DesingTree| *Domains*)
    (make-instance '|DesingTreeType|)))
```

———-

1.56.8 DifferentialSparseMultivariatePolynomial

— defclass DifferentialSparseMultivariatePolynomialType —

```
(defclass |DifferentialSparseMultivariatePolynomialType| (|DifferentialPolynomialCategoryType|)
  ((parents :initform '(|DifferentialPolynomialCategory|))
   (name :initform "DifferentialSparseMultivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'DSMP)
   (comment :initform (list
     "DifferentialSparseMultivariatePolynomial implements"
     "an ordinary differential polynomial ring by combining a"
     "domain belonging to the category DifferentialVariableCategory"
     "with the domain SparseMultivariatePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialSparseMultivariatePolynomial|
  (progn
    (push '|DifferentialSparseMultivariatePolynomial| *Domains*)
    (make-instance '|DifferentialSparseMultivariatePolynomialType|)))
```

————-

1.56.9 DirectProduct

— defclass DirectProductType —

```
(defclass |DirectProductType| (|DirectProductCategoryType|)
  ((parents :initform '(|DirectProductCategory|))
   (name :initform "DirectProduct")
   (marker :initform 'domain)
   (abbreviation :initform 'DIRPROD)
   (comment :initform (list
     "This type represents the finite direct or cartesian product of an"
     "underlying component type. This contrasts with simple vectors in that"
     "the members can be viewed as having constant length. Thus many"
     "categorical properties can by lifted from the underlying component type."
     "Component extraction operations are provided but no updating operations."
     "Thus new direct product elements can either be created by converting"
     "vector elements using the directProduct function"
     "or by taking appropriate linear combinations of basis vectors provided"
     "by the unitVector operation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProduct|
  (progn
    (push '|DirectProduct| *Domains*)
    (make-instance '|DirectProductType|)))
```

———-

1.56.10 DirectProductMatrixModule

— defclass DirectProductMatrixModuleType —

```
(defclass |DirectProductMatrixModuleType| (|DirectProductCategoryType|)
  ((parents :initform '(|DirectProductCategory|))
   (name :initform "DirectProductMatrixModule")
   (marker :initform 'domain)
   (abbreviation :initform 'DPMM)
   (comment :initform (list
     "This constructor provides a direct product type with a"
     "left matrix-module view."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductMatrixModule|
  (progn
    (push '|DirectProductMatrixModule| *Domains*)
    (make-instance '|DirectProductMatrixModuleType|)))
```

———-

1.56.11 DirectProductModule

— defclass DirectProductModuleType —

```
(defclass |DirectProductModuleType| (|DirectProductCategoryType|)
  ((parents :initform '(|DirectProductCategory|))
   (name :initform "DirectProductModule")
   (marker :initform 'domain)
   (abbreviation :initform 'DPMO)
   (comment :initform (list
     "This constructor provides a direct product of R-modules"
     "with an R-module view."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductModule|
  (progn
    (push '|DirectProductModule| *Domains*)
    (make-instance '|DirectProductModuleType|)))
```

———-

1.56.12 DirichletRing

— defclass DirichletRingType —

```
(defclass |DirichletRingType| (|IntegralDomainType| |EltableType|)
  ((parents :initform '(|IntegralDomain| |Eltable|))
   (name :initform "DirichletRing")
   (marker :initform 'domain)
   (abbreviation :initform 'DIRRING)
   (comment :initform (list
     "DirichletRing is the ring of arithmetical functions"
     "with Dirichlet convolution as multiplication"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirichletRing|
  (progn
    (push '|DirichletRing| *Domains*)
    (make-instance '|DirichletRingType|)))
```

———-

1.56.13 DistributedMultivariatePolynomial

— defclass DistributedMultivariatePolynomialType —

```
(defclass |DistributedMultivariatePolynomialType| (|PolynomialCategoryType|)
  ((parents :initform '(|PolynomialCategory|))
   (name :initform "DistributedMultivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'DMP)
   (comment :initform (list
     "This type supports distributed multivariate polynomials"
     "whose variables are from a user specified list of symbols."
     "The coefficient ring may be non commutative,"
     "but the variables are assumed to commute."
     "The term ordering is lexicographic specified by the variable"
     "list parameter with the most significant variable first in the list."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DistributedMultivariatePolynomial|
  (progn
    (push '|DistributedMultivariatePolynomial| *Domains*)
    (make-instance '|DistributedMultivariatePolynomialType|)))
```

———-

1.56.14 Divisor

— defclass DivisorType —

```
(defclass |DivisorType| (|DivisorCategoryType|)
  ((parents :initform '(|DivisorCategory|))
   (name :initform "Divisor")
   (marker :initform 'domain)
   (abbreviation :initform 'DIV)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Divisor|
  (progn
    (push '|Divisor| *Domains*)
    (make-instance '|DivisorType|)))
```

———-

1.56.15 DoubleFloat

— defclass DoubleFloatType —

```
(defclass |DoubleFloatType| (|TranscendentalFunctionCategoryType|
                             |SpecialFunctionCategoryType|
                             |OpenMathType|
                             |FloatingPointSystemType|
                             |DifferentialRingType|)
  ((parents :initform '(|TranscendentalFunctionCategory|
                        |SpecialFunctionCategory|
                        |OpenMath|
                        |FloatingPointSystem|
                        |DifferentialRing|))
   (name :initform "DoubleFloat")
   (marker :initform 'domain)
   (abbreviation :initform 'DFLOAT)
   (comment :initform (list
     "DoubleFloat is intended to make accessible"
     "hardware floating point arithmetic in Axiom, either native double"
     "precision, or IEEE. On most machines, there will be hardware support for"
     "the arithmetic operations: +, *, / and possibly also the"
     "sqrt operation."
     "The operations exp, log, sin, cos, atan are normally coded in"
     "software based on minimax polynomial/rational approximations."
     " "
     "Some general comments about the accuracy of the operations:"
     "the operations +, *, / and sqrt are expected to be fully accurate."
     "The operations exp, log, sin, cos and atan are not expected to be"
     "fully accurate.  In particular, sin and cos"
     "will lose all precision for large arguments."
     " "
     "The Float domain provides an alternative to the DoubleFloat domain."
     "It provides an arbitrary precision model of floating point arithmetic."
     "This means that accuracy problems like those above are eliminated"
     "by increasing the working precision where necessary. Float"
     "provides some special functions such as erf, the error function"
     "in addition to the elementary functions.  The disadvantage of Float is that"
     "it is much more expensive than small floats when the latter can be used."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleFloat|
  (progn
    (push '|DoubleFloat| *Domains*)
    (make-instance '|DoubleFloatType|)))
```

———-

1.56.16 DoubleFloatMatrix

— defclass DoubleFloatMatrixType —

```
(defclass |DoubleFloatMatrixType| (|MatrixCategoryType|)
  ((parents :initform '(|MatrixCategory|))
   (name :initform "DoubleFloatMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'DFMAT)
   (comment :initform (list
     "This is a low-level domain which implements matrices"
     "(two dimensional arrays) of double precision floating point"
     "numbers.  Indexing is 0 based, there is no bound checking (unless"
     "provided by lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleFloatMatrix|
  (progn
    (push '|DoubleFloatMatrix| *Domains*)
    (make-instance '|DoubleFloatMatrixType|)))
```

———-

1.56.17 DoubleFloatVector

— defclass DoubleFloatVectorType —

```
(defclass |DoubleFloatVectorType| (|VectorCategoryType|)
  ((parents :initform '(|VectorCategory|))
   (name :initform "DoubleFloatVector")
   (marker :initform 'domain)
   (abbreviation :initform 'DFVEC)
   (comment :initform (list
     "This is a low-level domain which implements vectors"
     "(one dimensional arrays) of double precision floating point"
     "numbers.  Indexing is 0 based, there is no bound checking (unless"
     "provided by lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleFloatVector|
  (progn
    (push '|DoubleFloatVector| *Domains*)
    (make-instance '|DoubleFloatVectorType|)))
```

———-

1.56.18 DrawOption

— defclass DrawOptionType —

```
(defclass |DrawOptionType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "DrawOption")
   (marker :initform 'domain)
   (abbreviation :initform 'DROPT)
   (comment :initform (list
     "DrawOption allows the user to specify defaults for the"
     "creation and rendering of plots."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawOption|
  (progn
    (push '|DrawOption| *Domains*)
    (make-instance '|DrawOptionType|)))
```

———-

1.56.19 d01ajfAnnaType

— defclass d01ajfAnnaTypeType —

```
(defclass |d01ajfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01ajfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01AJFA)
   (comment :initform (list
     "d01ajfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01AJF, a general numerical integration routine which"
     "can handle some singularities in the input function.  The function"
     "measure measures the usefulness of the routine D01AJF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01ajfAnnaType|
  (progn
    (push '|d01ajfAnnaType| *Domains*)
    (make-instance '|d01ajfAnnaTypeType|)))
```

———-

1.56.20 d01akfAnnaType

— defclass d01akfAnnaTypeType —

```
(defclass |d01akfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01akfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01AKFA)
   (comment :initform (list
     "d01akfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01AKF, a numerical integration routine which is"
     "is suitable for oscillating, non-singular functions.  The function"
     "measure measures the usefulness of the routine D01AKF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01akfAnnaType|
  (progn
    (push '|d01akfAnnaType| *Domains*)
    (make-instance '|d01akfAnnaTypeType|)))
```

———-

1.56.21 d01alfAnnaType

— defclass d01alfAnnaTypeType —

```
(defclass |d01alfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01alfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01ALFA)
   (comment :initform (list
     "d01alfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01ALF, a general numerical integration routine which"
     "can handle a list of singularities.  The"
     "function measure measures the usefulness of the routine D01ALF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01alfAnnaType|
  (progn
    (push '|d01alfAnnaType| *Domains*)
    (make-instance '|d01alfAnnaTypeType|)))
```

———-

1.56.22 d01amfAnnaType

— defclass d01amfAnnaTypeType —

```
(defclass |d01amfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01amfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01AMFA)
   (comment :initform (list
     "d01amfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01AMF, a general numerical integration routine which"
     "can handle infinite or semi-infinite range of the input function.  The"
     "function measure measures the usefulness of the routine D01AMF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01amfAnnaType|
  (progn
    (push '|d01amfAnnaType| *Domains*)
    (make-instance '|d01amfAnnaTypeType|)))
```

———-

1.56.23 d01anfAnnaType

— defclass d01anfAnnaTypeType —

```
(defclass |d01anfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01anfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01ANFA)
   (comment :initform (list
     "d01anfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01ANF, a numerical integration routine which can"
     "handle weight functions of the form cos(omega x) or sin(omega x).  The"
     "function measure measures the usefulness of the routine D01ANF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01anfAnnaType|
  (progn
    (push '|d01anfAnnaType| *Domains*)
    (make-instance '|d01anfAnnaTypeType|)))
```

———-

1.56.24 d01apfAnnaType

— defclass d01apfAnnaTypeType —

```
(defclass |d01apfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01apfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01APFA)
   (comment :initform (list
     "d01apfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01APF, a general numerical integration routine which"
     "can handle end point singularities of the algebraico-logarithmic form"
     "   w(x) = (x-a)^c * (b-x)^d.  The"
     "function measure measures the usefulness of the routine D01APF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01apfAnnaType|
  (progn
    (push '|d01apfAnnaType| *Domains*)
    (make-instance '|d01apfAnnaTypeType|)))
```

———-

1.56.25 d01aqfAnnaType

— defclass d01aqfAnnaTypeType —

```
(defclass |d01aqfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01aqfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01AQFA)
   (comment :initform (list
     "d01aqfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01AQF, a general numerical integration routine which"
     "can solve an integral of the form"
     "/home/bjd/Axiom/anna/hypertex/bitmaps/d01aqf.xbm"
     "The function measure measures the usefulness of the routine"
     "D01AQF for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01aqfAnnaType|
  (progn
    (push '|d01aqfAnnaType| *Domains*)
    (make-instance '|d01aqfAnnaTypeType|)))
```

———-

1.56.26 d01asfAnnaType

— defclass d01asfAnnaTypeType —

```
(defclass |d01asfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01asfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01ASFA)
   (comment :initform (list
     "d01asfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01ASF, a numerical integration routine which can"
     "handle weight functions of the form cos(omega x) or sin(omega x) on an"
     "semi-infinite range.  The"
     "function measure measures the usefulness of the routine D01ASF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01asfAnnaType|
  (progn
    (push '|d01asfAnnaType| *Domains*)
    (make-instance '|d01asfAnnaTypeType|)))
```

———-

1.56.27 d01fcfAnnaType

— defclass d01fcfAnnaTypeType —

```
(defclass |d01fcfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01fcfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01FCFA)
   (comment :initform (list
     "d01fcfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01FCF, a numerical integration routine which can"
     "handle multi-dimensional quadrature over a finite region.  The"
     "function measure measures the usefulness of the routine D01GBF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |d01fcfAnnaType|
  (progn
    (push '|d01fcfAnnaType| *Domains*)
    (make-instance '|d01fcfAnnaTypeType|)))
```

———-

1.56.28 d01gbfAnnaType

— defclass d01gbfAnnaTypeType —

```
(defclass |d01gbfAnnaTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01gbfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01GBFA)
   (comment :initform (list
     "d01gbfAnnaType is a domain of"
     "NumericalIntegrationCategory"
     "for the NAG routine D01GBF, a numerical integration routine which can"
     "handle multi-dimensional quadrature over a finite region.  The"
     "function measure measures the usefulness of the routine D01GBF"
     "for the given problem.  The function numericalIntegration"
     "performs the integration by using NagIntegrationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01gbfAnnaType|
  (progn
    (push '|d01gbfAnnaType| *Domains*)
    (make-instance '|d01gbfAnnaTypeType|)))
```

———-

1.56.29 d01TransformFunctionType

— defclass d01TransformFunctionTypeType —

```
(defclass |d01TransformFunctionTypeType| (|NumericalIntegrationCategoryType|)
  ((parents :initform '(|NumericalIntegrationCategory|))
   (name :initform "d01TransformFunctionType")
   (marker :initform 'domain)
   (abbreviation :initform 'D01TRNS)
   (comment :initform (list
     "Since an infinite integral cannot be evaluated numerically"
     "it is necessary to transform the integral onto finite ranges."
     "d01TransformFunctionType uses the mapping x -> 1/x"
     "and contains the functions measure and"
     "numericalIntegration."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01TransformFunctionType|
  (progn
    (push '|d01TransformFunctionType| *Domains*)
    (make-instance '|d01TransformFunctionTypeType|)))
```

———-

1.56.30 d02bbfAnnaType

— defclass d02bbfAnnaTypeType —

```
(defclass |d02bbfAnnaTypeType| (|OrdinaryDifferentialEquationsSolverCategoryType|)
  ((parents :initform '(|OrdinaryDifferentialEquationsSolverCategory|))
   (name :initform "d02bbfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D02BBFA)
   (comment :initform (list
     "d02bbfAnnaType is a domain of"
     "OrdinaryDifferentialEquationsInitialValueProblemSolverCategory"
     "for the NAG routine D02BBF, a ODE routine which uses an"
     "Runge-Kutta method to solve a system of differential"
     "equations.  The function measure measures the"
     "usefulness of the routine D02BBF for the given problem.  The"
     "function ODESolve performs the integration by using"
     "NagOrdinaryDifferentialEquationsPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02bbfAnnaType|
  (progn
    (push '|d02bbfAnnaType| *Domains*)
    (make-instance '|d02bbfAnnaTypeType|)))
```

———-

1.56.31 d02bhfAnnaType

— defclass d02bhfAnnaTypeType —

```
(defclass |d02bhfAnnaTypeType| (|OrdinaryDifferentialEquationsSolverCategoryType|)
  ((parents :initform '(|OrdinaryDifferentialEquationsSolverCategory|))
   (name :initform "d02bhfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D02BHFA)
   (comment :initform (list
     "d02bhfAnnaType is a domain of"
     "OrdinaryDifferentialEquationsInitialValueProblemSolverCategory"
     "for the NAG routine D02BHF, a ODE routine which uses an"
     "Runge-Kutta method to solve a system of differential"
     "equations.  The function measure measures the"
     "usefulness of the routine D02BHF for the given problem.  The"
     "function ODESolve performs the integration by using"
     "NagOrdinaryDifferentialEquationsPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02bhfAnnaType|
  (progn
    (push '|d02bhfAnnaType| *Domains*)
    (make-instance '|d02bhfAnnaTypeType|)))
```

———-

1.56.32 d02cjfAnnaType

— defclass d02cjfAnnaTypeType —

```
(defclass |d02cjfAnnaTypeType| (|OrdinaryDifferentialEquationsSolverCategoryType|)
  ((parents :initform '(|OrdinaryDifferentialEquationsSolverCategory|))
   (name :initform "d02cjfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D02CJFA)
   (comment :initform (list
     "d02cjfAnnaType is a domain of"
     "OrdinaryDifferentialEquationsInitialValueProblemSolverCategory"
     "for the NAG routine D02CJF, a ODE routine which uses an"
     "Adams-Moulton-Bashworth method to solve a system of differential"
     "equations.  The function measure measures the"
     "usefulness of the routine D02CJF for the given problem.  The"
     "function ODESolve performs the integration by using"
     "NagOrdinaryDifferentialEquationsPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02cjfAnnaType|
  (progn
    (push '|d02cjfAnnaType| *Domains*)
    (make-instance '|d02cjfAnnaTypeType|)))
```

———-

1.56.33 d02ejfAnnaType

— defclass d02ejfAnnaTypeType —

```
(defclass |d02ejfAnnaTypeType| (|OrdinaryDifferentialEquationsSolverCategoryType|)
  ((parents :initform '(|OrdinaryDifferentialEquationsSolverCategory|))
   (name :initform "d02ejfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D02EJFA)
   (comment :initform (list
     "d02ejfAnnaType is a domain of"
     "OrdinaryDifferentialEquationsInitialValueProblemSolverCategory"
     "for the NAG routine D02EJF, a ODE routine which uses a backward"
     "differentiation formulae method to handle a stiff system"
     "of differential equations.  The function measure measures"
     "the usefulness of the routine D02EJF for the given problem.  The"
     "function ODESolve performs the integration by using"
     "NagOrdinaryDifferentialEquationsPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02ejfAnnaType|
  (progn
    (push '|d02ejfAnnaType| *Domains*)
    (make-instance '|d02ejfAnnaTypeType|)))
```

———-

1.56.34 d03eefAnnaType

— defclass d03eefAnnaTypeType —

```
(defclass |d03eefAnnaTypeType| (|PartialDifferentialEquationsSolverCategoryType|)
  ((parents :initform '(|PartialDifferentialEquationsSolverCategory|))
   (name :initform "d03eefAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D03EEFA)
   (comment :initform (list
     "d03eefAnnaType is a domain of"
     "PartialDifferentialEquationsSolverCategory"
     "for the NAG routines D03EEF/D03EDF."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d03eefAnnaType|
  (progn
    (push '|d03eefAnnaType| *Domains*)
    (make-instance '|d03eefAnnaTypeType|)))
```

———-

1.56.35 d03fafAnnaType

— defclass d03fafAnnaTypeType —

```
(defclass |d03fafAnnaTypeType| (|PartialDifferentialEquationsSolverCategoryType|)
  ((parents :initform '(|PartialDifferentialEquationsSolverCategory|))
   (name :initform "d03fafAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'D03FAFA)
   (comment :initform (list
     "d03fafAnnaType is a domain of"
     "PartialDifferentialEquationsSolverCategory"
     "for the NAG routine D03FAF."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d03fafAnnaType|
  (progn
    (push '|d03fafAnnaType| *Domains*)
    (make-instance '|d03fafAnnaTypeType|)))
```

———-

1.57 E

1.57.1 ElementaryFunctionsUnivariateLaurentSeries

— defclass ElementaryFunctionsUnivariateLaurentSeriesType —

```
(defclass |ElementaryFunctionsUnivariateLaurentSeriesType| (|PartialTranscendentalFunctionsType|)
  ((parents :initform '(|PartialTranscendentalFunctions|))
   (name :initform "ElementaryFunctionsUnivariateLaurentSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'EFULS)
   (comment :initform (list
     "This domain provides elementary functions on any Laurent series"
     "domain over a field which was constructed from a Taylor series"
     "domain.  These functions are implemented by calling the"
     "corresponding functions on the Taylor series domain.  We also"
     "provide 'partial functions' which compute transcendental"
     "functions of Laurent series when possible and return 'failed'"
     "when this is not possible."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |ElementaryFunctionsUnivariateLaurentSeries|
  (progn
    (push '|ElementaryFunctionsUnivariateLaurentSeries| *Domains*)
    (make-instance '|ElementaryFunctionsUnivariateLaurentSeriesType|)))
```

———–

1.57.2 ElementaryFunctionsUnivariatePuiseuxSeries

— defclass ElementaryFunctionsUnivariatePuiseuxSeriesType —

```
(defclass |ElementaryFunctionsUnivariatePuiseuxSeriesType| (|PartialTranscendentalFunctionsType|)
  ((parents :initform '(|PartialTranscendentalFunctions|))
   (name :initform "ElementaryFunctionsUnivariatePuiseuxSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'EFUPXS)
   (comment :initform (list
     "This package provides elementary functions on any Laurent series"
     "domain over a field which was constructed from a Taylor series"
     "domain.  These functions are implemented by calling the"
     "corresponding functions on the Taylor series domain.  We also"
     "provide 'partial functions' which compute transcendental"
     "functions of Laurent series when possible and return 'failed'"
     "when this is not possible."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionsUnivariatePuiseuxSeries|
  (progn
    (push '|ElementaryFunctionsUnivariatePuiseuxSeries| *Domains*)
    (make-instance '|ElementaryFunctionsUnivariatePuiseuxSeriesType|)))
```

———–

1.57.3 Equation

— defclass EquationType —

```
(defclass |EquationType| (|GroupType|
                          |InnerEvalableType|
                          |PartialDifferentialRingType|
                          |TypeType|
                          |VectorSpaceType|)
  ((parents :initform '(|Group|
                        |InnerEvalable|
                        |PartialDifferentialRing|
                        |Type|
                        |VectorSpace|))
```

```
   (name :initform "Equation")
   (marker :initform 'domain)
   (abbreviation :initform 'EQ)
   (comment :initform (list
     "Equations as mathematical objects.  All properties of the basis domain,"
     "for example being an abelian group are carried over the equation domain,"
     "by performing the structural operations on the left and on the"
     "right hand side."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Equation|
  (progn
    (push '|Equation| *Domains*)
    (make-instance '|EquationType|)))
```

————-

1.57.4 EqTable

— defclass EqTableType —

```
(defclass |EqTableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "EqTable")
   (marker :initform 'domain)
   (abbreviation :initform 'EQTBL)
   (comment :initform (list
     "This domain provides tables where the keys are compared using"
     "eq?.  Thus keys are considered equal only if they"
     "are the same instance of a structure."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EqTable|
  (progn
    (push '|EqTable| *Domains*)
    (make-instance '|EqTableType|)))
```

————-

1.57.5 EuclideanModularRing

— defclass EuclideanModularRingType —

```
(defclass |EuclideanModularRingType| (|EuclideanDomainType|)
  ((parents :initform '(|EuclideanDomain|))
   (name :initform "EuclideanModularRing")
   (marker :initform 'domain)
   (abbreviation :initform 'EMR)
   (comment :initform (list
     "These domains are used for the factorization and gcds"
     "of univariate polynomials over the integers in order to work modulo"
     "different  primes."
     "See ModularRing, ModularField"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EuclideanModularRing|
  (progn
    (push '|EuclideanModularRing| *Domains*)
    (make-instance '|EuclideanModularRingType|)))
```

———-

1.57.6 Exit

— defclass ExitType —

```
(defclass |ExitType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Exit")
   (marker :initform 'domain)
   (abbreviation :initform 'EXIT)
   (comment :initform (list
     "A function which does not return directly to its caller should"
     "have Exit as its return type."
     " "
     "Note that It is convenient to have a formal coerce into each type"
     "from type Exit. This allows, for example, errors to be raised in"
     "one half of a type-balanced if."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Exit|
  (progn
    (push '|Exit| *Domains*)
    (make-instance '|ExitType|)))
```

———-

1.57.7 ExponentialExpansion

— defclass ExponentialExpansionType —

```
(defclass |ExponentialExpansionType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "ExponentialExpansion")
   (marker :initform 'domain)
   (abbreviation :initform 'EXPEXPAN)
   (comment :initform (list
     "UnivariatePuiseuxSeriesWithExponentialSingularity is a domain used to"
     "represent essential singularities of functions.  Objects in this domain"
     "are quotients of sums, where each term in the sum is a univariate Puiseux"
     "series times the exponential of a univariate Puiseux series."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExponentialExpansion|
  (progn
    (push '|ExponentialExpansion| *Domains*)
    (make-instance '|ExponentialExpansionType|)))
```

———-

1.57.8 Expression

— defclass ExpressionType —

```
(defclass |ExpressionType| (|SpecialFunctionCategoryType|
                            |LiouvillianFunctionCategoryType|
                            |CombinatorialOpsCategoryType|
                            |AlgebraicallyClosedFunctionSpaceType|)
  ((parents :initform '(|SpecialFunctionCategory|
                        |LiouvillianFunctionCategory|
                        |CombinatorialOpsCategory|
                        |AlgebraicallyClosedFunctionSpace|))
   (name :initform "Expression")
   (marker :initform 'domain)
   (abbreviation :initform 'EXPR)
   (comment :initform (list
     "Top-level mathematical expressions involving symbolic functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Expression|
  (progn
    (push '|Expression| *Domains*)
    (make-instance '|ExpressionType|)))
```

———-

1.57.9 ExponentialOfUnivariatePuiseuxSeries

— defclass ExponentialOfUnivariatePuiseuxSeriesType —

```
(defclass |ExponentialOfUnivariatePuiseuxSeriesType| (|OrderedAbelianMonoidType|
                                                       |UnivariatePuiseuxSeriesCategoryType|)
  ((parents :initform '(|OrderedAbelianMonoid| |UnivariatePuiseuxSeriesCategory|))
   (name :initform "ExponentialOfUnivariatePuiseuxSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'EXPUPXS)
   (comment :initform (list
     "ExponentialOfUnivariatePuiseuxSeries is a domain used to represent"
     "essential singularities of functions.  An object in this domain is a"
     "function of the form exp(f(x)), where f(x) is a Puiseux"
     "series with no terms of non-negative degree.  Objects are ordered"
     "according to order of singularity, with functions which tend more"
     "rapidly to zero or infinity considered to be larger.  Thus, if"
     "order(f(x)) < order(g(x)), the first non-zero term of"
     "f(x) has lower degree than the first non-zero term of g(x),"
     "then exp(f(x)) > exp(g(x)).  If order(f(x)) = order(g(x)),"
     "then the ordering is essentially random.  This domain is used"
     "in computing limits involving functions with essential singularities."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExponentialOfUnivariatePuiseuxSeries|
  (progn
    (push '|ExponentialOfUnivariatePuiseuxSeries| *Domains*)
    (make-instance '|ExponentialOfUnivariatePuiseuxSeriesType|)))
```

———-

1.57.10 ExtAlgBasis

— defclass ExtAlgBasisType —

```
(defclass |ExtAlgBasisType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "ExtAlgBasis")
   (marker :initform 'domain)
   (abbreviation :initform 'EAB)
   (comment :initform (list
     "A domain used in the construction of the exterior algebra on a set"
     "X over a ring R.  This domain represents the set of all ordered"
     "subsets of the set X, assumed to be in correspondance with"
     "{1,2,3, ...}.  The ordered subsets are themselves ordered"
     "lexicographically and are in bijective correspondance with an ordered"
     "basis of the exterior algebra.  In this domain we are dealing strictly"
     "with the exponents of basis elements which can only be 0 or 1."
     " "
     "The multiplicative identity element of the exterior algebra corresponds"
     "to the empty subset of X.  A coerce from List Integer to an"
     "ordered basis element is provided to allow the convenient input of"
     "expressions. Another exported function forgets the ordered structure"
     "and simply returns the list corresponding to an ordered subset."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExtAlgBasis|
  (progn
    (push '|ExtAlgBasis| *Domains*)
    (make-instance '|ExtAlgBasisType|)))
```

———-

1.57.11 e04dgfAnnaType

— defclass e04dgfAnnaTypeType —

```
(defclass |e04dgfAnnaTypeType| (|NumericalOptimizationCategoryType|)
  ((parents :initform '(|NumericalOptimizationCategory|))
   (name :initform "e04dgfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'E04DGFA)
   (comment :initform (list
     "e04dgfAnnaType is a domain of NumericalOptimization"
     "for the NAG routine E04DGF, a general optimization routine which"
     "can handle some singularities in the input function.  The function"
     "measure measures the usefulness of the routine E04DGF"
     "for the given problem.  The function numericalOptimization"
     "performs the optimization by using NagOptimisationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04dgfAnnaType|
  (progn
    (push '|e04dgfAnnaType| *Domains*)
    (make-instance '|e04dgfAnnaTypeType|)))
```

———-

1.57.12 e04fdfAnnaType

— defclass e04fdfAnnaTypeType —

```
(defclass |e04fdfAnnaTypeType| (|NumericalOptimizationCategoryType|)
  ((parents :initform '(|NumericalOptimizationCategory|))
   (name :initform "e04fdfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'E04FDFA)
   (comment :initform (list
     "e04fdfAnnaType is a domain of NumericalOptimization"
     "for the NAG routine E04FDF, a general optimization routine which"
     "can handle some singularities in the input function.  The function"
     "measure measures the usefulness of the routine E04FDF"
     "for the given problem.  The function numericalOptimization"
     "performs the optimization by using NagOptimisationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04fdfAnnaType|
  (progn
    (push '|e04fdfAnnaType| *Domains*)
    (make-instance '|e04fdfAnnaTypeType|)))
```

———-

1.57.13 e04gcfAnnaType

— defclass e04gcfAnnaTypeType —

```
(defclass |e04gcfAnnaTypeType| (|NumericalOptimizationCategoryType|)
  ((parents :initform '(|NumericalOptimizationCategory|))
   (name :initform "e04gcfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'E04GCFA)
   (comment :initform (list
     "e04gcfAnnaType is a domain of NumericalOptimization"
     "for the NAG routine E04GCF, a general optimization routine which"
     "can handle some singularities in the input function.  The function"
     "measure measures the usefulness of the routine E04GCF"
     "for the given problem.  The function numericalOptimization"
     "performs the optimization by using NagOptimisationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04gcfAnnaType|
  (progn
    (push '|e04gcfAnnaType| *Domains*)
    (make-instance '|e04gcfAnnaTypeType|)))
```

———-

1.57.14 e04jafAnnaType

— defclass e04jafAnnaTypeType —

```
(defclass |e04jafAnnaTypeType| (|NumericalOptimizationCategoryType|)
  ((parents :initform '(|NumericalOptimizationCategory|))
   (name :initform "e04jafAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'E04JAFA)
   (comment :initform (list
     "e04jafAnnaType is a domain of NumericalOptimization"
     "for the NAG routine E04JAF, a general optimization routine which"
     "can handle some singularities in the input function.  The function"
     "measure measures the usefulness of the routine E04JAF"
     "for the given problem.  The function numericalOptimization"
     "performs the optimization by using NagOptimisationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04jafAnnaType|
  (progn
    (push '|e04jafAnnaType| *Domains*)
    (make-instance '|e04jafAnnaTypeType|)))
```

———-

1.57.15 e04mbfAnnaType

— defclass e04mbfAnnaTypeType —

```
(defclass |e04mbfAnnaTypeType| (|NumericalOptimizationCategoryType|)
  ((parents :initform '(|NumericalOptimizationCategory|))
   (name :initform "e04mbfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'E04MBFA)
   (comment :initform (list
     "e04mbfAnnaType is a domain of NumericalOptimization"
     "for the NAG routine E04MBF, an optimization routine for Linear functions."
     "The function"
     "measure measures the usefulness of the routine E04MBF"
     "for the given problem.  The function numericalOptimization"
     "performs the optimization by using NagOptimisationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04mbfAnnaType|
  (progn
    (push '|e04mbfAnnaType| *Domains*)
    (make-instance '|e04mbfAnnaTypeType|)))
```

———-

1.57.16 e04nafAnnaType

— defclass e04nafAnnaTypeType —

```
(defclass |e04nafAnnaTypeType| (|NumericalOptimizationCategoryType|)
  ((parents :initform '(|NumericalOptimizationCategory|))
   (name :initform "e04nafAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'E04NAFA)
   (comment :initform (list
     "e04nafAnnaType is a domain of NumericalOptimization"
     "for the NAG routine E04NAF, an optimization routine for Quadratic functions."
     "The function"
     "measure measures the usefulness of the routine E04NAF"
     "for the given problem.  The function numericalOptimization"
     "performs the optimization by using NagOptimisationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04nafAnnaType|
  (progn
    (push '|e04nafAnnaType| *Domains*)
    (make-instance '|e04nafAnnaTypeType|)))
```

———-

1.57.17 e04ucfAnnaType

— defclass e04ucfAnnaTypeType —

```
(defclass |e04ucfAnnaTypeType| (|NumericalOptimizationCategoryType|)
  ((parents :initform '(|NumericalOptimizationCategory|))
   (name :initform "e04ucfAnnaType")
   (marker :initform 'domain)
   (abbreviation :initform 'E04UCFA)
   (comment :initform (list
     "e04ucfAnnaType is a domain of NumericalOptimization"
     "for the NAG routine E04UCF, a general optimization routine which"
     "can handle some singularities in the input function.  The function"
     "measure measures the usefulness of the routine E04UCF"
     "for the given problem.  The function numericalOptimization"
     "performs the optimization by using NagOptimisationPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04ucfAnnaType|
  (progn
    (push '|e04ucfAnnaType| *Domains*)
    (make-instance '|e04ucfAnnaTypeType|)))
```

———-

1.58 F

1.58.1 Factored

— defclass FactoredType —

```
(defclass |FactoredType| (|UniqueFactorizationDomainType|
                          |RealConstantType|
                          |FullyRetractableToType|
                          |FullyEvalableOverType|
                          |DifferentialExtensionType|)
  ((parents :initform '(|UniqueFactorizationDomain|
                        |RealConstant|
                        |FullyRetractableTo|
                        |FullyEvalableOver|
                        |DifferentialExtension|))
   (name :initform "Factored")
   (marker :initform 'domain)
   (abbreviation :initform 'FR)
   (comment :initform (list
     "Factored creates a domain whose objects are kept in"
     "factored form as long as possible.  Thus certain operations like"
     "multiplication and gcd are relatively easy to do.  Others, like"
     "addition require somewhat more work, and unless the argument"
     "domain provides a factor function, the result may not be"
     "completely factored.  Each object consists of a unit and a list of"
     "factors, where a factor has a member of R (the 'base'), and"
     "exponent and a flag indicating what is known about the base.  A"
     "flag may be one of 'nil', 'sqfr', 'irred' or 'prime', which respectively mean"
     "that nothing is known about the base, it is square-free, it is"
     "irreducible, or it is prime.  The current"
     "restriction to integral domains allows simplification to be"
     "performed without worrying about multiplication order."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Factored|
  (progn
    (push '|Factored| *Domains*)
    (make-instance '|FactoredType|)))
```

———-

1.58.2 File

— defclass FileType —

```
(defclass |FileType| (|FileCategoryType|)
  ((parents :initform '(|FileCategory|))
   (name :initform "File")
   (marker :initform 'domain)
   (abbreviation :initform 'FILE)
   (comment :initform (list
     "This domain provides a basic model of files to save arbitrary values."
     "The operations provide sequential access to the contents."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |File|
  (progn
    (push '|File| *Domains*)
    (make-instance '|FileType|)))
```

———-

1.58.3 FileName

— defclass FileNameType —

```
(defclass |FileNameType| (|FileNameCategoryType|)
  ((parents :initform '(|FileNameCategory|))
   (name :initform "FileName")
   (marker :initform 'domain)
   (abbreviation :initform 'FNAME)
   (comment :initform (list
     "This domain provides an interface to names in the file system."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FileName|
  (progn
    (push '|FileName| *Domains*)
    (make-instance '|FileNameType|)))
```

———-

1.58.4 FiniteDivisor

— defclass FiniteDivisorType —

```
(defclass |FiniteDivisorType| (|FiniteDivisorCategoryType|)
  ((parents :initform '(|FiniteDivisorCategory|))
   (name :initform "FiniteDivisor")
   (marker :initform 'domain)
   (abbreviation :initform 'FDIV)
   (comment :initform (list
     "This domains implements finite rational divisors on a curve, that"
     "is finite formal sums SUM(n * P) where the n's are integers and the"
     "P's are finite rational points on the curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteDivisor|
  (progn
    (push '|FiniteDivisor| *Domains*)
    (make-instance '|FiniteDivisorType|)))
```

———-

1.58.5 FiniteField

— defclass FiniteFieldType —

```
(defclass |FiniteFieldType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteField")
   (marker :initform 'domain)
   (abbreviation :initform 'FF)
   (comment :initform (list
     "FiniteField(p,n) implements finite fields with p**n elements."
     "This packages checks that p is prime."
     "For a non-checking version, see InnerFiniteField."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteField|
  (progn
    (push '|FiniteField| *Domains*)
    (make-instance '|FiniteFieldType|)))
```

———-

1.58.6 FiniteFieldCyclicGroup

— defclass FiniteFieldCyclicGroupType —

```
(defclass |FiniteFieldCyclicGroupType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldCyclicGroup")
   (marker :initform 'domain)
   (abbreviation :initform 'FFCG)
   (comment :initform (list
     "FiniteFieldCyclicGroup(p,n) implements a finite field extension of degee n"
     "over the prime field with p elements. Its elements are represented by"
     "powers of a primitive element, a generator of the multiplicative"
     "(cyclic) group. As primitive element we choose the root of the extension"
     "polynomial, which is created by createPrimitivePoly from"
     "FiniteFieldPolynomialPackage. The Zech logarithms are stored"
     "in a table of size half of the field size, and use SingleInteger"
     "for representing field elements, hence, there are restrictions"
     "on the size of the field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCyclicGroup|
  (progn
    (push '|FiniteFieldCyclicGroup| *Domains*)
    (make-instance '|FiniteFieldCyclicGroupType|)))
```

————-

1.58.7 FiniteFieldCyclicGroupExtension

— defclass FiniteFieldCyclicGroupExtensionType —

```
(defclass |FiniteFieldCyclicGroupExtensionType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldCyclicGroupExtension")
   (marker :initform 'domain)
   (abbreviation :initform 'FFCGX)
   (comment :initform (list
     "FiniteFieldCyclicGroupExtension(GF,n)  implements a extension of degree n"
     "over the ground field GF. Its elements are represented by powers of"
     "a primitive element, a generator of the multiplicative (cyclic) group."
     "As primitive element we choose the root of the extension polynomial, which"
     "is created by createPrimitivePoly from"
     "FiniteFieldPolynomialPackage. Zech logarithms are stored"
     "in a table of size half of the field size, and use SingleInteger"
     "for representing field elements, hence, there are restrictions"
     "on the size of the field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCyclicGroupExtension|
  (progn
    (push '|FiniteFieldCyclicGroupExtension| *Domains*)
    (make-instance '|FiniteFieldCyclicGroupExtensionType|)))
```

———-

1.58.8 FiniteFieldCyclicGroupExtensionByPolynomial

— defclass FiniteFieldCyclicGroupExtensionByPolynomialType —

```
(defclass |FiniteFieldCyclicGroupExtensionByPolynomialType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldCyclicGroupExtensionByPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'FFCGP)
   (comment :initform (list
     "FiniteFieldCyclicGroupExtensionByPolynomial(GF,defpol)  implements a"
     "finite extension field of the ground field GF. Its elements are"
     "represented by powers of a primitive element, a generator of the"
     "multiplicative (cyclic) group. As primitive"
     "element we choose the root of the extension polynomial defpol,"
     "which MUST be primitive (user responsibility). Zech logarithms are stored"
     "in a table of size half of the field size, and use SingleInteger"
     "for representing field elements, hence, there are restrictions"
     "on the size of the field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCyclicGroupExtensionByPolynomial|
  (progn
    (push '|FiniteFieldCyclicGroupExtensionByPolynomial| *Domains*)
    (make-instance '|FiniteFieldCyclicGroupExtensionByPolynomialType|)))
```

———-

1.58.9 FiniteFieldExtension

— defclass FiniteFieldExtensionType —

```
(defclass |FiniteFieldExtensionType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldExtension")
   (marker :initform 'domain)
   (abbreviation :initform 'FFX)
   (comment :initform (list
     "FiniteFieldExtensionByPolynomial(GF, n) implements an extension"
     "of the finite field GF of degree n generated by the extension"
     "polynomial constructed by createIrreduciblePoly from"
     "FiniteFieldPolynomialPackage."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldExtension|
  (progn
    (push '|FiniteFieldExtension| *Domains*)
    (make-instance '|FiniteFieldExtensionType|)))
```

———-

1.58.10 FiniteFieldExtensionByPolynomial

— defclass FiniteFieldExtensionByPolynomialType —

```
(defclass |FiniteFieldExtensionByPolynomialType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldExtensionByPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'FFP)
   (comment :initform (list
     "FiniteFieldExtensionByPolynomial(GF, defpol) implements the extension"
     "of the finite field GF generated by the extension polynomial"
     "defpol which MUST be irreducible."
     "Note: the user has the responsibility to ensure that"
     "defpol is irreducible."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldExtensionByPolynomial|
  (progn
    (push '|FiniteFieldExtensionByPolynomial| *Domains*)
    (make-instance '|FiniteFieldExtensionByPolynomialType|)))
```

———-

1.58.11 FiniteFieldNormalBasis

— defclass FiniteFieldNormalBasisType —

```
(defclass |FiniteFieldNormalBasisType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldNormalBasis")
   (marker :initform 'domain)
   (abbreviation :initform 'FFNB)
   (comment :initform (list
     "FiniteFieldNormalBasis(p,n) implements a"
     "finite extension field of degree n over the prime field with p elements."
     "The elements are represented by coordinate vectors with respect to"
     "a normal basis,"
     "a basis consisting of the conjugates (q-powers) of an element, in"
     "this case called normal element."
     "This is chosen as a root of the extension polynomial"
     "created by createNormalPoly"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldNormalBasis|
  (progn
    (push '|FiniteFieldNormalBasis| *Domains*)
    (make-instance '|FiniteFieldNormalBasisType|)))
```

———-

1.58.12 FiniteFieldNormalBasisExtension

— defclass FiniteFieldNormalBasisExtensionType —

```
(defclass |FiniteFieldNormalBasisExtensionType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldNormalBasisExtension")
   (marker :initform 'domain)
   (abbreviation :initform 'FFNBX)
   (comment :initform (list
     "FiniteFieldNormalBasisExtensionByPolynomial(GF,n)  implements a"
     "finite extension field of degree n over the ground field GF."
     "The elements are represented by coordinate vectors with respect"
     "to a normal basis,"
     "a basis consisting of the conjugates (q-powers) of an element,"
     "in this case called normal element. This is chosen as a root of the extension"
     "polynomial, created by createNormalPoly from"
     "FiniteFieldPolynomialPackage"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldNormalBasisExtension|
  (progn
    (push '|FiniteFieldNormalBasisExtension| *Domains*)
    (make-instance '|FiniteFieldNormalBasisExtensionType|)))
```

———-

1.58.13 FiniteFieldNormalBasisExtensionByPolynomial

— defclass FiniteFieldNormalBasisExtensionByPolynomialType —

```
(defclass |FiniteFieldNormalBasisExtensionByPolynomialType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "FiniteFieldNormalBasisExtensionByPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'FFNBP)
   (comment :initform (list
     "FiniteFieldNormalBasisExtensionByPolynomial(GF,uni) implements a"
     "finite extension of the ground field GF. The elements are"
     "represented by coordinate vectors with respect to a normal basis, a basis"
     "consisting of the conjugates (q-powers) of an element, in this case"
     "called normal element, where q is the size of GF."
     "The normal element is chosen as a root of the extension"
     "polynomial, which MUST be normal over GF  (user responsibility)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldNormalBasisExtensionByPolynomial|
  (progn
    (push '|FiniteFieldNormalBasisExtensionByPolynomial| *Domains*)
    (make-instance '|FiniteFieldNormalBasisExtensionByPolynomialType|)))
```

———-

1.58.14 FlexibleArray

— defclass FlexibleArrayType —

```
(defclass |FlexibleArrayType| (|ExtensibleLinearAggregateType|
                               |OneDimensionalArrayAggregateType|)
  ((parents :initform '(|ExtensibleLinearAggregate| |OneDimensionalArrayAggregate|))
   (name :initform "FlexibleArray")
   (marker :initform 'domain)
   (abbreviation :initform 'FARRAY)
   (comment :initform (list
     "A FlexibleArray is the notion of an array intended to allow for growth"
     "at the end only.  Hence the following efficient operations"
     "append(x,a) meaning append item x at the end of the array a"
     "delete(a,n) meaning delete the last item from the array a"
     "Flexible arrays support the other operations inherited from"
     "ExtensibleLinearAggregate. However, these are not efficient."
     "Flexible arrays combine the O(1) access time property of arrays"
     "with growing and shrinking at the end in O(1) (average) time."
     "This is done by using an ordinary array which may have zero or more"
     "empty slots at the end.  When the array becomes full it is copied"
     "into a new larger (50% larger) array.  Conversely, when the array"
     "becomes less than 1/2 full, it is copied into a smaller array."
     "Flexible arrays provide for an efficient implementation of many"
     "data structures in particular heaps, stacks and sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FlexibleArray|
  (progn
    (push '|FlexibleArray| *Domains*)
    (make-instance '|FlexibleArrayType|)))
```

——–

1.58.15 Float

— defclass FloatType —

```
(defclass |FloatType| (|TranscendentalFunctionCategoryType|
                       |OpenMathType|
                       |FloatingPointSystemType|
                       |DifferentialRingType|)
  ((parents :initform '(|TranscendentalFunctionCategory|
                        |OpenMath|
                        |FloatingPointSystem|
                        |DifferentialRing|))
   (name :initform "Float")
   (marker :initform 'domain)
   (abbreviation :initform 'FLOAT)
   (comment :initform (list
     "Float implements arbitrary precision floating point arithmetic."
     "The number of significant digits of each operation can be set"
     "to an arbitrary value (the default is 20 decimal digits)."
     "The operation float(mantissa,exponent,base) for integer"
     "mantissa, exponent specifies the number"
     "mantissa * base ** exponent"
     "The underlying representation for floats is binary"
     "not decimal. The implications of this are described below."
     " "
     "The model adopted is that arithmetic operations are rounded to"
     "to nearest unit in the last place, that is, accurate to within"
     "2**(-bits). Also, the elementary functions and constants are"
     "accurate to one unit in the last place."
     "A float is represented as a record of two integers, the mantissa"
     "and the exponent.  The base of the representation is binary, hence"
     "a Record(m:mantissa,e:exponent) represents the number"
     "m * 2 ** e."
     "Though it is not assumed that the underlying integers are represented"
     "with a binary base, the code will be most efficient when this is the"
     "the case (this is true in most implementations of Lisp)."
     "The decision to choose the base to be binary has some unfortunate"
     "consequences.  First, decimal numbers like 0.3 cannot be represented"
     "exactly.  Second, there is a further loss of accuracy during"
     "conversion to decimal for output.  To compensate for this, if d digits"
     "of precision are specified, 1 + ceiling(log2(10^d)) bits are used."
     "Two numbers that are displayed identically may therefore be"
     "not equal.  On the other hand, a significant efficiency loss would"
     "be incurred if we chose to use a decimal base when the underlying"
     "integer base is binary."
     " "
     "Algorithms used:"
     "For the elementary functions, the general approach is to apply"
     "identities so that the taylor series can be used, and, so"
     "that it will converge within O( sqrt n ) steps.  For example,"
     "using the identity exp(x) = exp(x/2)**2, we can compute"
     "exp(1/3) to n digits of precision as follows.  We have"
     "exp(1/3) = exp(2 ** (-sqrt s) / 3) ** (2 ** sqrt s)."
     "The taylor series will converge in less than sqrt n steps and the"
     "exponentiation requires sqrt n multiplications for a total of"
     "2 sqrt n multiplications.  Assuming integer multiplication costs"
     "O( n**2 ) the overall running time is O( sqrt(n) n**2 )."
     "This approach is the best known approach for precisions up to"
     "about 10,000 digits at which point the methods of Brent"
     "which are O( log(n) n**2 ) become competitive.  Note also that"
     "summing the terms of the taylor series for the elementary"
     "functions is done using integer operations.  This avoids the"
     "overhead of floating point operations and results in efficient"
     "code at low precisions.  This implementation makes no attempt"
     "to reuse storage, relying on the underlying system to do"
     "garbage collection. I estimate that the efficiency of this"
     "package at low precisions could be improved by a factor of 2"
     "if in-place operations were available."
     " "
     "Running times: in the following, n is the number of bits of precision"
     "*, /, sqrt, pi, exp1, log2, log10:  O( n**2 )"
     "exp, log, sin, atan: O(sqrt(n) n**2)"
     "The other elementary functions are coded in terms of the ones above."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Float|
  (progn
    (push '|Float| *Domains*)
    (make-instance '|FloatType|)))
```

————-

1.58.16 FortranCode

— defclass FortranCodeType —

```
(defclass |FortranCodeType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "FortranCode")
   (marker :initform 'domain)
   (abbreviation :initform 'FC)
   (comment :initform (list
     "This domain builds representations of program code segments for use with"
     "the FortranProgram domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranCode|
  (progn
    (push '|FortranCode| *Domains*)
    (make-instance '|FortranCodeType|)))
```

———-

1.58.17 FortranExpression

— defclass FortranExpressionType —

```
(defclass |FortranExpressionType| (|AlgebraType|
                                   |ExpressionSpaceType|
                                   |PartialDifferentialRingType|)
  ((parents :initform '(|Algebra|
                        |ExpressionSpace|
                        |PartialDifferentialRing|))
   (name :initform "FortranExpression")
   (marker :initform 'domain)
   (abbreviation :initform 'FEXPR)
   (comment :initform (list
     "A domain of expressions involving functions which can be"
     "translated into standard Fortran-77, with some extra extensions from"
     "the NAG Fortran Library."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranExpression|
  (progn
    (push '|FortranExpression| *Domains*)
    (make-instance '|FortranExpressionType|)))
```

———-

1.58.18 FortranProgram

— defclass FortranProgramType —

```
(defclass |FortranProgramType| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "FortranProgram")
   (marker :initform 'domain)
   (abbreviation :initform 'FORTRAN)
   (comment :initform (list
     "FortranProgram allows the user to build and manipulate simple"
     "models of FORTRAN subprograms.  These can then be transformed into"
     "actual FORTRAN notation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranProgram|
  (progn
    (push '|FortranProgram| *Domains*)
    (make-instance '|FortranProgramType|)))
```

———-

1.58.19 FortranScalarType

— defclass FortranScalarTypeType —

```
(defclass |FortranScalarTypeType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "FortranScalarType")
   (marker :initform 'domain)
   (abbreviation :initform 'FST)
   (comment :initform (list
     "Creates and manipulates objects which correspond to the"
     "basic FORTRAN data types: REAL, INTEGER, COMPLEX, LOGICAL and CHARACTER"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranScalarType|
  (progn
    (push '|FortranScalarType| *Domains*)
    (make-instance '|FortranScalarTypeType|)))
```

———-

1.58.20 FortranTemplate

— defclass FortranTemplateType —

```
(defclass |FortranTemplateType| (|FileCategoryType|)
  ((parents :initform '(|FileCategory|))
   (name :initform "FortranTemplate")
   (marker :initform 'domain)
   (abbreviation :initform 'FTEM)
   (comment :initform (list
     "Code to manipulate Fortran templates"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranTemplate|
  (progn
    (push '|FortranTemplate| *Domains*)
    (make-instance '|FortranTemplateType|)))
```

———-

1.58.21 FortranType

— defclass FortranTypeType —

```
(defclass |FortranTypeType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "FortranType")
   (marker :initform 'domain)
   (abbreviation :initform 'FT)
   (comment :initform (list
     "Creates and manipulates objects which correspond to FORTRAN"
     "data types, including array dimensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranType|
  (progn
    (push '|FortranType| *Domains*)
    (make-instance '|FortranTypeType|)))
```

———-

1.58.22 FourierComponent

— defclass FourierComponentType —

```
(defclass |FourierComponentType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "FourierComponent")
   (marker :initform 'domain)
   (abbreviation :initform 'FCOMP)
   (comment :initform (list
     "This domain creates kernels for use in Fourier series"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FourierComponent|
  (progn
    (push '|FourierComponent| *Domains*)
    (make-instance '|FourierComponentType|)))
```

———-

1.58.23 FourierSeries

— defclass FourierSeriesType —

```
(defclass |FourierSeriesType| (|AlgebraType|)
  ((parents :initform '(|Algebra|))
   (name :initform "FourierSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'FSERIES)
   (comment :initform (list
     "This domain converts terms into Fourier series"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FourierSeries|
  (progn
    (push '|FourierSeries| *Domains*)
    (make-instance '|FourierSeriesType|)))
```

———-

1.58.24 Fraction

— defclass FractionType —

```
(defclass |FractionType| (|QuotientFieldCategoryType| |OpenMathType|)
  ((parents :initform '(|QuotientFieldCategory| |OpenMath|))
   (name :initform "Fraction")
   (marker :initform 'domain)
   (abbreviation :initform 'FRAC)
   (comment :initform (list
     "Fraction takes an IntegralDomain S and produces"
     "the domain of Fractions with numerators and denominators from S."
     "If S is also a GcdDomain, then gcd's between numerator and"
     "denominator will be cancelled during all operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Fraction|
  (progn
    (push '|Fraction| *Domains*)
    (make-instance '|FractionType|)))
```

———-

1.58.25 FractionalIdeal

— defclass FractionalIdealType —

```
(defclass |FractionalIdealType| (|GroupType|)
  ((parents :initform '(|Group|))
   (name :initform "FractionalIdeal")
   (marker :initform 'domain)
   (abbreviation :initform 'FRIDEAL)
   (comment :initform (list
     "Fractional ideals in a framed algebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionalIdeal|
  (progn
    (push '|FractionalIdeal| *Domains*)
    (make-instance '|FractionalIdealType|)))
```

———-

1.58.26 FramedModule

— defclass FramedModuleType —

```
(defclass |FramedModuleType| (|MonoidType|)
  ((parents :initform '(|Monoid|))
   (name :initform "FramedModule")
   (marker :initform 'domain)
   (abbreviation :initform 'FRMOD)
   (comment :initform (list
     "Module representation of fractional ideals."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedModule|
  (progn
    (push '|FramedModule| *Domains*)
    (make-instance '|FramedModuleType|)))
```

———-

1.58.27 FreeAbelianGroup

— defclass FreeAbelianGroupType —

```
(defclass |FreeAbelianGroupType| (|OrderedSetType|
                                  |ModuleType|
                                  |FreeAbelianMonoidCategoryType|)
  ((parents :initform '(|OrderedSet| |Module| |FreeAbelianMonoidCategory|))
   (name :initform "FreeAbelianGroup")
   (marker :initform 'domain)
   (abbreviation :initform 'FAGROUP)
   (comment :initform (list
     "Free abelian group on any set of generators"
     "The free abelian group on a set S is the monoid of finite sums of"
     "the form reduce(+,[ni * si]) where the si's are in S, and the ni's"
     "are integers. The operation is commutative."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeAbelianGroup|
  (progn
    (push '|FreeAbelianGroup| *Domains*)
    (make-instance '|FreeAbelianGroupType|)))
```

———-

1.58.28 FreeAbelianMonoid

— defclass FreeAbelianMonoidType —

```
(defclass |FreeAbelianMonoidType| (|FreeAbelianMonoidCategoryType|)
  ((parents :initform '(|FreeAbelianMonoidCategory|))
   (name :initform "FreeAbelianMonoid")
   (marker :initform 'domain)
   (abbreviation :initform 'FAMONOID)
   (comment :initform (list
     "Free abelian monoid on any set of generators"
     "The free abelian monoid on a set S is the monoid of finite sums of"
     "the form reduce(+,[ni * si]) where the si's are in S, and the ni's"
     "are non-negative integers. The operation is commutative."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeAbelianMonoid|
  (progn
    (push '|FreeAbelianMonoid| *Domains*)
    (make-instance '|FreeAbelianMonoidType|)))
```

————-

1.58.29 FreeGroup

— defclass FreeGroupType —

```
(defclass |FreeGroupType| (|GroupType| |RetractableToType|)
  ((parents :initform '(|Group| |RetractableTo|))
   (name :initform "FreeGroup")
   (marker :initform 'domain)
   (abbreviation :initform 'FGROUP)
   (comment :initform (list
     "Free group on any set of generators"
     "The free group on a set S is the group of finite products of"
     "the form reduce(*,[si ** ni]) where the si's are in S, and the ni's"
     "are integers. The multiplication is not commutative."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeGroup|
  (progn
    (push '|FreeGroup| *Domains*)
    (make-instance '|FreeGroupType|)))
```

————-

1.58.30 FreeModule

— defclass FreeModuleType —

```
(defclass |FreeModuleType| (|IndexedDirectProductCategoryType|
                            |ModuleType|)
  ((parents :initform '(|IndexedDirectProductCategory| |Module|))
   (name :initform "FreeModule")
   (marker :initform 'domain)
   (abbreviation :initform 'FM)
   (comment :initform (list
     "A bi-module is a free module"
     "over a ring with generators indexed by an ordered set."
     "Each element can be expressed as a finite linear combination of"
     "generators. Only non-zero terms are stored."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeModule|
  (progn
    (push '|FreeModule| *Domains*)
    (make-instance '|FreeModuleType|)))
```

———-

1.58.31 FreeModule1

— defclass FreeModule1Type —

```
(defclass |FreeModule1Type| (|FreeModuleCatType|)
  ((parents :initform '(|FreeModuleCat|))
   (name :initform "FreeModule1")
   (marker :initform 'domain)
   (abbreviation :initform 'FM1)
   (comment :initform (list
     "This domain implements linear combinations"
     "of elements from the domain S with coefficients"
     "in the domain R where S is an ordered set"
     "and R is a ring (which may be non-commutative)."
     "This domain is used by domains of non-commutative algebra such as:"
     "XDistributedPolynomial, XRecursivePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeModule1|
  (progn
    (push '|FreeModule1| *Domains*)
    (make-instance '|FreeModule1Type|)))
```

———-

1.58.32 FreeMonoid

— defclass FreeMonoidType —

```
(defclass |FreeMonoidType| (|MonoidType| |OrderedSetType| |RetractableToType|)
  ((parents :initform '(|Monoid| |OrderedSet| |RetractableTo|))
   (name :initform "FreeMonoid")
   (marker :initform 'domain)
   (abbreviation :initform 'FMONOID)
   (comment :initform (list
     "Free monoid on any set of generators"
     "The free monoid on a set S is the monoid of finite products of"
     "the form reduce(*,[si ** ni]) where the si's are in S, and the ni's"
     "are nonnegative integers. The multiplication is not commutative."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeMonoid|
  (progn
    (push '|FreeMonoid| *Domains*)
    (make-instance '|FreeMonoidType|)))
```

———-

1.58.33 FreeNilpotentLie

— defclass FreeNilpotentLieType —

```
(defclass |FreeNilpotentLieType| (|NonAssociativeAlgebraType|)
  ((parents :initform '(|NonAssociativeAlgebra|))
   (name :initform "FreeNilpotentLie")
   (marker :initform 'domain)
   (abbreviation :initform 'FNLA)
   (comment :initform (list
     "Generate the Free Lie Algebra over a ring R with identity;"
     "A P. Hall basis is generated by a package call to HallBasis."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeNilpotentLie|
  (progn
    (push '|FreeNilpotentLie| *Domains*)
    (make-instance '|FreeNilpotentLieType|)))
```

———-

1.58.34 FullPartialFractionExpansion

— defclass FullPartialFractionExpansionType —

```
(defclass |FullPartialFractionExpansionType| (|SetCategoryType| |ConvertibleToType|)
  ((parents :initform '(|SetCategory| |ConvertibleTo|))
   (name :initform "FullPartialFractionExpansion")
   (marker :initform 'domain)
   (abbreviation :initform 'FPARFRAC)
   (comment :initform (list
     "Full partial fraction expansion of rational functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullPartialFractionExpansion|
  (progn
    (push '|FullPartialFractionExpansion| *Domains*)
    (make-instance '|FullPartialFractionExpansionType|)))
```

———-

1.58.35 FunctionCalled

— defclass FunctionCalledType —

```
(defclass |FunctionCalledType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "FunctionCalled")
   (marker :initform 'domain)
   (abbreviation :initform 'FUNCTION)
   (comment :initform (list
     "This domain implements named functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionCalled|
  (progn
    (push '|FunctionCalled| *Domains*)
    (make-instance '|FunctionCalledType|)))
```

———-

1.59 G

1.59.1 GeneralDistributedMultivariatePolynomial

— defclass GeneralDistributedMultivariatePolynomialType —

```
(defclass |GeneralDistributedMultivariatePolynomialType| (|PolynomialCategoryType|)
  ((parents :initform '(|PolynomialCategory|))
   (name :initform "GeneralDistributedMultivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'GDMP)
   (comment :initform (list
     "This type supports distributed multivariate polynomials"
     "whose variables are from a user specified list of symbols."
     "The coefficient ring may be non commutative,"
     "but the variables are assumed to commute."
     "The term ordering is specified by its third parameter."
     "Suggested types which define term orderings include:"
     "DirectProduct, HomogeneousDirectProduct,"
     "SplitHomogeneousDirectProduct and finally"
     "OrderedDirectProduct which accepts an arbitrary user"
     "function to define a term ordering."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralDistributedMultivariatePolynomial|
  (progn
    (push '|GeneralDistributedMultivariatePolynomial| *Domains*)
    (make-instance '|GeneralDistributedMultivariatePolynomialType|)))
```

———-

1.59.2 GeneralModulePolynomial

— defclass GeneralModulePolynomialType —

```
(defclass |GeneralModulePolynomialType| (|ModuleType|)
  ((parents :initform '(|Module|))
   (name :initform "GeneralModulePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'GMODPOL)
   (comment :initform (list
     "This package is undocumented"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralModulePolynomial|
  (progn
    (push '|GeneralModulePolynomial| *Domains*)
    (make-instance '|GeneralModulePolynomialType|)))
```

———-

1.59.3 GenericNonAssociativeAlgebra

— defclass GenericNonAssociativeAlgebraType —

```
(defclass |GenericNonAssociativeAlgebraType| (|FramedNonAssociativeAlgebraType|)
  ((parents :initform '(|FramedNonAssociativeAlgebra|))
   (name :initform "GenericNonAssociativeAlgebra")
   (marker :initform 'domain)
   (abbreviation :initform 'GCNAALG)
   (comment :initform (list
     "AlgebraGenericElementPackage allows you to create generic elements"
     "of an algebra, the scalars are extended to include symbolic"
     "coefficients"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenericNonAssociativeAlgebra|
  (progn
    (push '|GenericNonAssociativeAlgebra| *Domains*)
    (make-instance '|GenericNonAssociativeAlgebraType|)))
```

———-

1.59.4 GeneralPolynomialSet

— defclass GeneralPolynomialSetType —

```
(defclass |GeneralPolynomialSetType| (|PolynomialSetCategoryType|)
  ((parents :initform '(|PolynomialSetCategory|))
   (name :initform "GeneralPolynomialSet")
   (marker :initform 'domain)
   (abbreviation :initform 'GPOLSET)
   (comment :initform (list
     "A domain for polynomial sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralPolynomialSet|
  (progn
    (push '|GeneralPolynomialSet| *Domains*)
    (make-instance '|GeneralPolynomialSetType|)))
```

———-

1.59.5 GeneralSparseTable

— defclass GeneralSparseTableType —

```
(defclass |GeneralSparseTableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "GeneralSparseTable")
   (marker :initform 'domain)
   (abbreviation :initform 'GSTBL)
   (comment :initform (list
     "A sparse table has a default entry, which is returned if no other"
     "value has been explicitly stored for a key."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralSparseTable|
  (progn
    (push '|GeneralSparseTable| *Domains*)
    (make-instance '|GeneralSparseTableType|)))
```

———-

1.59.6 GeneralTriangularSet

— defclass GeneralTriangularSetType —

```
(defclass |GeneralTriangularSetType| (|TriangularSetCategoryType|)
  ((parents :initform '(|TriangularSetCategory|))
   (name :initform "GeneralTriangularSet")
   (marker :initform 'domain)
   (abbreviation :initform 'GTSET)
   (comment :initform (list
     "A domain constructor of the category TriangularSetCategory."
     "The only requirement for a list of polynomials to be a member of such"
     "a domain is the following: no polynomial is constant and two distinct"
     "polynomials have distinct main variables. Such a triangular set may"
     "not be auto-reduced or consistent. Triangular sets are stored"
     "as sorted lists w.r.t. the main variables of their members but they"
     "are displayed in reverse order."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralTriangularSet|
  (progn
    (push '|GeneralTriangularSet| *Domains*)
    (make-instance '|GeneralTriangularSetType|)))
```

———-

1.59.7 GeneralUnivariatePowerSeries

— defclass GeneralUnivariatePowerSeriesType —

```
(defclass |GeneralUnivariatePowerSeriesType| (|UnivariatePuiseuxSeriesCategoryType|)
  ((parents :initform '(|UnivariatePuiseuxSeriesCategory|))
   (name :initform "GeneralUnivariatePowerSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'GSERIES)
   (comment :initform (list
     "This is a category of univariate Puiseux series constructed"
     "from univariate Laurent series.  A Puiseux series is represented"
     "by a pair [r,f(x)], where r is a positive rational number and"
     "f(x) is a Laurent series.  This pair represents the Puiseux"
     "series f(x\^r)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralUnivariatePowerSeries|
  (progn
    (push '|GeneralUnivariatePowerSeries| *Domains*)
    (make-instance '|GeneralUnivariatePowerSeriesType|)))
```

——————-

1.59.8 GraphImage

— defclass GraphImageType —

```
(defclass |GraphImageType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "GraphImage")
   (marker :initform 'domain)
   (abbreviation :initform 'GRIMAGE)
   (comment :initform (list
     "TwoDimensionalGraph creates virtual two dimensional graphs"
     "(to be displayed on TwoDimensionalViewports)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GraphImage|
  (progn
    (push '|GraphImage| *Domains*)
    (make-instance '|GraphImageType|)))
```

——————-

1.59.9 GuessOption

— defclass GuessOptionType —

```
(defclass |GuessOptionType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "GuessOption")
   (marker :initform 'domain)
   (abbreviation :initform 'GOPT)
   (comment :initform (list
     "GuessOption is a domain whose elements are various options used"
     "by Guess."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessOption|
  (progn
    (push '|GuessOption| *Domains*)
    (make-instance '|GuessOptionType|)))
```

———-

1.59.10 GuessOptionFunctions0

— defclass GuessOptionFunctions0Type —

```
(defclass |GuessOptionFunctions0Type| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "GuessOptionFunctions0")
   (marker :initform 'domain)
   (abbreviation :initform 'GOPT0)
   (comment :initform (list
     "GuessOptionFunctions0 provides operations that extract the"
     "values of options for Guess."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessOptionFunctions0|
  (progn
    (push '|GuessOptionFunctions0| *Domains*)
    (make-instance '|GuessOptionFunctions0Type|)))
```

———-

1.60 H

1.60.1 HashTable

— defclass HashTableType —

```
(defclass |HashTableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "HashTable")
   (marker :initform 'domain)
   (abbreviation :initform 'HASHTBL)
   (comment :initform (list
     "This domain provides access to the underlying Lisp hash tables."
     "By varying the hashfn parameter, tables suited for different"
     "purposes can be obtained."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HashTable|
   (progn
     (push '|HashTable| *Domains*)
     (make-instance '|HashTableType|)))
```

———-

1.60.2 Heap

— defclass HeapType —

```
(defclass |HeapType| (|PriorityQueueAggregateType|)
  ((parents :initform '(|PriorityQueueAggregate|))
   (name :initform "Heap")
   (marker :initform 'domain)
   (abbreviation :initform 'HEAP)
   (comment :initform (list
     "Heap implemented in a flexible array to allow for insertions"
     "Complexity: O(log n) insertion, extraction and O(n) construction"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Heap|
  (progn
    (push '|Heap| *Domains*)
    (make-instance '|HeapType|)))
```

———-

1.60.3 HexadecimalExpansion

— defclass HexadecimalExpansionType —

```
(defclass |HexadecimalExpansionType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "HexadecimalExpansion")
   (marker :initform 'domain)
   (abbreviation :initform 'HEXADEC)
   (comment :initform (list
     "This domain allows rational numbers to be presented as repeating"
     "hexadecimal expansions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HexadecimalExpansion|
  (progn
    (push '|HexadecimalExpansion| *Domains*)
    (make-instance '|HexadecimalExpansionType|)))
```

———-

1.60.4 HTMLFormat

— defclass HTMLFormatType —

```
(defclass |HTMLFormatType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "HTMLFormat")
   (marker :initform 'domain)
   (abbreviation :initform 'HTMLFORM)
   (comment :initform (list
     "HtmlFormat provides a coercion from OutputForm to html."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HTMLFormat|
  (progn
    (push '|HTMLFormat| *Domains*)
    (make-instance '|HTMLFormatType|)))
```

———-

1.60.5 HomogeneousDirectProduct

— defclass HomogeneousDirectProductType —

```
(defclass |HomogeneousDirectProductType| (|DirectProductCategoryType|)
  ((parents :initform '(|DirectProductCategory|))
   (name :initform "HomogeneousDirectProduct")
   (marker :initform 'domain)
   (abbreviation :initform 'HDP)
   (comment :initform (list
     "This type represents the finite direct or cartesian product of an"
     "underlying ordered component type. The vectors are ordered first"
     "by the sum of their components, and then refined using a reverse"
     "lexicographic ordering. This type is a suitable third argument for"
     "GeneralDistributedMultivariatePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HomogeneousDirectProduct|
  (progn
    (push '|HomogeneousDirectProduct| *Domains*)
    (make-instance '|HomogeneousDirectProductType|)))
```

———-

1.60.6 HomogeneousDistributedMultivariatePolynomial

— defclass HomogeneousDistributedMultivariatePolynomialType —

```
(defclass |HomogeneousDistributedMultivariatePolynomialType| (|PolynomialCategoryType|)
 ((parents :initform '(|PolynomialCategory|))
  (name :initform "HomogeneousDistributedMultivariatePolynomial")
  (marker :initform 'domain)
  (abbreviation :initform 'HDMP)
  (comment :initform (list
    "This type supports distributed multivariate polynomials"
    "whose variables are from a user specified list of symbols."
    "The coefficient ring may be non commutative,"
    "but the variables are assumed to commute."
    "The term ordering is total degree ordering refined by reverse"
    "lexicographic ordering with respect to the position that the variables"
    "appear in the list of variables parameter."))
  (argslist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |HomogeneousDistributedMultivariatePolynomial|
 (progn
  (push '|HomogeneousDistributedMultivariatePolynomial| *Domains*)
  (make-instance '|HomogeneousDistributedMultivariatePolynomialType|)))
```

———-

1.60.7 HyperellipticFiniteDivisor

— defclass HyperellipticFiniteDivisorType —

```
(defclass |HyperellipticFiniteDivisorType| (|FiniteDivisorCategoryType|)
  ((parents :initform '(|FiniteDivisorCategory|))
   (name :initform "HyperellipticFiniteDivisor")
   (marker :initform 'domain)
   (abbreviation :initform 'HELLFDIV)
   (comment :initform (list
     "This domains implements finite rational divisors on an hyperelliptic curve,"
     "that is finite formal sums SUM(n * P) where the n's are integers and the"
     "P's are finite rational points on the curve."
     "The equation of the curve must be  y^2 = f(x) and f must have odd degree."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HyperellipticFiniteDivisor|
  (progn
    (push '|HyperellipticFiniteDivisor| *Domains*)
    (make-instance '|HyperellipticFiniteDivisorType|)))
```

————-

1.61 I

1.61.1 InfClsPt

— defclass InfClsPtType —

```
(defclass |InfClsPtType| (|InfinitlyClosePointCategoryType|)
  ((parents :initform '(|InfinitlyClosePointCategory|))
   (name :initform "InfClsPt")
   (marker :initform 'domain)
   (abbreviation :initform 'ICP)
   (comment :initform (list
     "This domain is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfClsPt|
  (progn
    (push '|InfClsPt| *Domains*)
    (make-instance '|InfClsPtType|)))
```

————-

1.61.2 IndexCard

— defclass IndexCardType —

```
(defclass |IndexCardType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "IndexCard")
   (marker :initform 'domain)
   (abbreviation :initform 'ICARD)
   (comment :initform (list
     "This domain implements a container of information about the AXIOM library"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexCard|
  (progn
    (push '|IndexCard| *Domains*)
    (make-instance '|IndexCardType|)))
```

——–-

1.61.3 IndexedBits

— defclass IndexedBitsType —

```
(defclass |IndexedBitsType| (|BitAggregateType|)
  ((parents :initform '(|BitAggregate|))
   (name :initform "IndexedBits")
   (marker :initform 'domain)
   (abbreviation :initform 'IBITS)
   (comment :initform (list
     "IndexedBits is a domain to compactly represent"
     "large quantities of Boolean data."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedBits|
  (progn
    (push '|IndexedBits| *Domains*)
    (make-instance '|IndexedBitsType|)))
```

——–-

1.61.4 IndexedDirectProductAbelianGroup

— defclass IndexedDirectProductAbelianGroupType —

```
(defclass |IndexedDirectProductAbelianGroupType| (|AbelianGroupType|
                                                  |IndexedDirectProductCategoryType|)
  ((parents :initform '(|AbelianGroup|
                        |IndexedDirectProductCategory|))
   (name :initform "IndexedDirectProductAbelianGroup")
   (marker :initform 'domain)
   (abbreviation :initform 'IDPAG)
   (comment :initform (list
     "Indexed direct products of abelian groups over an abelian group A of"
     "generators indexed by the ordered set S."
     "All items have finite support: only non-zero terms are stored."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductAbelianGroup|
  (progn
    (push '|IndexedDirectProductAbelianGroup| *Domains*)
    (make-instance '|IndexedDirectProductAbelianGroupType|)))
```

———-

1.61.5 IndexedDirectProductAbelianMonoid

— defclass IndexedDirectProductAbelianMonoidType —

```
(defclass |IndexedDirectProductAbelianMonoidType| (|AbelianMonoidType|
                                                    |IndexedDirectProductCategoryType|)
  ((parents :initform '(|AbelianMonoid|
                        |IndexedDirectProductCategory|))
   (name :initform "IndexedDirectProductAbelianMonoid")
   (marker :initform 'domain)
   (abbreviation :initform 'IDPAM)
   (comment :initform (list
     "Indexed direct products of abelian monoids over an abelian monoid"
     "A of generators indexed by the ordered set S. All items have"
     "finite support. Only non-zero terms are stored."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductAbelianMonoid|
  (progn
    (push '|IndexedDirectProductAbelianMonoid| *Domains*)
    (make-instance '|IndexedDirectProductAbelianMonoidType|)))
```

———-

1.61.6 IndexedDirectProductObject

— defclass IndexedDirectProductObjectType —

```
(defclass |IndexedDirectProductObjectType| (|IndexedDirectProductCategoryType|)
  ((parents :initform '(|IndexedDirectProductCategory|))
   (name :initform "IndexedDirectProductObject")
   (marker :initform 'domain)
   (abbreviation :initform 'IDPO)
   (comment :initform (list
     "Indexed direct products of objects over a set A"
     "of generators indexed by an ordered set S. All items have finite support."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductObject|
  (progn
    (push '|IndexedDirectProductObject| *Domains*)
    (make-instance '|IndexedDirectProductObjectType|)))
```

————-

1.61.7 IndexedDirectProductOrderedAbelianMonoid

— defclass IndexedDirectProductOrderedAbelianMonoidType —

```
(defclass |IndexedDirectProductOrderedAbelianMonoidType| (|OrderedAbelianMonoidType|
                                                          |IndexedDirectProductCategoryType|)
  ((parents :initform '(|OrderedAbelianMonoid|
                        |IndexedDirectProductCategory|))
   (name :initform "IndexedDirectProductOrderedAbelianMonoid")
   (marker :initform 'domain)
   (abbreviation :initform 'IDPOAM)
   (comment :initform (list
     "Indexed direct products of ordered abelian monoids A of"
     "generators indexed by the ordered set S."
     "The inherited order is lexicographical."
     "All items have finite support: only non-zero terms are stored."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductOrderedAbelianMonoid|
  (progn
    (push '|IndexedDirectProductOrderedAbelianMonoid| *Domains*)
    (make-instance '|IndexedDirectProductOrderedAbelianMonoidType|)))
```

————-

1.61.8 IndexedDirectProductOrderedAbelianMonoidSup

— defclass IndexedDirectProductOrderedAbelianMonoidSupType —

```
(defclass |IndexedDirectProductOrderedAbelianMonoidSupType| (|IndexedDirectProductCategoryType|
                                                             |OrderedAbelianMonoidSupType|)
  ((parents :initform '(|IndexedDirectProductCategory| |OrderedAbelianMonoidSup|))
   (name :initform "IndexedDirectProductOrderedAbelianMonoidSup")
   (marker :initform 'domain)
   (abbreviation :initform 'IDPOAMS)
   (comment :initform (list
     "Indexed direct products of ordered abelian monoid sups A,"
     "generators indexed by the ordered set S."
     "All items have finite support: only non-zero terms are stored."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductOrderedAbelianMonoidSup|
  (progn
    (push '|IndexedDirectProductOrderedAbelianMonoidSup| *Domains*)
    (make-instance '|IndexedDirectProductOrderedAbelianMonoidSupType|)))
```

———-

1.61.9 IndexedExponents

— defclass IndexedExponentsType —

```
(defclass |IndexedExponentsType| (|IndexedDirectProductCategoryType|
                                  |OrderedAbelianMonoidSupType|)
  ((parents :initform '(|IndexedDirectProductCategory| |OrderedAbelianMonoidSup|))
   (name :initform "IndexedExponents")
   (marker :initform 'domain)
   (abbreviation :initform 'INDE)
   (comment :initform (list
     "IndexedExponents of an ordered set of variables gives a representation"
     "for the degree of polynomials in commuting variables. It gives an ordered"
     "pairing of non negative integer exponents with variables"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedExponents|
  (progn
    (push '|IndexedExponents| *Domains*)
    (make-instance '|IndexedExponentsType|)))
```

———-

1.61.10 IndexedFlexibleArray

— defclass IndexedFlexibleArrayType —

```
(defclass |IndexedFlexibleArrayType| (|ExtensibleLinearAggregateType|
                                      |OneDimensionalArrayAggregateType|)
  ((parents :initform '(|ExtensibleLinearAggregate| |OneDimensionalArrayAggregate|))
   (name :initform "IndexedFlexibleArray")
   (marker :initform 'domain)
   (abbreviation :initform 'IFARRAY)
   (comment :initform (list
     "A FlexibleArray is the notion of an array intended to allow for growth"
     "at the end only.  Hence the following efficient operations"
     "append(x,a) meaning append item x at the end of the array a"
     "delete(a,n) meaning delete the last item from the array a"
     "Flexible arrays support the other operations inherited from"
     "ExtensibleLinearAggregate. However, these are not efficient."
     "Flexible arrays combine the O(1) access time property of arrays"
     "with growing and shrinking at the end in O(1) (average) time."
     "This is done by using an ordinary array which may have zero or more"
     "empty slots at the end.  When the array becomes full it is copied"
     "into a new larger (50% larger) array.  Conversely, when the array"
     "becomes less than 1/2 full, it is copied into a smaller array."
     "Flexible arrays provide for an efficient implementation of many"
     "data structures in particular heaps, stacks and sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedFlexibleArray|
  (progn
    (push '|IndexedFlexibleArray| *Domains*)
    (make-instance '|IndexedFlexibleArrayType|)))
```

——————-

1.61.11 IndexedList

— defclass IndexedListType —

```
(defclass |IndexedListType| (|ListAggregateType|)
  ((parents :initform '(|ListAggregate|))
   (name :initform "IndexedList")
   (marker :initform 'domain)
   (abbreviation :initform 'ILIST)
   (comment :initform (list
     "IndexedList is a basic implementation of the functions"
     "in ListAggregate, often using functions in the underlying"
     "LISP system. The second parameter to the constructor (mn)"
     "is the beginning index of the list. That is, if l is a"
     "list, then elt(l,mn) is the first value. This constructor"
     "is probably best viewed as the implementation of singly-linked"
     "lists that are addressable by index rather than as a mere wrapper"
     "for LISP lists."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedList|
  (progn
    (push '|IndexedList| *Domains*)
    (make-instance '|IndexedListType|)))
```

———-

1.61.12 IndexedMatrix

— defclass IndexedMatrixType —

```
(defclass |IndexedMatrixType| (|MatrixCategoryType|)
  ((parents :initform '(|MatrixCategory|))
   (name :initform "IndexedMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'IMATRIX)
   (comment :initform (list
     "An IndexedMatrix is a matrix where the minimal row and column"
     "indices are parameters of the type.  The domains Row and Col"
     "are both IndexedVectors."
     "The index of the 'first' row may be obtained by calling the"
     "function minRowIndex.  The index of the 'first' column may"
     "be obtained by calling the function minColIndex.  The index of"
     "the first element of a 'Row' is the same as the index of the"
     "first column in a matrix and vice versa."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedMatrix|
  (progn
    (push '|IndexedMatrix| *Domains*)
    (make-instance '|IndexedMatrixType|)))
```

———-

1.61.13 IndexedOneDimensionalArray

— defclass IndexedOneDimensionalArrayType —

```
(defclass |IndexedOneDimensionalArrayType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "IndexedOneDimensionalArray")
   (marker :initform 'domain)
   (abbreviation :initform 'IARRAY1)
   (comment :initform (list
     "This is the basic one dimensional array data type."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedOneDimensionalArray|
  (progn
    (push '|IndexedOneDimensionalArray| *Domains*)
    (make-instance '|IndexedOneDimensionalArrayType|)))
```

——-

1.61.14 IndexedString

— defclass IndexedStringType —

```
(defclass |IndexedStringType| (|StringAggregateType|)
  ((parents :initform '(|StringAggregate|))
   (name :initform "IndexedString")
   (marker :initform 'domain)
   (abbreviation :initform 'ISTRING)
   (comment :initform (list
     "This domain implements low-level strings"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedString|
  (progn
    (push '|IndexedString| *Domains*)
    (make-instance '|IndexedStringType|)))
```

——-

1.61.15 IndexedTwoDimensionalArray

— defclass IndexedTwoDimensionalArrayType —

```
(defclass |IndexedTwoDimensionalArrayType| (|TwoDimensionalArrayCategoryType|)
  ((parents :initform '(|TwoDimensionalArrayCategory|))
   (name :initform "IndexedTwoDimensionalArray")
   (marker :initform 'domain)
   (abbreviation :initform 'IARRAY2)
   (comment :initform (list
     "This domain implements two dimensional arrays"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |IndexedTwoDimensionalArray|
  (progn
    (push '|IndexedTwoDimensionalArray| *Domains*)
    (make-instance '|IndexedTwoDimensionalArrayType|)))
```

———-

1.61.16 IndexedVector

— defclass IndexedVectorType —

```
(defclass |IndexedVectorType| (|VectorCategoryType|)
  ((parents :initform '(|VectorCategory|))
   (name :initform "IndexedVector")
   (marker :initform 'domain)
   (abbreviation :initform 'IVECTOR)
   (comment :initform (list
     "This type represents vector like objects with varying lengths"
     "and a user-specified initial index."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedVector|
  (progn
    (push '|IndexedVector| *Domains*)
    (make-instance '|IndexedVectorType|)))
```

———-

1.61.17 InfiniteTuple

— defclass InfiniteTupleType —

```
(defclass |InfiniteTupleType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "InfiniteTuple")
   (marker :initform 'domain)
   (abbreviation :initform 'ITUPLE)
   (comment :initform (list
     "This package implements 'infinite tuples' for the interpreter."
     "The representation is a stream."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteTuple|
  (progn
    (push '|InfiniteTuple| *Domains*)
    (make-instance '|InfiniteTupleType|)))
```

———-

1.61.18 InfinitlyClosePoint

— defclass InfinitlyClosePointType —

```
(defclass |InfinitlyClosePointType| (|InfinitlyClosePointCategoryType|)
  ((parents :initform '(|InfinitlyClosePointCategory|))
   (name :initform "InfinitlyClosePoint")
   (marker :initform 'domain)
   (abbreviation :initform 'INFCLSPT)
   (comment :initform (list
     "This domain is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfinitlyClosePoint|
  (progn
    (push '|InfinitlyClosePoint| *Domains*)
    (make-instance '|InfinitlyClosePointType|)))
```

———-

1.61.19 InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField

— defclass InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteFieldType —

```
(defclass |InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteFieldType| (|InfinitlyClosePointCategoryType|)
  ((parents :initform '(|InfinitlyClosePointCategory|))
   (name :initform "InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField")
   (marker :initform 'domain)
   (abbreviation :initform 'INFCLSPS)
   (comment :initform (list
     "This domain is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.61.20 InnerAlgebraicNumber

— defclass InnerAlgebraicNumberType —

```
(defclass |InnerAlgebraicNumberType| (|AlgebraicallyClosedFieldType|
                                      |CharacteristicZeroType|
                                      |DifferentialRingType|
                                      |ExpressionSpaceType|
                                      |LinearlyExplicitRingOverType|
                                      |RealConstantType|)
  ((parents :initform '(|AlgebraicallyClosedField|
                        |CharacteristicZero|
                        |DifferentialRing|
                        |ExpressionSpace|
                        |LinearlyExplicitRingOver|
                        |RealConstant|))
   (name :initform "InnerAlgebraicNumber")
   (marker :initform 'domain)
   (abbreviation :initform 'IAN)
   (comment :initform (list
     "Algebraic closure of the rational numbers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerAlgebraicNumber|
  (progn
    (push '|InnerAlgebraicNumber| *Domains*)
    (make-instance '|InnerAlgebraicNumberType|)))
```

———-

1.61.21 InnerFiniteField

— defclass InnerFiniteFieldType —

```
(defclass |InnerFiniteFieldType| (|FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|FiniteAlgebraicExtensionField|))
   (name :initform "InnerFiniteField")
   (marker :initform 'domain)
   (abbreviation :initform 'IFF)
   (comment :initform (list
     "InnerFiniteField(p,n) implements finite fields with p**n elements"
     "where p is assumed prime but does not check."
     "For a version which checks that p is prime, see FiniteField."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerFiniteField|
  (progn
    (push '|InnerFiniteField| *Domains*)
    (make-instance '|InnerFiniteFieldType|)))
```

———-

1.61.22 InnerFreeAbelianMonoid

— defclass InnerFreeAbelianMonoidType —

```
(defclass |InnerFreeAbelianMonoidType| (|FreeAbelianMonoidCategoryType|)
  ((parents :initform '(|FreeAbelianMonoidCategory|))
   (name :initform "InnerFreeAbelianMonoid")
   (marker :initform 'domain)
   (abbreviation :initform 'IFAMON)
   (comment :initform (list
     "Internal implementation of a free abelian monoid on any set of generators"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerFreeAbelianMonoid|
  (progn
    (push '|InnerFreeAbelianMonoid| *Domains*)
    (make-instance '|InnerFreeAbelianMonoidType|)))
```

———-

1.61.23 InnerIndexedTwoDimensionalArray

— defclass InnerIndexedTwoDimensionalArrayType —

```
(defclass |InnerIndexedTwoDimensionalArrayType| (|TwoDimensionalArrayCategoryType|)
  ((parents :initform '(|TwoDimensionalArrayCategory|))
   (name :initform "InnerIndexedTwoDimensionalArray")
   (marker :initform 'domain)
   (abbreviation :initform 'IIARRAY2)
   (comment :initform (list
     "There is no description for this domain"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerIndexedTwoDimensionalArray|
  (progn
    (push '|InnerIndexedTwoDimensionalArray| *Domains*)
    (make-instance '|InnerIndexedTwoDimensionalArrayType|)))
```

———-

1.61.24 InnerPAdicInteger

— defclass InnerPAdicIntegerType —

```
(defclass |InnerPAdicIntegerType| (|PAdicIntegerCategoryType|)
  ((parents :initform '(|PAdicIntegerCategory|))
   (name :initform "InnerPAdicInteger")
   (marker :initform 'domain)
   (abbreviation :initform 'IPADIC)
   (comment :initform (list
     "This domain implements Zp, the p-adic completion of the integers."
     "This is an internal domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerPAdicInteger|
  (progn
    (push '|InnerPAdicInteger| *Domains*)
    (make-instance '|InnerPAdicIntegerType|)))
```

———-

1.61.25 InnerPrimeField

— defclass InnerPrimeFieldType —

```
(defclass |InnerPrimeFieldType| (|ConvertibleToType|
                                 |FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|ConvertibleTo| |FiniteAlgebraicExtensionField|))
   (name :initform "InnerPrimeField")
   (marker :initform 'domain)
   (abbreviation :initform 'IPF)
   (comment :initform (list
     "InnerPrimeField(p) implements the field with p elements."
     "Note: argument p MUST be a prime (this domain does not check)."
     "See PrimeField for a domain that does check."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerPrimeField|
  (progn
    (push '|InnerPrimeField| *Domains*)
    (make-instance '|InnerPrimeFieldType|)))
```

———-

1.61.26 InnerSparseUnivariatePowerSeries

— defclass InnerSparseUnivariatePowerSeriesType —

```
(defclass |InnerSparseUnivariatePowerSeriesType| (|UnivariatePowerSeriesCategoryType|)
  ((parents :initform '(|UnivariatePowerSeriesCategory|))
   (name :initform "InnerSparseUnivariatePowerSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'ISUPS)
   (comment :initform (list
     "InnerSparseUnivariatePowerSeries is an internal domain"
     "used for creating sparse Taylor and Laurent series."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerSparseUnivariatePowerSeries|
  (progn
    (push '|InnerSparseUnivariatePowerSeries| *Domains*)
    (make-instance '|InnerSparseUnivariatePowerSeriesType|)))
```

———-

1.61.27 InnerTable

— defclass InnerTableType —

```
(defclass |InnerTableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "InnerTable")
   (marker :initform 'domain)
   (abbreviation :initform 'INTABL)
   (comment :initform (list
     "This domain is used to provide a conditional 'add' domain"
     "for the implementation of Table."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerTable|
  (progn
    (push '|InnerTable| *Domains*)
    (make-instance '|InnerTableType|)))
```

———-

1.61.28 InnerTaylorSeries

— defclass InnerTaylorSeriesType —

```
(defclass |InnerTaylorSeriesType| (|IntegralDomainType|)
  ((parents :initform '(|IntegralDomain|))
   (name :initform "InnerTaylorSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'ITAYLOR)
   (comment :initform (list
     "This is an internal Taylor series type in which Taylor series"
     "are represented by a Stream of Ring elements."
     "For univariate series, the Stream elements are the Taylor"
     "coefficients. For multivariate series, the nth Stream element"
     "is a form of degree n in the power series variables."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerTaylorSeries|
  (progn
    (push '|InnerTaylorSeries| *Domains*)
    (make-instance '|InnerTaylorSeriesType|)))
```

———-

1.61.29 InputForm

— defclass InputFormType —

```
(defclass |InputFormType| (|ConvertibleToType| |SExpressionCategoryType|)
  ((parents :initform '(|ConvertibleTo| |SExpressionCategory|))
   (name :initform "InputForm")
   (marker :initform 'domain)
   (abbreviation :initform 'INFORM)
   (comment :initform (list
     "Domain of parsed forms which can be passed to the interpreter."
     "This is also the interface between algebra code and facilities"
     "in the interpreter."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InputForm|
  (progn
    (push '|InputForm| *Domains*)
    (make-instance '|InputFormType|)))
```

———-

1.61.30 Integer

— defclass IntegerType —

```
(defclass |IntegerType| (|IntegerNumberSystemType| |OpenMathType|)
  ((parents :initform '(|IntegerNumberSystem| |OpenMath|))
   (name :initform "Integer")
   (marker :initform 'domain)
   (abbreviation :initform 'INT)
   (comment :initform (list
     "Integer provides the domain of arbitrary precision integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Integer|
  (progn
    (push '|Integer| *Domains*)
    (make-instance '|IntegerType|)))
```

———-

1.61.31 IntegerMod

— defclass IntegerModType —

```
(defclass |IntegerModType| (|StepThroughType|
                            |FiniteType|
                            |ConvertibleToType|
                            |CommutativeRingType|)
  ((parents :initform '(|StepThrough|
                        |Finite|
                        |ConvertibleTo|
                        |CommutativeRing|))
   (name :initform "IntegerMod")
   (marker :initform 'domain)
   (abbreviation :initform 'ZMOD)
   (comment :initform (list
     "IntegerMod(n) creates the ring of integers reduced modulo the integer n."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerMod|
  (progn
    (push '|IntegerMod| *Domains*)
    (make-instance '|IntegerModType|)))
```

———-

1.61.32 IntegrationFunctionsTable

— defclass IntegrationFunctionsTableType —

```
(defclass |IntegrationFunctionsTableType| (|AxiomClass|)
  ((parents :initform '())
   (name :initform "IntegrationFunctionsTable")
   (marker :initform 'domain)
   (abbreviation :initform 'INTFTBL)
   (comment :initform (list
     "There is no description for this domain"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationFunctionsTable|
  (progn
    (push '|IntegrationFunctionsTable| *Domains*)
    (make-instance '|IntegrationFunctionsTableType|)))
```

———-

1.61.33 IntegrationResult

— defclass IntegrationResultType —

```
(defclass |IntegrationResultType| (|RetractableToType| |ModuleType|)
  ((parents :initform '(|RetractableTo| |Module|))
   (name :initform "IntegrationResult")
   (marker :initform 'domain)
   (abbreviation :initform 'IR)
   (comment :initform (list
     "The result of a transcendental integration."
     "If a function f has an elementary integral g, then g can be written"
     "in the form g = h + c1 log(u1) + c2 log(u2) + ... + cn log(un)"
     "where h, which is in the same field than f, is called the rational"
     "part of the integral, and c1 log(u1) + ... cn log(un) is called the"
     "logarithmic part of the integral. This domain manipulates integrals"
     "represented in that form, by keeping both parts separately. The logs"
     "are not explicitly computed."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResult|
  (progn
    (push '|IntegrationResult| *Domains*)
    (make-instance '|IntegrationResultType|)))
```

———-

1.61.34 Interval

— defclass IntervalType —

```
(defclass |IntervalType| (|IntervalCategoryType|)
  ((parents :initform '(|IntervalCategory|))
   (name :initform "Interval")
   (marker :initform 'domain)
   (abbreviation :initform 'INTRVL)
   (comment :initform (list
     "This domain is an implementation of interval arithmetic and transcendental"
     "functions over intervals."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Interval|
  (progn
    (push '|Interval| *Domains*)
    (make-instance '|IntervalType|)))
```

———-

1.62 K

1.62.1 Kernel

— defclass KernelType —

```
(defclass |KernelType| (|CachableSetType| |PatternableType|)
  ((parents :initform '(|CachableSet| |Patternable|))
   (name :initform "Kernel")
   (marker :initform 'domain)
   (abbreviation :initform 'KERNEL)
   (comment :initform (list
     "A kernel over a set S is an operator applied to a given list"
     "of arguments from S."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Kernel|
  (progn
    (push '|Kernel| *Domains*)
    (make-instance '|KernelType|)))
```

———-

1.62.2 KeyedAccessFile

— defclass KeyedAccessFileType —

```
(defclass |KeyedAccessFileType| (|TableAggregateType| |FileCategoryType|)
  ((parents :initform '(|TableAggregate| |FileCategory|))
   (name :initform "KeyedAccessFile")
   (marker :initform 'domain)
   (abbreviation :initform 'KAFILE)
   (comment :initform (list
     "This domain allows a random access file to be viewed both as a table"
     "and as a file object. The KeyedAccessFile format is a directory"
     "containing a single file called ``index.kaf''. This file is a random"
     "access file. The first thing in the file is an integer which is the"
     "byte offset of an association list (the dictionary) at the end of"
     "the file. The association list is of the form"
     "((key . byteoffset) (key . byteoffset)...)"
     "where the byte offset is the number of bytes from the beginning of"
     "the file. This offset contains an s-expression for the value of the key."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |KeyedAccessFile|
  (progn
    (push '|KeyedAccessFile| *Domains*)
    (make-instance '|KeyedAccessFileType|)))
```

———-

1.63 L

1.63.1 LaurentPolynomial

— defclass LaurentPolynomialType —

```
(defclass |LaurentPolynomialType| (|FullyRetractableToType|
                                   |EuclideanDomainType|
                                   |DifferentialExtensionType|
                                   |ConvertibleToType|
                                   |CharacteristicZeroType|
                                   |CharacteristicNonZeroType|)
  ((parents :initform '(|FullyRetractableTo|
                        |EuclideanDomain|
                        |DifferentialExtension|
                        |ConvertibleTo|
                        |CharacteristicZero|
                        |CharacteristicNonZero|))
   (name :initform "LaurentPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'LAUPOL)
   (comment :initform (list
     "Univariate polynomials with negative and positive exponents."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LaurentPolynomial|
  (progn
    (push '|LaurentPolynomial| *Domains*)
    (make-instance '|LaurentPolynomialType|)))
```

———-

1.63.2 Library

— defclass LibraryType —

```
(defclass |LibraryType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "Library")
   (marker :initform 'domain)
   (abbreviation :initform 'LIB)
   (comment :initform (list
     "This domain provides a simple way to save values in files."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Library|
  (progn
    (push '|Library| *Domains*)
    (make-instance '|LibraryType|)))
```

———-

1.63.3 LieExponentials

— defclass LieExponentialsType —

```
(defclass |LieExponentialsType| (|GroupType|)
  ((parents :initform '(|Group|))
   (name :initform "LieExponentials")
   (marker :initform 'domain)
   (abbreviation :initform 'LEXP)
   (comment :initform (list
     "Management of the Lie Group associated with a"
     "free nilpotent Lie algebra. Every Lie bracket with"
     "length greater than Order are assumed to be null."
     "The implementation inherits from the XPBWPolynomial"
     "domain constructor: Lyndon coordinates are exponential coordinates"
     "of the second kind. "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LieExponentials|
  (progn
    (push '|LieExponentials| *Domains*)
    (make-instance '|LieExponentialsType|)))
```

———-

1.63.4 LiePolynomial

— defclass LiePolynomialType —

```
(defclass |LiePolynomialType| (|FreeLieAlgebraType| |FreeModuleCatType|)
  ((parents :initform '(|FreeLieAlgebra| |FreeModuleCat|))
   (name :initform "LiePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'LPOLY)
   (comment :initform (list
     "This type supports Lie polynomials in Lyndon basis"
     "see Free Lie Algebras by C. Reutenauer"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LiePolynomial|
  (progn
    (push '|LiePolynomial| *Domains*)
    (make-instance '|LiePolynomialType|)))
```

———-

1.63.5 LieSquareMatrix

— defclass LieSquareMatrixType —

```
(defclass |LieSquareMatrixType| (|FramedNonAssociativeAlgebraType|
                                 |SquareMatrixCategoryType|)
  ((parents :initform '(|FramedNonAssociativeAlgebra| |SquareMatrixCategory|))
   (name :initform "LieSquareMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'LSQM)
   (comment :initform (list
     "LieSquareMatrix(n,R) implements the Lie algebra of the n by n"
     "matrices over the commutative ring R."
     "The Lie bracket (commutator) of the algebra is given by"
     "a*b := (a *$SQMATRIX(n,R) b - b *$SQMATRIX(n,R) a),"
     "where *$SQMATRIX(n,R) is the usual matrix multiplication."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LieSquareMatrix|
  (progn
    (push '|LieSquareMatrix| *Domains*)
    (make-instance '|LieSquareMatrixType|)))
```

———-

1.63.6 LinearOrdinaryDifferentialOperator

— defclass LinearOrdinaryDifferentialOperatorType —

```
(defclass |LinearOrdinaryDifferentialOperatorType| (|LinearOrdinaryDifferentialOperatorCategoryType|)
  ((parents :initform '(|LinearOrdinaryDifferentialOperatorCategory|))
   (name :initform "LinearOrdinaryDifferentialOperator")
   (marker :initform 'domain)
   (abbreviation :initform 'LODO)
   (comment :initform (list
     "LinearOrdinaryDifferentialOperator defines a ring of"
     "differential operators with coefficients in a ring A with a given"
     "derivation."
     "Multiplication of operators corresponds to functional composition:"
     "(L1 * L2).(f) = L1 L2 f"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperator|
  (progn
    (push '|LinearOrdinaryDifferentialOperator| *Domains*)
    (make-instance '|LinearOrdinaryDifferentialOperatorType|)))
```

———-

1.63.7 LinearOrdinaryDifferentialOperator1

— defclass LinearOrdinaryDifferentialOperator1Type —

```
(defclass |LinearOrdinaryDifferentialOperator1Type| (|LinearOrdinaryDifferentialOperatorCategoryType|)
  ((parents :initform '(|LinearOrdinaryDifferentialOperatorCategory|))
   (name :initform "LinearOrdinaryDifferentialOperator1")
   (marker :initform 'domain)
   (abbreviation :initform 'LODO1)
   (comment :initform (list
     "LinearOrdinaryDifferentialOperator1 defines a ring of"
     "differential operators with coefficients in a differential ring A"
     "Multiplication of operators corresponds to functional composition:"
     "(L1 * L2).(f) = L1 L2 f"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperator1|
  (progn
    (push '|LinearOrdinaryDifferentialOperator1| *Domains*)
    (make-instance '|LinearOrdinaryDifferentialOperator1Type|)))
```

————-

1.63.8 LinearOrdinaryDifferentialOperator2

— defclass LinearOrdinaryDifferentialOperator2Type —

```
(defclass |LinearOrdinaryDifferentialOperator2Type| (|LinearOrdinaryDifferentialOperatorCategoryType|)
  ((parents :initform '(|LinearOrdinaryDifferentialOperatorCategory|))
   (name :initform "LinearOrdinaryDifferentialOperator2")
   (marker :initform 'domain)
   (abbreviation :initform 'LODO2)
   (comment :initform (list
     "LinearOrdinaryDifferentialOperator2 defines a ring of"
     "differential operators with coefficients in a differential ring A"
     "and acting on an A-module M."
     "Multiplication of operators corresponds to functional composition:"
     "(L1 * L2).(f) = L1 L2 f"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperator2|
  (progn
    (push '|LinearOrdinaryDifferentialOperator2| *Domains*)
    (make-instance '|LinearOrdinaryDifferentialOperator2Type|)))
```

————-

1.63.9 List

— defclass ListType —

```
(defclass |ListType| (|OpenMathType| |ListAggregateType|)
  ((parents :initform '(|OpenMath| |ListAggregate|))
   (name :initform "List")
   (marker :initform 'domain)
   (abbreviation :initform 'LIST)
   (comment :initform (list
     "List implements singly-linked lists that are"
     "addressable by indices; the index of the first element"
     "is 1. In addition to the operations provided by"
     "IndexedList, this constructor provides some"
     "LISP-like functions such as null and cons."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |List|
  (progn
    (push '|List| *Domains*)
    (make-instance '|ListType|)))
```

———-

1.63.10 ListMonoidOps

— defclass ListMonoidOpsType —

```
(defclass |ListMonoidOpsType| (|SetCategoryType| |RetractableToType|)
  ((parents :initform '(|SetCategory| |RetractableTo|))
   (name :initform "ListMonoidOps")
   (marker :initform 'domain)
   (abbreviation :initform 'LMOPS)
   (comment :initform (list
     "This internal package represents monoid (abelian or not, with or"
     "without inverses) as lists and provides some common operations"
     "to the various flavors of monoids."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListMonoidOps|
  (progn
    (push '|ListMonoidOps| *Domains*)
    (make-instance '|ListMonoidOpsType|)))
```

———-

1.63.11 ListMultiDictionary

— defclass ListMultiDictionaryType —

```
(defclass |ListMultiDictionaryType| (|MultiDictionaryType|)
  ((parents :initform '(|MultiDictionary|))
   (name :initform "ListMultiDictionary")
   (marker :initform 'domain)
   (abbreviation :initform 'LMDICT)
   (comment :initform (list
     "The ListMultiDictionary domain implements a"
     "dictionary with duplicates"
     "allowed.  The representation is a list with duplicates represented"
     "explicitly.  Hence most operations will be relatively inefficient"
     "when the number of entries in the dictionary becomes large."
     "If the objects in the dictionary belong to an ordered set,"
     "the entries are maintained in ascending order."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListMultiDictionary|
  (progn
    (push '|ListMultiDictionary| *Domains*)
    (make-instance '|ListMultiDictionaryType|)))
```

———-

1.63.12 LocalAlgebra

— defclass LocalAlgebraType —

```
(defclass |LocalAlgebraType| (|AlgebraType| |OrderedRingType|)
  ((parents :initform '(|Algebra| |OrderedRing|))
   (name :initform "LocalAlgebra")
   (marker :initform 'domain)
   (abbreviation :initform 'LA)
   (comment :initform (list
     "LocalAlgebra produces the localization of an algebra,"
     "fractions whose numerators come from some R algebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LocalAlgebra|
  (progn
    (push '|LocalAlgebra| *Domains*)
    (make-instance '|LocalAlgebraType|)))
```

———-

1.63.13 Localize

— defclass LocalizeType —

```
(defclass |LocalizeType| (|ModuleType| |OrderedAbelianGroupType|)
  ((parents :initform '(|Module| |OrderedAbelianGroup|))
   (name :initform "Localize")
   (marker :initform 'domain)
   (abbreviation :initform 'LO)
   (comment :initform (list
     "Localize(M,R,S) produces fractions with numerators"
     "from an R module M and denominators from some multiplicative subset D of R."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Localize|
  (progn
    (push '|Localize| *Domains*)
    (make-instance '|LocalizeType|)))
```

———–

1.63.14 LyndonWord

— defclass LyndonWordType —

```
(defclass |LyndonWordType| (|OrderedSetType| |RetractableToType|)
  ((parents :initform '(|OrderedSet| |RetractableTo|))
   (name :initform "LyndonWord")
   (marker :initform 'domain)
   (abbreviation :initform 'LWORD)
   (comment :initform (list
     "Lyndon words over arbitrary (ordered) symbols:"
     "see Free Lie Algebras by C. Reutenauer (Oxford science publications)."
     "A Lyndon word is a word which is smaller than any of its right factors"
     "w.r.t. the pure lexicographical ordering."
     "If a and b are two Lyndon words such that a < b"
     "holds w.r.t lexicographical ordering then a*b is a Lyndon word."
     "Parenthesized Lyndon words can be generated from symbols by using the"
     "following rule:"
     "[[a,b],c] is a Lyndon word iff a*b < c <= b holds."
     "Lyndon words are internally represented by binary trees using the"
     "Magma domain constructor."
     "Two ordering are provided: lexicographic and"
     "length-lexicographic. "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |LyndonWord|
  (progn
    (push '|LyndonWord| *Domains*)
    (make-instance '|LyndonWordType|)))
```

———-

1.64 M

1.64.1 MachineComplex

— insane —

```
(defclass |MachineComplexType| (|ComplexCategoryType| |FortranMachineTypeCategoryType|)
  ((parents :initform '(|ComplexCategory| |FortranMachineTypeCategory|))
   (name :initform "MachineComplex")
   (marker :initform 'domain)
   (abbreviation :initform 'MCMPLX)
   (comment :initform (list
     "A domain which models the complex number representation"
     "used by machines in the AXIOM-NAG link."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MachineComplex|
  (progn
    (push '|MachineComplex| *Domains*)
    (make-instance '|MachineComplexType|)))
```

———-

1.64.2 MachineFloat

— insane —

```
(defclass |MachineFloatType| (|FloatingPointSystemType| |FortranMachineTypeCategoryType|)
  ((parents :initform '(|FloatingPointSystem| |FortranMachineTypeCategory|))
   (name :initform "MachineFloat")
   (marker :initform 'domain)
   (abbreviation :initform 'MFLOAT)
   (comment :initform (list
     "A domain which models the floating point representation"
     " used by machines in the AXIOM-NAG link."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |MachineFloat|
  (progn
    (push '|MachineFloat| *Domains*)
    (make-instance '|MachineFloatType|)))
```

———-

1.64.3 MachineInteger

— defclass MachineIntegerType —

```
(defclass |MachineIntegerType| (|IntegerNumberSystemType|); |FortranMachineTypeCategoryType|)
  ((parents :initform '(|IntegerNumberSystem| |FortranMachineTypeCategory|))
   (name :initform "MachineInteger")
   (marker :initform 'domain)
   (abbreviation :initform 'MINT)
   (comment :initform (list
     "A domain which models the integer representation"
     " used by machines in the AXIOM-NAG link."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MachineInteger|
  (progn
    (push '|MachineInteger| *Domains*)
    (make-instance '|MachineIntegerType|)))
```

———-

1.64.4 Magma

— defclass MagmaType —

```
(defclass |MagmaType| (|OrderedSetType| |RetractableToType|)
  ((parents :initform '(|OrderedSet| |RetractableTo|))
   (name :initform "Magma")
   (marker :initform 'domain)
   (abbreviation :initform 'MAGMA)
   (comment :initform (list
     "This type is the basic representation of"
     "parenthesized words (binary trees over arbitrary symbols)"
     "useful in LiePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Magma|
  (progn
    (push '|Magma| *Domains*)
    (make-instance '|MagmaType|)))
```

———-

1.64.5 MakeCachableSet

— defclass MakeCachableSetType —

```
(defclass |MakeCachableSetType| (|CachableSetType|)
  ((parents :initform '(|CachableSet|))
   (name :initform "MakeCachableSet")
   (marker :initform 'domain)
   (abbreviation :initform 'MKCHSET)
   (comment :initform (list
     "MakeCachableSet(S) returns a cachable set which is equal to S as a set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeCachableSet|
  (progn
    (push '|MakeCachableSet| *Domains*)
    (make-instance '|MakeCachableSetType|)))
```

———-

1.64.6 MathMLFormat

— defclass MathMLFormatType —

```
(defclass |MathMLFormatType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "MathMLFormat")
   (marker :initform 'domain)
   (abbreviation :initform 'MMLFORM)
   (comment :initform (list
     "This package is based on the TeXFormat domain by Robert S. Sutor"
     "MathMLFormat provides a coercion from OutputForm"
     "to MathML format."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MathMLFormat|
  (progn
    (push '|MathMLFormat| *Domains*)
    (make-instance '|MathMLFormatType|)))
```

———-

1.64.7 Matrix

— defclass MatrixType —

```
(defclass |MatrixType| (|ConvertibleToType| |MatrixCategoryType|)
  ((parents :initform '(|ConvertibleTo| |MatrixCategory|))
   (name :initform "Matrix")
   (marker :initform 'domain)
   (abbreviation :initform 'MATRIX)
   (comment :initform (list
     "Matrix is a matrix domain where 1-based indexing is used"
     "for both rows and columns."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Matrix|
  (progn
    (push '|Matrix| *Domains*)
    (make-instance '|MatrixType|)))
```

———-

1.64.8 ModMonic

— defclass ModMonicType —

```
(defclass |ModMonicType| (|UnivariatePolynomialCategoryType| |FiniteType|)
  ((parents :initform '(|UnivariatePolynomialCategory| |Finite|))
   (name :initform "ModMonic")
   (marker :initform 'domain)
   (abbreviation :initform 'MODMON)
   (comment :initform (list
     "This package has not been documented"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModMonic|
  (progn
    (push '|ModMonic| *Domains*)
    (make-instance '|ModMonicType|)))
```

———-

1.64.9 ModularField

— defclass ModularFieldType —

```
(defclass |ModularFieldType| (|FieldType|)
  ((parents :initform '(|Field|))
   (name :initform "ModularField")
   (marker :initform 'domain)
   (abbreviation :initform 'MODFIELD)
   (comment :initform (list
     "These domains are used for the factorization and gcds"
     "of univariate polynomials over the integers in order to work modulo"
     "different  primes."
     "See ModularRing, EuclideanModularRing"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularField|
  (progn
    (push '|ModularField| *Domains*)
    (make-instance '|ModularFieldType|)))
```

———-

1.64.10 ModularRing

— defclass ModularRingType —

```
(defclass |ModularRingType| (|RingType|)
  ((parents :initform '(|Ring|))
   (name :initform "ModularRing")
   (marker :initform 'domain)
   (abbreviation :initform 'MODRING)
   (comment :initform (list
     "These domains are used for the factorization and gcds"
     "of univariate polynomials over the integers in order to work modulo"
     "different  primes."
     "See EuclideanModularRing ,ModularField"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularRing|
  (progn
    (push '|ModularRing| *Domains*)
    (make-instance '|ModularRingType|)))
```

———-

1.64.11 ModuleMonomial

— defclass ModuleMonomialType —

```
(defclass |ModuleMonomialType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "ModuleMonomial")
   (marker :initform 'domain)
   (abbreviation :initform 'MODMONOM)
   (comment :initform (list
     "This package has no documentation"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModuleMonomial|
  (progn
    (push '|ModuleMonomial| *Domains*)
    (make-instance '|ModuleMonomialType|)))
```

———-

1.64.12 ModuleOperator

— defclass ModuleOperatorType —

```
(defclass |ModuleOperatorType| (|AlgebraType|
                                |CharacteristicNonZeroType|
                                |CharacteristicZeroType|
                                |EltableType|
                                |RetractableToType|)
  ((parents :initform '(|Algebra|
                        |CharacteristicNonZero|
                        |CharacteristicZero|
                        |Eltable|
                        |RetractableTo|))
   (name :initform "ModuleOperator")
   (marker :initform 'domain)
   (abbreviation :initform 'MODOP)
   (comment :initform (list
     "Algebra of ADDITIVE operators on a module."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModuleOperator|
  (progn
    (push '|ModuleOperator| *Domains*)
    (make-instance '|ModuleOperatorType|)))
```

———-

1.64.13 MoebiusTransform

— defclass MoebiusTransformType —

```
(defclass |MoebiusTransformType| (|GroupType|)
  ((parents :initform '(|Group|))
   (name :initform "MoebiusTransform")
   (marker :initform 'domain)
   (abbreviation :initform 'MOEBIUS)
   (comment :initform (list
     "MoebiusTransform(F) is the domain of fractional linear (Moebius)"
     "transformations over F. This a domain of 2-by-2 matrices acting on P1(F)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MoebiusTransform|
  (progn
    (push '|MoebiusTransform| *Domains*)
    (make-instance '|MoebiusTransformType|)))
```

———-

1.64.14 MonoidRing

— defclass MonoidRingType —

```
(defclass |MonoidRingType| (|RetractableToType|
                            |FiniteType|
                            |CharacteristicZeroType|
                            |CharacteristicNonZeroType|
                            |AlgebraType|)
  ((parents :initform '(|RetractableTo|
                        |Finite|
                        |CharacteristicZero|
                        |CharacteristicNonZero|
                        |Algebra|))
   (name :initform "MonoidRing")
   (marker :initform 'domain)
   (abbreviation :initform 'MRING)
   (comment :initform (list
     "MonoidRing(R,M), implements the algebra"
     "of all maps from the monoid M to the commutative ring R with"
     "finite support."
     "Multiplication of two maps f and g is defined"
     "to map an element c of M to the (convolution) sum over f(a)g(b)"
     "such that ab = c. Thus M can be identified with a canonical"
     "basis and the maps can also be considered as formal linear combinations"
     "of the elements in M. Scalar multiples of a basis element are called"
     "monomials. A prominent example is the class of polynomials"
     "where the monoid is a direct product of the natural numbers"
     "with pointwise addition. When M is"
     "FreeMonoid Symbol, one gets polynomials"
     "in infinitely many non-commuting variables. Another application"
     "area is representation theory of finite groups G, where modules"
     "over MonoidRing(R,G) are studied."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonoidRing|
  (progn
    (push '|MonoidRing| *Domains*)
    (make-instance '|MonoidRingType|)))
```

———-

1.64.15 Multiset

— defclass MultisetType —

```
(defclass |MultisetType| (|MultisetAggregateType|)
  ((parents :initform '(|MultisetAggregate|))
   (name :initform "Multiset")
   (marker :initform 'domain)
   (abbreviation :initform 'MSET)
   (comment :initform (list
     "A multiset is a set with multiplicities."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Multiset|
  (progn
    (push '|Multiset| *Domains*)
    (make-instance '|MultisetType|)))
```

———-

1.64.16 MultivariatePolynomial

— defclass MultivariatePolynomialType —

```
(defclass |MultivariatePolynomialType| (|PolynomialCategoryType|)
  ((parents :initform '(|PolynomialCategory|))
   (name :initform "MultivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'MPOLY)
   (comment :initform (list
     "This type is the basic representation of sparse recursive multivariate"
     "polynomials whose variables are from a user specified list of symbols."
     "The ordering is specified by the position of the variable in the list."
     "The coefficient ring may be non commutative,"
     "but the variables are assumed to commute."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariatePolynomial|
  (progn
    (push '|MultivariatePolynomial| *Domains*)
    (make-instance '|MultivariatePolynomialType|)))
```

———-

1.64.17 MyExpression

— defclass MyExpressionType —

```
(defclass |MyExpressionType| (|FunctionSpaceType| |CombinatorialOpsCategoryType|)
  ((parents :initform '(|FunctionSpace| |CombinatorialOpsCategory|))
   (name :initform "MyExpression")
   (marker :initform 'domain)
   (abbreviation :initform 'MYEXPR)
   (comment :initform (list
     "This domain has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MyExpression|
  (progn
    (push '|MyExpression| *Domains*)
    (make-instance '|MyExpressionType|)))
```

———-

1.64.18 MyUnivariatePolynomial

— defclass MyUnivariatePolynomialType —

```
(defclass |MyUnivariatePolynomialType| (|UnivariatePolynomialCategoryType|)
  ((parents :initform '(|UnivariatePolynomialCategory|))
   (name :initform "MyUnivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'MYUP)
   (comment :initform (list
     "This domain has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MyUnivariatePolynomial|
  (progn
    (push '|MyUnivariatePolynomial| *Domains*)
    (make-instance '|MyUnivariatePolynomialType|)))
```

——————-

1.65 N

1.65.1 NeitherSparseOrDensePowerSeries

— defclass NeitherSparseOrDensePowerSeriesType —

```
(defclass |NeitherSparseOrDensePowerSeriesType| (|LocalPowerSeriesCategoryType|
                                                  |LazyStreamAggregateType|)
  ((parents :initform '(|LocalPowerSeriesCategory| |LazyStreamAggregate|))
   (name :initform "NeitherSparseOrDensePowerSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'NSDPS)
   (comment :initform (list
     "This domain is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NeitherSparseOrDensePowerSeries|
  (progn
    (push '|NeitherSparseOrDensePowerSeries| *Domains*)
    (make-instance '|NeitherSparseOrDensePowerSeriesType|)))
```

——————-

1.65.2 NewSparseMultivariatePolynomial

— defclass NewSparseMultivariatePolynomialType —

```
(defclass |NewSparseMultivariatePolynomialType| (|RecursivePolynomialCategoryType|)
  ((parents :initform '(|RecursivePolynomialCategory|))
   (name :initform "NewSparseMultivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'NSMP)
   (comment :initform (list
     "A post-facto extension for SMP in order"
     "to speed up operations related to pseudo-division and gcd."
     "This domain is based on the NSUP constructor which is"
     "itself a post-facto extension of the SUP constructor."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewSparseMultivariatePolynomial|
  (progn
    (push '|NewSparseMultivariatePolynomial| *Domains*)
    (make-instance '|NewSparseMultivariatePolynomialType|)))
```

——————-

1.65.3 NewSparseUnivariatePolynomial

— defclass NewSparseUnivariatePolynomialType —

```
(defclass |NewSparseUnivariatePolynomialType| (|UnivariatePolynomialCategoryType|)
  ((parents :initform '(|UnivariatePolynomialCategory|))
   (name :initform "NewSparseUnivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'NSUP)
   (comment :initform (list
     "A post-facto extension for SUP in order"
     "to speed up operations related to pseudo-division and gcd for"
     "both SUP and, consequently, NSMP."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewSparseUnivariatePolynomial|
  (progn
    (push '|NewSparseUnivariatePolynomial| *Domains*)
    (make-instance '|NewSparseUnivariatePolynomialType|)))
```

——————-

1.65.4 None

— defclass NoneType —

```
(defclass |NoneType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "None")
   (marker :initform 'domain)
   (abbreviation :initform 'NONE)
   (comment :initform (list
     "None implements a type with no objects. It is mainly"
     "used in technical situations where such a thing is needed (for example,"
     "the interpreter and some of the internal Expression code)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |None|
  (progn
    (push '|None| *Domains*)
    (make-instance '|NoneType|)))
```

———-

1.65.5 NonNegativeInteger

— defclass NonNegativeIntegerType —

```
(defclass |NonNegativeIntegerType| (|OrderedAbelianMonoidSupType| |MonoidType|)
  ((parents :initform '(|OrderedAbelianMonoidSup| |Monoid|))
   (name :initform "NonNegativeInteger")
   (marker :initform 'domain)
   (abbreviation :initform 'NNI)
   (comment :initform (list
     "NonNegativeInteger provides functions for non negative integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonNegativeInteger|
  (progn
    (push '|NonNegativeInteger| *Domains*)
    (make-instance '|NonNegativeIntegerType|)))
```

———-

1.65.6 NottinghamGroup

— defclass NottinghamGroupType —

```
(defclass |NottinghamGroupType| (|GroupType|)
  ((parents :initform '(|Group|))
   (name :initform "NottinghamGroup")
   (marker :initform 'domain)
   (abbreviation :initform 'NOTTING)
   (comment :initform (list
     "This is an implmenentation of the Nottingham Group"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NottinghamGroup|
  (progn
    (push '|NottinghamGroup| *Domains*)
    (make-instance '|NottinghamGroupType|)))
```

———–

1.65.7 NumericalIntegrationProblem

— defclass NumericalIntegrationProblemType —

```
(defclass |NumericalIntegrationProblemType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "NumericalIntegrationProblem")
   (marker :initform 'domain)
   (abbreviation :initform 'NIPROB)
   (comment :initform (list
     "NumericalIntegrationProblem is a domain"
     "for the representation of Numerical Integration problems for use"
     "by ANNA."
     " "
     "The representation is a Union of two record types - one for integration of"
     "a function of one variable:"
     " "
     "Record(var:Symbol,"
     "fn:Expression DoubleFloat,"
     "range:Segment OrderedCompletion DoubleFloat,"
     "abserr:DoubleFloat,"
     "relerr:DoubleFloat,)"
     " "
     "and one for multivariate integration:"
     " "
     "Record(fn:Expression DoubleFloat,"
     "range:List Segment OrderedCompletion DoubleFloat,"
     "abserr:DoubleFloat,"
     "relerr:DoubleFloat,)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalIntegrationProblem|
  (progn
    (push '|NumericalIntegrationProblem| *Domains*)
    (make-instance '|NumericalIntegrationProblemType|)))
```

———-

1.65.8 NumericalODEProblem

— defclass NumericalODEProblemType —

```
(defclass |NumericalODEProblemType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "NumericalODEProblem")
   (marker :initform 'domain)
   (abbreviation :initform 'ODEPROB)
   (comment :initform (list
     "NumericalODEProblem is a domain"
     "for the representation of Numerical ODE problems for use"
     "by ANNA."
     " "
     "The representation is of type:"
     " "
     "Record(xinit:DoubleFloat,"
     "xend:DoubleFloat,"
     "fn:Vector Expression DoubleFloat,"
     "yinit:List DoubleFloat,intvals:List DoubleFloat,"
     "g:Expression DoubleFloat,abserr:DoubleFloat,"
     "relerr:DoubleFloat)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalODEProblem|
  (progn
    (push '|NumericalODEProblem| *Domains*)
    (make-instance '|NumericalODEProblemType|)))
```

———-

1.65.9 NumericalOptimizationProblem

— defclass NumericalOptimizationProblemType —

```
(defclass |NumericalOptimizationProblemType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "NumericalOptimizationProblem")
   (marker :initform 'domain)
   (abbreviation :initform 'OPTPROB)
   (comment :initform (list
     "NumericalOptimizationProblem is a domain"
     "for the representation of Numerical Optimization problems for use"
     "by ANNA."
     " "
     "The representation is a Union of two record types - one for optimization of"
     "a single function of one or more variables:"
     " "
     "Record("
     "fn:Expression DoubleFloat,"
     "init:List DoubleFloat,"
     "lb:List OrderedCompletion DoubleFloat,"
     "cf:List Expression DoubleFloat,"
     "ub:List OrderedCompletion DoubleFloat)"
     " "
     "and one for least-squares problems that is, optimization of a set of"
     "observations of a data set:"
     " "
     "Record(lfn:List Expression DoubleFloat,"
     "init:List DoubleFloat).")
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalOptimizationProblem|
  (progn
    (push '|NumericalOptimizationProblem| *Domains*)
    (make-instance '|NumericalOptimizationProblemType|)))
```

———-

1.65.10 NumericalPDEProblem

— defclass NumericalPDEProblemType —

```
(defclass |NumericalPDEProblemType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "NumericalPDEProblem")
   (marker :initform 'domain)
   (abbreviation :initform 'PDEPROB)
   (comment :initform (list
     "NumericalPDEProblem is a domain"
     "for the representation of Numerical PDE problems for use"
     "by ANNA."
     " "
     "The representation is of type:"
     " "
     "Record(pde:List Expression DoubleFloat,"
     "constraints:List PDEC,"
     "f:List List Expression DoubleFloat,"
     "st:String,"
     "tol:DoubleFloat)"
     " "
     "where PDEC is of type:"
     " "
     "Record(start:DoubleFloat,"
     "finish:DoubleFloat,"
     "grid:NonNegativeInteger,"
     "boundaryType:Integer,"
     "dStart:Matrix DoubleFloat,"
     "dFinish:Matrix DoubleFloat)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalPDEProblem|
  (progn
    (push '|NumericalPDEProblem| *Domains*)
    (make-instance '|NumericalPDEProblemType|)))
```

————-

1.66 O

1.66.1 Octonion

— defclass OctonionType —

```
(defclass |OctonionType| (|OctonionCategoryType|)
  ((parents :initform '(|OctonionCategory|))
   (name :initform "Octonion")
   (marker :initform 'domain)
   (abbreviation :initform 'OCT)
   (comment :initform (list
     "Octonion implements octonions (Cayley-Dixon algebra) over a"
     "commutative ring, an eight-dimensional non-associative"
     "algebra, doubling the quaternions in the same way as doubling"
     "the complex numbers to get the quaternions"
     "the main constructor function is octon which takes 8"
     "arguments: the real part, the i imaginary part, the j"
     "imaginary part, the k imaginary part, (as with quaternions)"
     "and in addition the imaginary parts E, I, J, K."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Octonion|
  (progn
    (push '|Octonion| *Domains*)
    (make-instance '|OctonionType|)))
```

————-

1.66.2 ODEIntensityFunctionsTable

— defclass ODEIntensityFunctionsTableType —

```
(defclass |ODEIntensityFunctionsTableType| (|AxiomClass|)
  ((parents :initform '())
   (name :initform "ODEIntensityFunctionsTable")
   (marker :initform 'domain)
   (abbreviation :initform 'ODEIFTBL)
   (comment :initform (list
     "ODEIntensityFunctionsTable() provides a dynamic table and a set of"
     "functions to store details found out about sets of ODE's."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ODEIntensityFunctionsTable|
  (progn
    (push '|ODEIntensityFunctionsTable| *Domains*)
    (make-instance '|ODEIntensityFunctionsTableType|)))
```

———-

1.66.3 OneDimensionalArray

— defclass OneDimensionalArrayType —

```
(defclass |OneDimensionalArrayType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "OneDimensionalArray")
   (marker :initform 'domain)
   (abbreviation :initform 'ARRAY1)
   (comment :initform (list
     "This is the domain of 1-based one dimensional arrays"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OneDimensionalArray|
  (progn
    (push '|OneDimensionalArray| *Domains*)
    (make-instance '|OneDimensionalArrayType|)))
```

———-

1.66.4 OnePointCompletion

— defclass OnePointCompletionType —

```
(defclass |OnePointCompletionType| (|OrderedRingType| |FullyRetractableToType|)
  ((parents :initform '(|OrderedRing| |FullyRetractableTo|))
   (name :initform "OnePointCompletion")
   (marker :initform 'domain)
   (abbreviation :initform 'ONECOMP)
   (comment :initform (list
     "Completion with infinity."
     "Adjunction of a complex infinity to a set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OnePointCompletion|
  (progn
    (push '|OnePointCompletion| *Domains*)
    (make-instance '|OnePointCompletionType|)))
```

————-

1.66.5 OpenMathConnection

— defclass OpenMathConnectionType —

```
(defclass |OpenMathConnectionType| (|AxiomClass|)
  ((parents :initform '())
   (name :initform "OpenMathConnection")
   (marker :initform 'domain)
   (abbreviation :initform 'OMCONN)
   (comment :initform (list
     "OpenMathConnection provides low-level functions"
     "for handling connections to and from OpenMathDevices."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathConnection|
  (progn
    (push '|OpenMathConnection| *Domains*)
    (make-instance '|OpenMathConnectionType|)))
```

————-

1.66.6 OpenMathDevice

— defclass OpenMathDeviceType —

```
(defclass |OpenMathDeviceType| (|AxiomClass|)
  ((parents :initform '())
   (name :initform "OpenMathDevice")
   (marker :initform 'domain)
   (abbreviation :initform 'OMDEV)
   (comment :initform (list
     "OpenMathDevice provides support for reading"
     "and writing openMath objects to files, strings etc.  It also provides"
     "access to low-level operations from within the interpreter."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathDevice|
  (progn
    (push '|OpenMathDevice| *Domains*)
    (make-instance '|OpenMathDeviceType|)))
```

———-

1.66.7 OpenMathEncoding

— defclass OpenMathEncodingType —

```
(defclass |OpenMathEncodingType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "OpenMathEncoding")
   (marker :initform 'domain)
   (abbreviation :initform 'OMENC)
   (comment :initform (list
     "OpenMathEncoding is the set of valid OpenMath encodings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathEncoding|
  (progn
    (push '|OpenMathEncoding| *Domains*)
    (make-instance '|OpenMathEncodingType|)))
```

———-

1.66.8 OpenMathError

— defclass OpenMathErrorType —

```
(defclass |OpenMathErrorType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "OpenMathError")
   (marker :initform 'domain)
   (abbreviation :initform 'OMERR)
   (comment :initform (list
     "OpenMathError is the domain of OpenMath errors."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathError|
  (progn
    (push '|OpenMathError| *Domains*)
    (make-instance '|OpenMathErrorType|)))
```

———-

1.66.9 OpenMathErrorKind

— defclass OpenMathErrorKindType —

```
(defclass |OpenMathErrorKindType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "OpenMathErrorKind")
   (marker :initform 'domain)
   (abbreviation :initform 'OMERRK)
   (comment :initform (list
     "OpenMathErrorKind represents different kinds"
     "of OpenMath errors: specifically parse errors, unknown CD or symbol"
     "errors, and read errors."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathErrorKind|
  (progn
    (push '|OpenMathErrorKind| *Domains*)
    (make-instance '|OpenMathErrorKindType|)))
```

———-

1.66.10 Operator

— defclass OperatorType —

```
(defclass |OperatorType| (|AlgebraType|
                          |CharacteristicNonZeroType|
                          |CharacteristicZeroType|
                          |EltableType|
                          |RetractableToType|)
  ((parents :initform '(|Algebra|
                        |CharacteristicNonZero|
                        |CharacteristicZero|
                        |Eltable|
                        |RetractableTo|))
   (name :initform "Operator")
   (marker :initform 'domain)
   (abbreviation :initform 'OP)
   (comment :initform (list
     "Algebra of ADDITIVE operators over a ring."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Operator|
  (progn
    (push '|Operator| *Domains*)
    (make-instance '|OperatorType|)))
```

———–

1.66.11 OppositeMonogenicLinearOperator

— defclass OppositeMonogenicLinearOperatorType —

```
(defclass |OppositeMonogenicLinearOperatorType| (|DifferentialRingType|
                                                 |MonogenicLinearOperatorType|)
  ((parents :initform '(|DifferentialRing| |MonogenicLinearOperator|))
   (name :initform "OppositeMonogenicLinearOperator")
   (marker :initform 'domain)
   (abbreviation :initform 'OMLO)
   (comment :initform (list
     "This constructor creates the MonogenicLinearOperator domain"
     "which is 'opposite' in the ring sense to P."
     "That is, as sets P = $ but a * b in $ is equal to"
     "b * a in P."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OppositeMonogenicLinearOperator|
  (progn
    (push '|OppositeMonogenicLinearOperator| *Domains*)
    (make-instance '|OppositeMonogenicLinearOperatorType|)))
```

———–

1.66.12 OrderedCompletion

— defclass OrderedCompletionType —

```
(defclass |OrderedCompletionType| (|OrderedRingType| |FullyRetractableToType|)
  ((parents :initform '(|OrderedRing| |FullyRetractableTo|))
   (name :initform "OrderedCompletion")
   (marker :initform 'domain)
   (abbreviation :initform 'ORDCOMP)
   (comment :initform (list
     "Completion with + and - infinity."
     "Adjunction of two real infinites quantities to a set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedCompletion|
  (progn
    (push '|OrderedCompletion| *Domains*)
    (make-instance '|OrderedCompletionType|)))
```

———-

1.66.13 OrderedDirectProduct

— defclass OrderedDirectProductType —

```
(defclass |OrderedDirectProductType| (|DirectProductCategoryType|)
  ((parents :initform '(|DirectProductCategory|))
   (name :initform "OrderedDirectProduct")
   (marker :initform 'domain)
   (abbreviation :initform 'ODP)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedDirectProduct|
  (progn
    (push '|OrderedDirectProduct| *Domains*)
    (make-instance '|OrderedDirectProductType|)))
```

———-

1.66.14 OrderedFreeMonoid

— defclass OrderedFreeMonoidType —

```
(defclass |OrderedFreeMonoidType| (|RetractableToType| |OrderedMonoidType|)
  ((parents :initform '(|RetractableTo| |OrderedMonoid|))
   (name :initform "OrderedFreeMonoid")
   (marker :initform 'domain)
   (abbreviation :initform 'OFMONOID)
   (comment :initform (list
     "The free monoid on a set S is the monoid of finite products of"
     "the form reduce(*,[si ** ni]) where the si's are in S, and the ni's"
     "are non-negative integers. The multiplication is not commutative."
     "For two elements x and y the relation x < y"
     "holds if either length(x) < length(y) holds or if these lengths"
     "are equal and if x is smaller than y w.r.t. the"
     "lexicographical ordering induced by S."
     "This domain inherits implementation from FreeMonoid."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedFreeMonoid|
  (progn
    (push '|OrderedFreeMonoid| *Domains*)
    (make-instance '|OrderedFreeMonoidType|)))
```

———-

1.66.15 OrderedVariableList

— defclass OrderedVariableListType —

```
(defclass |OrderedVariableListType| (|ConvertibleToType| |OrderedFiniteType|)
 ((parents :initform '(|ConvertibleTo| |OrderedFinite|))
  (name :initform "OrderedVariableList")
  (marker :initform 'domain)
  (abbreviation :initform 'OVAR)
  (comment :initform (list
    "This domain implements ordered variables"))
  (argslist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |OrderedVariableList|
 (progn
  (push '|OrderedVariableList| *Domains*)
  (make-instance '|OrderedVariableListType|)))
```

———-

1.66.16 OrderlyDifferentialPolynomial

— defclass OrderlyDifferentialPolynomialType —

```
(defclass |OrderlyDifferentialPolynomialType| (|DifferentialPolynomialCategoryType|)
 ((parents :initform '(|DifferentialPolynomialCategory|))
   (name :initform "OrderlyDifferentialPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'ODPOL)
   (comment :initform (list
     "OrderlyDifferentialPolynomial implements"
     "an ordinary differential polynomial ring in arbitrary number"
     "of differential indeterminates, with coefficients in a"
     "ring.  The ranking on the differential indeterminate is orderly."
     "This is analogous to the domain Polynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderlyDifferentialPolynomial|
  (progn
    (push '|OrderlyDifferentialPolynomial| *Domains*)
    (make-instance '|OrderlyDifferentialPolynomialType|)))
```

———-

1.66.17 OrderlyDifferentialVariable

— defclass OrderlyDifferentialVariableType —

```
(defclass |OrderlyDifferentialVariableType| (|DifferentialVariableCategoryType|)
  ((parents :initform '(|DifferentialVariableCategory|))
   (name :initform "OrderlyDifferentialVariable")
   (marker :initform 'domain)
   (abbreviation :initform 'ODVAR)
   (comment :initform (list
     "OrderlyDifferentialVariable adds a commonly used orderly"
     "ranking to the set of derivatives of an ordered list of differential"
     "indeterminates.  An orderly ranking is a ranking < of the"
     "derivatives with the property that for two derivatives u and v,"
     "u < v if the order of u is less than that of v."
     "This domain belongs to DifferentialVariableCategory.  It"
     "defines weight to be just order, and it"
     "defines an orderly ranking < on derivatives u via the"
     "lexicographic order on the pair"
     "(order(u), variable}(u)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderlyDifferentialVariable|
  (progn
    (push '|OrderlyDifferentialVariable| *Domains*)
    (make-instance '|OrderlyDifferentialVariableType|)))
```

———-

1.66.18 OrdinaryDifferentialRing

— defclass OrdinaryDifferentialRingType —

```
(defclass |OrdinaryDifferentialRingType| (|FieldType| |DifferentialRingType|)
  ((parents :initform '(|Field| |DifferentialRing|))
   (name :initform "OrdinaryDifferentialRing")
   (marker :initform 'domain)
   (abbreviation :initform 'ODR)
   (comment :initform (list
     "This constructor produces an ordinary differential ring from"
     "a partial differential ring by specifying a variable."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdinaryDifferentialRing|
  (progn
    (push '|OrdinaryDifferentialRing| *Domains*)
    (make-instance '|OrdinaryDifferentialRingType|)))
```

———-

1.66.19 OrdinaryWeightedPolynomials

— defclass OrdinaryWeightedPolynomialsType —

```
(defclass |OrdinaryWeightedPolynomialsType| (|AlgebraType|)
  ((parents :initform '(|Algebra|))
   (name :initform "OrdinaryWeightedPolynomials")
   (marker :initform 'domain)
   (abbreviation :initform 'OWP)
   (comment :initform (list
     "This domain represents truncated weighted polynomials over the"
     "'Polynomial' type. The variables must be"
     "specified, as must the weights."
     "The representation is sparse"
     "in the sense that only non-zero terms are represented."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdinaryWeightedPolynomials|
  (progn
    (push '|OrdinaryWeightedPolynomials| *Domains*)
    (make-instance '|OrdinaryWeightedPolynomialsType|)))
```

———-

1.66.20 OrdSetInts

— defclass OrdSetIntsType —

```
(defclass |OrdSetIntsType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "OrdSetInts")
   (marker :initform 'domain)
   (abbreviation :initform 'OSI)
   (comment :initform (list
     "A domain used in order to take the free R-module on the"
     "Integers I.  This is actually the forgetful functor from OrderedRings"
     "to OrderedSets applied to I"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdSetInts|
  (progn
    (push '|OrdSetInts| *Domains*)
    (make-instance '|OrdSetIntsType|)))
```

———-

1.66.21 OutputForm

— defclass OutputFormType —

```
(defclass |OutputFormType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "OutputForm")
   (marker :initform 'domain)
   (abbreviation :initform 'OUTFORM)
   (comment :initform (list
     "This domain is used to create and manipulate mathematical expressions"
     "for output.  It is intended to provide an insulating layer between"
     "the expression rendering software (for example, FORTRAN or TeX) and"
     "the output coercions in the various domains."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OutputForm|
  (progn
    (push '|OutputForm| *Domains*)
    (make-instance '|OutputFormType|)))
```

———-

1.67 P

1.67.1 PAdicInteger

— defclass PAdicIntegerType —

```
(defclass |PAdicIntegerType| (|PAdicIntegerCategoryType|)
  ((parents :initform '(|PAdicIntegerCategory|))
   (name :initform "PAdicInteger")
   (marker :initform 'domain)
   (abbreviation :initform 'PADIC)
   (comment :initform (list
     "Stream-based implementation of Zp: p-adic numbers are represented as"
     "sum(i = 0.., a[i] * p^i), where the a[i] lie in 0,1,...,(p - 1)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicInteger|
  (progn
    (push '|PAdicInteger| *Domains*)
    (make-instance '|PAdicIntegerType|)))
```

———-

1.67.2 PAdicRational

— defclass PAdicRationalType —

```
(defclass |PAdicRationalType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "PAdicRational")
   (marker :initform 'domain)
   (abbreviation :initform 'PADICRAT)
   (comment :initform (list
     "Stream-based implementation of Qp: numbers are represented as"
     " sum(i = k.., a[i] * p^i) where the a[i] lie in 0,1,...,(p - 1)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicRational|
  (progn
    (push '|PAdicRational| *Domains*)
    (make-instance '|PAdicRationalType|)))
```

———-

1.67.3 PAdicRationalConstructor

— defclass PAdicRationalConstructorType —

```
(defclass |PAdicRationalConstructorType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "PAdicRationalConstructor")
   (marker :initform 'domain)
   (abbreviation :initform 'PADICRC)
   (comment :initform (list
     "This is the category of stream-based representations of Qp."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicRationalConstructor|
  (progn
    (push '|PAdicRationalConstructor| *Domains*)
    (make-instance '|PAdicRationalConstructorType|)))
```

———-

1.67.4 Palette

— defclass PaletteType —

```
(defclass |PaletteType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Palette")
   (marker :initform 'domain)
   (abbreviation :initform 'PALETTE)
   (comment :initform (list
     "This domain describes four groups of color shades (palettes)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Palette|
  (progn
    (push '|Palette| *Domains*)
    (make-instance '|PaletteType|)))
```

———-

1.67.5 ParametricPlaneCurve

— defclass ParametricPlaneCurveType —

```
(defclass |ParametricPlaneCurveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricPlaneCurve")
   (marker :initform 'domain)
   (abbreviation :initform 'PARPCURV)
   (comment :initform (list
     "ParametricPlaneCurve is used for plotting parametric plane"
     "curves in the affine plane."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricPlaneCurve|
  (progn
    (push '|ParametricPlaneCurve| *Domains*)
    (make-instance '|ParametricPlaneCurveType|)))
```

———-

1.67.6 ParametricSpaceCurve

— defclass ParametricSpaceCurveType —

```
(defclass |ParametricSpaceCurveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricSpaceCurve")
   (marker :initform 'domain)
   (abbreviation :initform 'PARSCURV)
   (comment :initform (list
     "ParametricSpaceCurve is used for plotting parametric space"
     "curves in affine 3-space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSpaceCurve|
  (progn
    (push '|ParametricSpaceCurve| *Domains*)
    (make-instance '|ParametricSpaceCurveType|)))
```

———-

1.67.7 ParametricSurface

— defclass ParametricSurfaceType —

```
(defclass |ParametricSurfaceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricSurface")
   (marker :initform 'domain)
   (abbreviation :initform 'PARSURF)
   (comment :initform (list
     "ParametricSurface is used for plotting parametric surfaces in"
     "affine 3-space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSurface|
  (progn
    (push '|ParametricSurface| *Domains*)
    (make-instance '|ParametricSurfaceType|)))
```

———-

1.67.8 PartialFraction

— defclass PartialFractionType —

```
(defclass |PartialFractionType| (|FieldType|)
  ((parents :initform '(|Field|))
   (name :initform "PartialFraction")
   (marker :initform 'domain)
   (abbreviation :initform 'PFR)
   (comment :initform (list
     "The domain PartialFraction implements partial fractions"
     "over a euclidean domain R. This requirement on the"
     "argument domain allows us to normalize the fractions.  Of"
     "particular interest are the 2 forms for these fractions.  The"
     "``compact'' form has only one fractional term per prime in the"
     "denominator, while the ``p-adic'' form expands each numerator"
     "p-adically via the prime p in the denominator.  For computational"
     "efficiency, the compact form is used, though the p-adic form may"
     "be gotten by calling the function padicFraction}.  For a"
     "general euclidean domain, it is not known how to factor the"
     "denominator.  Thus the function partialFraction takes as its"
     "second argument an element of Factored(R)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialFraction|
  (progn
    (push '|PartialFraction| *Domains*)
    (make-instance '|PartialFractionType|)))
```

———-

1.67.9 Partition

— defclass PartitionType —

```
(defclass |PartitionType| (|ConvertibleToType| |OrderedCancellationAbelianMonoidType|)
  ((parents :initform '(|ConvertibleTo| |OrderedCancellationAbelianMonoid|))
   (name :initform "Partition")
   (marker :initform 'domain)
   (abbreviation :initform 'PRTITION)
   (comment :initform (list
     "Partition is an OrderedCancellationAbelianMonoid which is used"
     "as the basis for symmetric polynomial representation of the"
     "sums of powers in SymmetricPolynomial.  Thus, (5 2 2 1) will"
     "represent s5 * s2**2 * s1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Partition|
  (progn
    (push '|Partition| *Domains*)
    (make-instance '|PartitionType|)))
```

———-

1.67.10 Pattern

— defclass PatternType —

```
(defclass |PatternType| (|SetCategoryType| |RetractableToType|)
  ((parents :initform '(|SetCategory| |RetractableTo|))
   (name :initform "Pattern")
   (marker :initform 'domain)
   (abbreviation :initform 'PATTERN)
   (comment :initform (list
     "Patterns for use by the pattern matcher."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Pattern|
  (progn
    (push '|Pattern| *Domains*)
    (make-instance '|PatternType|)))
```

———-

1.67.11 PatternMatchListResult

— defclass PatternMatchListResultType —

```
(defclass |PatternMatchListResultType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "PatternMatchListResult")
   (marker :initform 'domain)
   (abbreviation :initform 'PATLRES)
   (comment :initform (list
     "A PatternMatchListResult is an object internally returned by the"
     "pattern matcher when matching on lists."
     "It is either a failed match, or a pair of PatternMatchResult,"
     "one for atoms (elements of the list), and one for lists."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchListResult|
  (progn
    (push '|PatternMatchListResult| *Domains*)
    (make-instance '|PatternMatchListResultType|)))
```

———-

1.67.12 PatternMatchResult

— defclass PatternMatchResultType —

```
(defclass |PatternMatchResultType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "PatternMatchResult")
   (marker :initform 'domain)
   (abbreviation :initform 'PATRES)
   (comment :initform (list
     "A PatternMatchResult is an object internally returned by the"
     "pattern matcher; It is either a failed match, or a list of"
     "matches of the form (var, expr) meaning that the variable var"
     "matches the expression expr."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchResult|
  (progn
    (push '|PatternMatchResult| *Domains*)
    (make-instance '|PatternMatchResultType|)))
```

———-

1.67.13 PendantTree

— defclass PendantTreeType —

```
(defclass |PendantTreeType| (|BinaryRecursiveAggregateType|)
  ((parents :initform '(|BinaryRecursiveAggregate|))
   (name :initform "PendantTree")
   (marker :initform 'domain)
   (abbreviation :initform 'PENDTREE)
   (comment :initform (list
     "A PendantTree(S) is either a leaf? and is an S or has"
     "a left and a right both PendantTree(S)'s"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PendantTree|
  (progn
    (push '|PendantTree| *Domains*)
    (make-instance '|PendantTreeType|)))
```

———-

1.67.14 Permutation

— defclass PermutationType —

```
(defclass |PermutationType| (|PermutationCategoryType|)
  ((parents :initform '(|PermutationCategory|))
   (name :initform "Permutation")
   (marker :initform 'domain)
   (abbreviation :initform 'PERM)
   (comment :initform (list
     "Permutation(S) implements the group of all bijections"
     "on a set S, which move only a finite number of points."
     "A permutation is considered as a map from S into S. In particular"
     "multiplication is defined as composition of maps:"
     "pi1 * pi2 = pi1 o pi2."
     "The internal representation of permuatations are two lists"
     "of equal length representing preimages and images."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Permutation|
  (progn
    (push '|Permutation| *Domains*)
    (make-instance '|PermutationType|)))
```

———-

1.67.15 PermutationGroup

— defclass PermutationGroupType —

```
(defclass |PermutationGroupType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "PermutationGroup")
   (marker :initform 'domain)
   (abbreviation :initform 'PERMGRP)
   (comment :initform (list
     "PermutationGroup implements permutation groups acting"
     "on a set S, all subgroups of the symmetric group of S,"
     "represented as a list of permutations (generators). Note that"
     "therefore the objects are not members of the Axiom category"
     "Group."
     "Using the idea of base and strong generators by Sims,"
     "basic routines and algorithms"
     "are implemented so that the word problem for"
     "permutation groups can be solved."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PermutationGroup|
  (progn
    (push '|PermutationGroup| *Domains*)
    (make-instance '|PermutationGroupType|)))
```

———-

1.67.16 Pi

— defclass PiType —

```
(defclass |PiType| (|CharacteristicZeroType|
                    |FieldType|
                    |RealConstantType|
                    |RetractableToType|)
  ((parents :initform '(|CharacteristicZero|
                        |Field|
                        |RealConstant|
                        |RetractableTo|))
   (name :initform "Pi")
   (marker :initform 'domain)
   (abbreviation :initform 'HACKPI)
   (comment :initform (list
     "Symbolic fractions in %pi with integer coefficients;"
     "The point for using Pi as the default domain for those fractions"
     "is that Pi is coercible to the float types, and not Expression."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Pi|
  (progn
    (push '|Pi| *Domains*)
    (make-instance '|PiType|)))
```

———-

1.67.17 PlaneAlgebraicCurvePlot

— defclass PlaneAlgebraicCurvePlotType —

```
(defclass |PlaneAlgebraicCurvePlotType| (|PlottablePlaneCurveCategoryType|)
  ((parents :initform '(|PlottablePlaneCurveCategory|))
   (name :initform "PlaneAlgebraicCurvePlot")
   (marker :initform 'domain)
   (abbreviation :initform 'ACPLOT)
   (comment :initform (list
     "Plot a NON-SINGULAR plane algebraic curve p(x,y) = 0."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlaneAlgebraicCurvePlot|
  (progn
    (push '|PlaneAlgebraicCurvePlot| *Domains*)
    (make-instance '|PlaneAlgebraicCurvePlotType|)))
```

———-

1.67.18 Places

— defclass PlacesType —

```
(defclass |PlacesType| (|PlacesCategoryType|)
  ((parents :initform '(|PlacesCategory|))
   (name :initform "Places")
   (marker :initform 'domain)
   (abbreviation :initform 'PLACES)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Places|
  (progn
    (push '|Places| *Domains*)
    (make-instance '|PlacesType|)))
```

———-

1.67.19 PlacesOverPseudoAlgebraicClosureOfFiniteField

— defclass PlacesOverPseudoAlgebraicClosureOfFiniteFieldType —

```
(defclass |PlacesOverPseudoAlgebraicClosureOfFiniteFieldType| (|PlacesCategoryType|)
  ((parents :initform '(|PlacesCategory|))
   (name :initform "PlacesOverPseudoAlgebraicClosureOfFiniteField")
   (marker :initform 'domain)
   (abbreviation :initform 'PLACESPS)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlacesOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|PlacesOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|PlacesOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.67.20 Plcs

— defclass PlcsType —

```
(defclass |PlcsType| (|PlacesCategoryType|)
  ((parents :initform '(|PlacesCategory|))
   (name :initform "Plcs")
   (marker :initform 'domain)
   (abbreviation :initform 'PLCS)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Plcs|
  (progn
    (push '|Plcs| *Domains*)
    (make-instance '|PlcsType|)))
```

———-

1.67.21 Plot

— defclass PlotType —

```
(defclass |PlotType| (|PlottablePlaneCurveCategoryType|)
  ((parents :initform '(|PlottablePlaneCurveCategory|))
   (name :initform "Plot")
   (marker :initform 'domain)
   (abbreviation :initform 'PLOT)
   (comment :initform (list
     "The Plot domain supports plotting of functions defined over a"
     "real number system.  A real number system is a model for the real"
     "numbers and as such may be an approximation.  For example"
     "floating point numbers and infinite continued fractions."
     "The facilities at this point are limited to 2-dimensional plots"
     "or either a single function or a parametric function."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Plot|
  (progn
    (push '|Plot| *Domains*)
    (make-instance '|PlotType|)))
```

———-

1.67.22 Plot3D

— defclass Plot3DType —

```
(defclass |Plot3DType| (|PlottableSpaceCurveCategoryType|)
  ((parents :initform '(|PlottableSpaceCurveCategory|))
   (name :initform "Plot3D")
   (marker :initform 'domain)
   (abbreviation :initform 'PLOT3D)
   (comment :initform (list
     "Plot3D supports parametric plots defined over a real"
     "number system.  A real number system is a model for the real"
     "numbers and as such may be an approximation.  For example,"
     "floating point numbers and infinite continued fractions are"
     "real number systems. The facilities at this point are limited"
     "to 3-dimensional parametric plots."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Plot3D|
  (progn
    (push '|Plot3D| *Domains*)
    (make-instance '|Plot3DType|)))
```

———-

1.67.23 PoincareBirkhoffWittLyndonBasis

— defclass PoincareBirkhoffWittLyndonBasisType —

```
(defclass |PoincareBirkhoffWittLyndonBasisType| (|OrderedSetType| |RetractableToType|)
  ((parents :initform '(|OrderedSet| |RetractableTo|))
   (name :initform "PoincareBirkhoffWittLyndonBasis")
   (marker :initform 'domain)
   (abbreviation :initform 'PBWLB)
   (comment :initform (list
     "This domain provides the internal representation"
     "of polynomials in non-commutative variables written"
     "over the Poincare-Birkhoff-Witt basis."
     "See the XPBWPolynomial domain constructor."
     "See Free Lie Algebras by C. Reutenauer "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PoincareBirkhoffWittLyndonBasis|
  (progn
    (push '|PoincareBirkhoffWittLyndonBasis| *Domains*)
    (make-instance '|PoincareBirkhoffWittLyndonBasisType|)))
```

———-

1.67.24 Point

— defclass PointType —

```
(defclass |PointType| (|PointCategoryType|)
  ((parents :initform '(|PointCategory|))
   (name :initform "Point")
   (marker :initform 'domain)
   (abbreviation :initform 'POINT)
   (comment :initform (list
     "This domain implements points in coordinate space"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Point|
  (progn
    (push '|Point| *Domains*)
    (make-instance '|PointType|)))
```

———–

1.67.25 Polynomial

— defclass PolynomialType —

```
(defclass |PolynomialType| (|PolynomialCategoryType|)
  ((parents :initform '(|PolynomialCategory|))
   (name :initform "Polynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'POLY)
   (comment :initform (list
     "This type is the basic representation of sparse recursive multivariate"
     "polynomials whose variables are arbitrary symbols. The ordering"
     "is alphabetic determined by the Symbol type."
     "The coefficient ring may be non commutative,"
     "but the variables are assumed to commute."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Polynomial|
  (progn
    (push '|Polynomial| *Domains*)
    (make-instance '|PolynomialType|)))
```

———–

1.67.26 PolynomialIdeals

— defclass PolynomialIdealsType —

```
(defclass |PolynomialIdealsType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "PolynomialIdeals")
   (marker :initform 'domain)
   (abbreviation :initform 'IDEAL)
   (comment :initform (list
     "This domain represents polynomial ideals with coefficients in any"
     "field and supports the basic ideal operations, including intersection"
     "sum and quotient."
     "An ideal is represented by a list of polynomials (the generators of"
     "the ideal) and a boolean that is true if the generators are a Groebner"
     "basis."
     "The algorithms used are based on Groebner basis computations. The"
     "ordering is determined by the datatype of the input polynomials."
     "Users may use refinements of total degree orderings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialIdeals|
  (progn
    (push '|PolynomialIdeals| *Domains*)
    (make-instance '|PolynomialIdealsType|)))
```

———-

1.67.27 PolynomialRing

— defclass PolynomialRingType —

```
(defclass |PolynomialRingType| (|FiniteAbelianMonoidRingType|)
  ((parents :initform '(|FiniteAbelianMonoidRing|))
   (name :initform "PolynomialRing")
   (marker :initform 'domain)
   (abbreviation :initform 'PR)
   (comment :initform (list
     "This domain represents generalized polynomials with coefficients"
     "(from a not necessarily commutative ring), and terms"
     "indexed by their exponents (from an arbitrary ordered abelian monoid)."
     "This type is used, for example,"
     "by the DistributedMultivariatePolynomial domain where"
     "the exponent domain is a direct product of non negative integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialRing|
  (progn
    (push '|PolynomialRing| *Domains*)
    (make-instance '|PolynomialRingType|)))
```

———-

1.67.28 PositiveInteger

— defclass PositiveIntegerType —

```
(defclass |PositiveIntegerType| (|OrderedSetType| |MonoidType| |AbelianSemiGroupType|)
  ((parents :initform '(|OrderedSet| |Monoid| |AbelianSemiGroup|))
   (name :initform "PositiveInteger")
   (marker :initform 'domain)
   (abbreviation :initform 'PI)
   (comment :initform (list
     "PositiveInteger provides functions for positive integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PositiveInteger|
  (progn
    (push '|PositiveInteger| *Domains*)
    (make-instance '|PositiveIntegerType|)))
```

———-

1.67.29 PrimeField

— defclass PrimeFieldType —

```
(defclass |PrimeFieldType| (|ConvertibleToType|
                            |FiniteAlgebraicExtensionFieldType|)
  ((parents :initform '(|ConvertibleTo|
                        |FiniteAlgebraicExtensionField|))
   (name :initform "PrimeField")
   (marker :initform 'domain)
   (abbreviation :initform 'PF)
   (comment :initform (list
     "PrimeField(p) implements the field with p elements if p is a prime number."
     "Error: if p is not prime."
     "Note: this domain does not check that argument is a prime."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimeField|
  (progn
    (push '|PrimeField| *Domains*)
    (make-instance '|PrimeFieldType|)))
```

———-

1.67.30 PrimitiveArray

— defclass PrimitiveArrayType —

```
(defclass |PrimitiveArrayType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "PrimitiveArray")
   (marker :initform 'domain)
   (abbreviation :initform 'PRIMARR)
   (comment :initform (list
     "This provides a fast array type with no bound checking on elt's."
     "Minimum index is 0 in this type, cannot be changed"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveArray|
  (progn
    (push '|PrimitiveArray| *Domains*)
    (make-instance '|PrimitiveArrayType|)))
```

——–-

1.67.31 Product

— defclass ProductType —

```
(defclass |ProductType| (|AbelianGroupType|
                         |FiniteType|
                         |GroupType|
                         |OrderedAbelianMonoidSupType|)
  ((parents :initform '(|AbelianGroup| |Finite| |Group| |OrderedAbelianMonoidSup|))
   (name :initform "Product")
   (marker :initform 'domain)
   (abbreviation :initform 'PRODUCT)
   (comment :initform (list
     "This domain implements cartesian product"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Product|
  (progn
    (push '|Product| *Domains*)
    (make-instance '|ProductType|)))
```

——–-

1.67.32 ProjectivePlane

— defclass ProjectivePlaneType —

```
(defclass |ProjectivePlaneType| (|ProjectiveSpaceCategoryType|)
  ((parents :initform '(|ProjectiveSpaceCategory|))
   (name :initform "ProjectivePlane")
   (marker :initform 'domain)
   (abbreviation :initform 'PROJPL)
   (comment :initform (list
     "This is part of the PAFF package, related to projective space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectivePlane|
  (progn
    (push '|ProjectivePlane| *Domains*)
    (make-instance '|ProjectivePlaneType|)))
```

———-

1.67.33 ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField

— defclass ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteFieldType —

```
(defclass |ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteFieldType| (|ProjectiveSpaceCategoryType|)
  ((parents :initform '(|ProjectiveSpaceCategory|))
   (name :initform "ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField")
   (marker :initform 'domain)
   (abbreviation :initform 'PROJPLPS)
   (comment :initform (list
     "This is part of the PAFF package, related to projective space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.67.34 ProjectiveSpace

— defclass ProjectiveSpaceType —

```
(defclass |ProjectiveSpaceType| (|ProjectiveSpaceCategoryType|)
  ((parents :initform '(|ProjectiveSpaceCategory|))
   (name :initform "ProjectiveSpace")
   (marker :initform 'domain)
   (abbreviation :initform 'PROJSP)
   (comment :initform (list
     "This is part of the PAFF package, related to projective space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectiveSpace|
  (progn
    (push '|ProjectiveSpace| *Domains*)
    (make-instance '|ProjectiveSpaceType|)))
```

———-

1.67.35 PseudoAlgebraicClosureOfAlgExtOfRationalNumber

— defclass PseudoAlgebraicClosureOfAlgExtOfRationalNumberType —

```
(defclass |PseudoAlgebraicClosureOfAlgExtOfRationalNumberType| (
                    |PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategoryType|)
  ((parents :initform '(|PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory|))
   (name :initform "PseudoAlgebraicClosureOfAlgExtOfRationalNumber")
   (marker :initform 'domain)
   (abbreviation :initform 'PACEXT)
   (comment :initform (list
     "This domain implement dynamic extension over the"
     "PseudoAlgebraicClosureOfRationalNumber."
     "A tower extension T  of the ground field K is any sequence of field"
     "extension (T : K_0, K_1, ..., K_i...,K_n) where K_0 = K"
     "and for i =1,2,...,n, K_i is an extension of K_{i-1} of degree > 1"
     "and defined by an irreducible polynomial p(Z) in K_{i-1}."
     "Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1) and "
     "(T_2: K_02, K_12,...,K_i2,...,K_n2)"
     "are said to be related if T_1 <= T_2 (or T_1 >= T_2),"
     "that is if K_i1 = K_i2 for i=1,2,...,n1"
     "(or i=1,2,...,n2). Any algebraic operations defined for several elements"
     "are only defined if all of the concerned elements are comming from"
     "a set of related tour extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfAlgExtOfRationalNumber|
  (progn
    (push '|PseudoAlgebraicClosureOfAlgExtOfRationalNumber| *Domains*)
    (make-instance '|PseudoAlgebraicClosureOfAlgExtOfRationalNumberType|)))
```

———-

1.67.36 PseudoAlgebraicClosureOfFiniteField

— defclass PseudoAlgebraicClosureOfFiniteFieldType —

```
(defclass |PseudoAlgebraicClosureOfFiniteFieldType| (|PseudoAlgebraicClosureOfFiniteFieldCategoryType|
                                                     |ExtensionFieldType|)
  ((parents :initform '(|PseudoAlgebraicClosureOfFiniteFieldCategory| |ExtensionField|))
   (name :initform "PseudoAlgebraicClosureOfFiniteField")
   (marker :initform 'domain)
   (abbreviation :initform 'PACOFF)
   (comment :initform (list
     "This domain implement  dynamic extension using the simple notion of"
     "tower extensions. A tower extension T  of the ground  field K is any"
     "sequence of field extension (T : K_0, K_1, ..., K_i...,K_n) where K_0 = K"
     "and for i =1,2,...,n, K_i is an extension of K_{i-1} of degree > 1 and"
     "defined by an irreducible polynomial p(Z) in K_{i-1}."
     "Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1)"
     "and (T_2: K_02, K_12,...,K_i2,...,K_n2) are said to be related"
     "if T_1 <= T_2 (or T_1 >= T_2), that is if K_i1 = K_i2 for i=1,2,...,n1"
     "(or i=1,2,...,n2). Any algebraic operations defined for several elements"
     "are only defined if all of the concerned elements are comming from"
     "a set of related tour extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|PseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|PseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.67.37 PseudoAlgebraicClosureOfRationalNumber

— defclass PseudoAlgebraicClosureOfRationalNumberType —

```
(defclass |PseudoAlgebraicClosureOfRationalNumberType| (
                        |PseudoAlgebraicClosureOfRationalNumberCategoryType|)
  ((parents :initform '(|PseudoAlgebraicClosureOfRationalNumberCategory|))
   (name :initform "PseudoAlgebraicClosureOfRationalNumber")
   (marker :initform 'domain)
   (abbreviation :initform 'PACRAT)
   (comment :initform (list
     "++ This domain implements dynamic extension using the simple notion of"
     "tower extensions. A tower extension T of the ground field K is any"
     "sequence of field extension (T : K_0, K_1, ..., K_i...,K_n) where K_0 = K"
     "and for i =1,2,...,n, K_i is an extension of K_{i-1} of degree > 1 and"
     "defined by an irreducible polynomial p(Z) in K_{i-1}."
     "Two towers (T_1: K_01, K_11,...,K_i1,...,K_n1) and"
     "(T_2: K_02, K_12,...,K_i2,...,K_n2) are said to be related if T_1 <= T_2"
     "(or T_1 >= T_2), that is if K_i1 = K_i2 for i=1,2,...,n1"
     "(or i=1,2,...,n2). Any algebraic operations defined for several elements"
     "are only defined if all of the concerned elements are comming from"
     "a set of related tour extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfRationalNumber|
  (progn
    (push '|PseudoAlgebraicClosureOfRationalNumber| *Domains*)
    (make-instance '|PseudoAlgebraicClosureOfRationalNumberType|)))
```

——————-

1.68 Q

1.68.1 QuadraticForm

— defclass QuadraticFormType —

```
(defclass |QuadraticFormType| (|AbelianGroupType|)
  ((parents :initform '(|AbelianGroup|))
   (name :initform "QuadraticForm")
   (marker :initform 'domain)
   (abbreviation :initform 'QFORM)
   (comment :initform (list
     "This domain provides modest support for quadratic forms."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuadraticForm|
  (progn
    (push '|QuadraticForm| *Domains*)
    (make-instance '|QuadraticFormType|)))
```

——————-

1.68.2 QuasiAlgebraicSet

— defclass QuasiAlgebraicSetType —

```
(defclass |QuasiAlgebraicSetType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "QuasiAlgebraicSet")
   (marker :initform 'domain)
   (abbreviation :initform 'QALGSET)
   (comment :initform (list
     "QuasiAlgebraicSet constructs a domain representing"
     "quasi-algebraic sets, which is the intersection of a Zariski"
     "closed set, defined as the common zeros of a given list of"
     "polynomials (the defining polynomials for equations), and a principal"
     "Zariski open set, defined as the complement of the common"
     "zeros of a polynomial f (the defining polynomial for the inequation)."
     "This domain provides simplification of a user-given representation"
     "using groebner basis computations."
     "There are two simplification routines: the first function"
     "idealSimplify  uses groebner"
     "basis of ideals alone, while the second, simplify uses both"
     "groebner basis and factorization.  The resulting defining equations L"
     "always form a groebner basis, and the resulting defining"
     "inequation f is always reduced.  The function simplify may"
     "be applied several times if desired.   A third simplification"
     "routine radicalSimplify is provided in"
     "QuasiAlgebraicSet2  for comparison study only,"
     "as it is inefficient compared to the other two, as well as is"
     "restricted to only certain coefficient domains.  For detail analysis"
     "and a comparison of the three methods, please consult the reference"
     "cited."
     " "
     "A polynomial function q defined on the quasi-algebraic set"
     "is equivalent to its reduced form with respect to L.  While"
     "this may be obtained using the usual normal form"
     "algorithm, there is no canonical form for q."
     " "
     "The ordering in groebner basis computation is determined by"
     "the data type of the input polynomials.  If it is possible"
     "we suggest to use refinements of total degree orderings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuasiAlgebraicSet|
  (progn
    (push '|QuasiAlgebraicSet| *Domains*)
    (make-instance '|QuasiAlgebraicSetType|)))
```

———-

1.68.3 Quaternion

— defclass QuaternionType —

```
(defclass |QuaternionType| (|QuaternionCategoryType|)
  ((parents :initform '(|QuaternionCategory|))
   (name :initform "Quaternion")
   (marker :initform 'domain)
   (abbreviation :initform 'QUAT)
   (comment :initform (list
     "Quaternion implements quaternions over a"
     "commutative ring. The main constructor function is quatern"
     "which takes 4 arguments: the real part, the i imaginary part, the j"
     "imaginary part and the k imaginary part."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Quaternion|
  (progn
    (push '|Quaternion| *Domains*)
    (make-instance '|QuaternionType|)))
```

————-

1.68.4 QueryEquation

— defclass QueryEquationType —

```
(defclass |QueryEquationType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "QueryEquation")
   (marker :initform 'domain)
   (abbreviation :initform 'QEQUAT)
   (comment :initform (list
     "This domain implements simple database queries"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QueryEquation|
  (progn
    (push '|QueryEquation| *Domains*)
    (make-instance '|QueryEquationType|)))
```

————-

1.68.5 Queue

— defclass QueueType —

```
(defclass |QueueType| (|QueueAggregateType|)
  ((parents :initform '(|QueueAggregate|))
   (name :initform "Queue")
   (marker :initform 'domain)
   (abbreviation :initform 'QUEUE)
   (comment :initform (list
     "Linked List implementation of a Queue"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Queue|
  (progn
    (push '|Queue| *Domains*)
    (make-instance '|QueueType|)))
```

———-

1.69 R

1.69.1 RadicalFunctionField

— defclass RadicalFunctionFieldType —

```
(defclass |RadicalFunctionFieldType| (|FunctionFieldCategoryType|)
  ((parents :initform '(|FunctionFieldCategory|))
   (name :initform "RadicalFunctionField")
   (marker :initform 'domain)
   (abbreviation :initform 'RADFF)
   (comment :initform (list
     "Function field defined by y**n = f(x)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadicalFunctionField|
  (progn
    (push '|RadicalFunctionField| *Domains*)
    (make-instance '|RadicalFunctionFieldType|)))
```

———-

1.69.2 RadixExpansion

— defclass RadixExpansionType —

```
(defclass |RadixExpansionType| (|QuotientFieldCategoryType|)
  ((parents :initform '(|QuotientFieldCategory|))
   (name :initform "RadixExpansion")
   (marker :initform 'domain)
   (abbreviation :initform 'RADIX)
   (comment :initform (list
     "This domain allows rational numbers to be presented as repeating"
     "decimal expansions or more generally as repeating expansions in any base."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadixExpansion|
  (progn
    (push '|RadixExpansion| *Domains*)
    (make-instance '|RadixExpansionType|)))
```

———-

1.69.3 RealClosure

— defclass RealClosureType —

```
(defclass |RealClosureType| (|RealClosedFieldType|)
  ((parents :initform '(|RealClosedField|))
   (name :initform "RealClosure")
   (marker :initform 'domain)
   (abbreviation :initform 'RECLOS)
   (comment :initform (list
     "This domain implements the real closure of an ordered field."
     "Note:"
     "The code here is generic it does not depend of the way the operations"
     "are done. The two macros PME and SEG should be passed as functorial"
     "arguments to the domain. It does not help much to write a category"
     "since non trivial methods cannot be placed there either."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealClosure|
  (progn
    (push '|RealClosure| *Domains*)
    (make-instance '|RealClosureType|)))
```

———-

1.69.4 RectangularMatrix

— defclass RectangularMatrixType —

```
(defclass |RectangularMatrixType| (|ConvertibleToType|
                                   |RectangularMatrixCategoryType|
                                   |VectorSpaceType|)
  ((parents :initform '(|ConvertibleTo|
                        |RectangularMatrixCategory|
                        |VectorSpace|))
   (name :initform "RectangularMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'RMATRIX)
   (comment :initform (list
     "RectangularMatrix is a matrix domain where the number of rows"
     "and the number of columns are parameters of the domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RectangularMatrix|
  (progn
    (push '|RectangularMatrix| *Domains*)
    (make-instance '|RectangularMatrixType|)))
```

———-

1.69.5 Reference

— defclass ReferenceType —

```
(defclass |ReferenceType| (|TypeType| |SetCategoryType|)
  ((parents :initform '(|Type| |SetCategory|))
   (name :initform "Reference")
   (marker :initform 'domain)
   (abbreviation :initform 'REF)
   (comment :initform (list
     "Reference is for making a changeable instance of something."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Reference|
  (progn
    (push '|Reference| *Domains*)
    (make-instance '|ReferenceType|)))
```

———-

1.69.6 RegularChain

— defclass RegularChainType —

```
(defclass |RegularChainType| (|RegularTriangularSetCategoryType|)
  ((parents :initform '(|RegularTriangularSetCategory|))
   (name :initform "RegularChain")
   (marker :initform 'domain)
   (abbreviation :initform 'RGCHAIN)
   (comment :initform (list
     "A domain for regular chains (regular triangular sets) over"
     "a Gcd-Domain and with a fix list of variables."
     "This is just a front-end for the RegularTriangularSet"
     "domain constructor."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularChain|
  (progn
    (push '|RegularChain| *Domains*)
    (make-instance '|RegularChainType|)))
```

———-

1.69.7 RegularTriangularSet

— defclass RegularTriangularSetType —

```
(defclass |RegularTriangularSetType| (|RegularTriangularSetCategoryType|)
  ((parents :initform '(|RegularTriangularSetCategory|))
   (name :initform "RegularTriangularSet")
   (marker :initform 'domain)
   (abbreviation :initform 'REGSET)
   (comment :initform (list
     "This domain provides an implementation of regular chains."
     "Moreover, the operation zeroSetSplit is an implementation of a new"
     "algorithm for solving polynomial systems by means of regular chains."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularTriangularSet|
  (progn
    (push '|RegularTriangularSet| *Domains*)
    (make-instance '|RegularTriangularSetType|)))
```

———-

1.69.8 ResidueRing

— defclass ResidueRingType —

```
(defclass |ResidueRingType| (|CommutativeRingType| |AlgebraType|)
  ((parents :initform '(|CommutativeRing| |Algebra|))
   (name :initform "ResidueRing")
   (marker :initform 'domain)
   (abbreviation :initform 'RESRING)
   (comment :initform (list
     "ResidueRing is the quotient of a polynomial ring by  an ideal."
     "The ideal is given as a list of generators. The elements of the domain"
     "are equivalence classes expressed in terms of reduced elements"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ResidueRing|
  (progn
    (push '|ResidueRing| *Domains*)
    (make-instance '|ResidueRingType|)))
```

———-

1.69.9 Result

— defclass ResultType —

```
(defclass |ResultType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "Result")
   (marker :initform 'domain)
   (abbreviation :initform 'RESULT)
   (comment :initform (list
     "A domain used to return the results from a call to the NAG"
     "Library.  It prints as a list of names and types, though the user may"
     "choose to display values automatically if he or she wishes."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Result|
  (progn
    (push '|Result| *Domains*)
    (make-instance '|ResultType|)))
```

———-

1.69.10 RewriteRule

— defclass RewriteRuleType —

```
(defclass |RewriteRuleType| (|SetCategoryType| |RetractableToType| |EltableType|)
  ((parents :initform '(|SetCategory| |RetractableTo| |Eltable|))
   (name :initform "RewriteRule")
   (marker :initform 'domain)
   (abbreviation :initform 'RULE)
   (comment :initform (list
     "Rules for the pattern matcher"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |RewriteRule|
  (progn
    (push '|RewriteRule| *Domains*)
    (make-instance '|RewriteRuleType|)))
```

———-

1.69.11 RightOpenIntervalRootCharacterization

— defclass RightOpenIntervalRootCharacterizationType —

```
(defclass |RightOpenIntervalRootCharacterizationType| (|RealRootCharacterizationCategoryType|)
  ((parents :initform '(|RealRootCharacterizationCategory|))
   (name :initform "RightOpenIntervalRootCharacterization")
   (marker :initform 'domain)
   (abbreviation :initform 'ROIRC)
   (comment :initform (list
     "RightOpenIntervalRootCharacterization provides work with"
     "interval root coding."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RightOpenIntervalRootCharacterization|
  (progn
    (push '|RightOpenIntervalRootCharacterization| *Domains*)
    (make-instance '|RightOpenIntervalRootCharacterizationType|)))
```

———-

1.69.12 RomanNumeral

— defclass RomanNumeralType —

```
(defclass |RomanNumeralType| (|IntegerNumberSystemType|)
  ((parents :initform '(|IntegerNumberSystem|))
   (name :initform "RomanNumeral")
   (marker :initform 'domain)
   (abbreviation :initform 'ROMAN)
   (comment :initform (list
     "RomanNumeral provides functions for converting"
     "integers to roman numerals."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RomanNumeral|
  (progn
    (push '|RomanNumeral| *Domains*)
    (make-instance '|RomanNumeralType|)))
```

———-

1.69.13 RoutinesTable

— defclass RoutinesTableType —

```
(defclass |RoutinesTableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "RoutinesTable")
   (marker :initform 'domain)
   (abbreviation :initform 'ROUTINE)
   (comment :initform (list
     "RoutinesTable implements a database and associated tuning"
     "mechanisms for a set of known NAG routines"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RoutinesTable|
  (progn
    (push '|RoutinesTable| *Domains*)
    (make-instance '|RoutinesTableType|)))
```

———-

1.69.14 RuleCalled

— defclass RuleCalledType —

```
(defclass |RuleCalledType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "RuleCalled")
   (marker :initform 'domain)
   (abbreviation :initform 'RULECOLD)
   (comment :initform (list
     "This domain implements named rules "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RuleCalled|
  (progn
    (push '|RuleCalled| *Domains*)
    (make-instance '|RuleCalledType|)))
```

———-

1.69.15 Ruleset

— defclass RulesetType —

```
(defclass |RulesetType| (|SetCategoryType| |EltableType|)
  ((parents :initform '(|SetCategory| |Eltable|))
   (name :initform "Ruleset")
   (marker :initform 'domain)
   (abbreviation :initform 'RULESET)
   (comment :initform (list
     "Sets of rules for the pattern matcher."
     "A ruleset is a set of pattern matching rules grouped together."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Ruleset|
  (progn
    (push '|Ruleset| *Domains*)
    (make-instance '|RulesetType|)))
```

———-

1.70 S

1.70.1 ScriptFormulaFormat

— defclass ScriptFormulaFormatType —

```
(defclass |ScriptFormulaFormatType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "ScriptFormulaFormat")
   (marker :initform 'domain)
   (abbreviation :initform 'FORMULA)
   (comment :initform (list
     "ScriptFormulaFormat provides a coercion from"
     "OutputForm to IBM SCRIPT/VS Mathematical Formula Format."
     "The basic SCRIPT formula format object consists of three parts:"
     "a prologue, a formula part and an epilogue.  The functions"
     "prologue, formula and epilogue"
     "extract these parts, respectively.  The central parts of the expression"
     "go into the formula part.  The other parts can be set"
     "(setPrologue!, setEpilogue!) so that contain the"
     "appropriate tags for printing.  For example, the prologue and"
     "epilogue might simply contain ':df.'  and ':edf.'  so that the"
     "formula section will be printed in display math mode."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ScriptFormulaFormat|
  (progn
    (push '|ScriptFormulaFormat| *Domains*)
    (make-instance '|ScriptFormulaFormatType|)))
```

———-

1.70.2 Segment

— defclass SegmentType —

```
(defclass |SegmentType| (|SetCategoryType| |SegmentExpansionCategoryType|)
  ((parents :initform '(|SetCategory| |SegmentExpansionCategory|))
   (name :initform "Segment")
   (marker :initform 'domain)
   (abbreviation :initform 'SEG)
   (comment :initform (list
     "This type is used to specify a range of values from type S."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Segment|
  (progn
    (push '|Segment| *Domains*)
    (make-instance '|SegmentType|)))
```

———-

1.70.3 SegmentBinding

— defclass SegmentBindingType —

```
(defclass |SegmentBindingType| (|TypeType| |SetCategoryType|)
  ((parents :initform '(|Type| |SetCategory|))
   (name :initform "SegmentBinding")
   (marker :initform 'domain)
   (abbreviation :initform 'SEGBIND)
   (comment :initform (list
     "This domain is used to provide the function argument syntax v=a..b."
     "This is used, for example, by the top-level draw functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |SegmentBinding|
  (progn
    (push '|SegmentBinding| *Domains*)
    (make-instance '|SegmentBindingType|)))
```

———-

1.70.4 Set

— defclass SetType —

```
(defclass |SetType| (|FiniteSetAggregateType|)
  ((parents :initform '(|FiniteSetAggregate|))
   (name :initform "Set")
   (marker :initform 'domain)
   (abbreviation :initform 'SET)
   (comment :initform (list
     "A set over a domain D models the usual mathematical notion of a finite set"
     "of elements from D."
     "Sets are unordered collections of distinct elements"
     "(that is, order and duplication does not matter)."
     "The notation set [a,b,c] can be used to create"
     "a set and the usual operations such as union and intersection are available"
     "to form new sets."
     "In our implementation, Axiom maintains the entries in"
     "sorted order.  Specifically, the parts function returns the entries"
     "as a list in ascending order and"
     "the extract operation returns the maximum entry."
     "Given two sets s and t where #s = m and #t = n,"
     "the complexity of"
     "     s = t is O(min(n,m)"
     "     s < t is O(max(n,m)"
     "     union(s,t), intersect(s,t), minus(s,t)"
     "        symmetricDifference(s,t) is O(max(n,m)"
     "     member(x,t) is O(n log n)"
     "     insert(x,t) and remove(x,t) is O(n)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Set|
  (progn
    (push '|Set| *Domains*)
    (make-instance '|SetType|)))
```

———-

1.70.5 SetOfMIntegersInOneToN

— defclass SetOfMIntegersInOneToNType —

```
(defclass |SetOfMIntegersInOneToNType| (|FiniteType|)
  ((parents :initform '(|Finite|))
   (name :initform "SetOfMIntegersInOneToN")
   (marker :initform 'domain)
   (abbreviation :initform 'SETMN)
   (comment :initform (list
     "SetOfMIntegersInOneToN implements the subsets of M integers"
     "in the interval [1..n]"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SetOfMIntegersInOneToN|
  (progn
    (push '|SetOfMIntegersInOneToN| *Domains*)
    (make-instance '|SetOfMIntegersInOneToNType|)))
```

———-

1.70.6 SequentialDifferentialPolynomial

— defclass SequentialDifferentialPolynomialType —

```
(defclass |SequentialDifferentialPolynomialType| (|DifferentialPolynomialCategoryType|)
  ((parents :initform '(|DifferentialPolynomialCategory|))
   (name :initform "SequentialDifferentialPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'SDPOL)
   (comment :initform (list
     "SequentialDifferentialPolynomial implements"
     "an ordinary differential polynomial ring in arbitrary number"
     "of differential indeterminates, with coefficients in a"
     "ring.  The ranking on the differential indeterminate is sequential."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SequentialDifferentialPolynomial|
  (progn
    (push '|SequentialDifferentialPolynomial| *Domains*)
    (make-instance '|SequentialDifferentialPolynomialType|)))
```

———-

1.70.7 SequentialDifferentialVariable

— defclass SequentialDifferentialVariableType —

```
(defclass |SequentialDifferentialVariableType| (|DifferentialVariableCategoryType|)
  ((parents :initform '(|DifferentialVariableCategory|))
   (name :initform "SequentialDifferentialVariable")
   (marker :initform 'domain)
   (abbreviation :initform 'SDVAR)
   (comment :initform (list
     "OrderlyDifferentialVariable adds a commonly used sequential"
     "ranking to the set of derivatives of an ordered list of differential"
     "indeterminates.  A sequential ranking is a ranking < of the"
     "derivatives with the property that for any derivative v,"
     "there are only a finite number of derivatives u with u < v."
     "This domain belongs to DifferentialVariableCategory. It"
     "defines weight to be just order, and it"
     "defines a sequential ranking < on derivatives u by the"
     "lexicographic order on the pair"
     "(variable(u), order(u))."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SequentialDifferentialVariable|
  (progn
    (push '|SequentialDifferentialVariable| *Domains*)
    (make-instance '|SequentialDifferentialVariableType|)))
```

———-

1.70.8 SExpression

— defclass SExpressionType —

```
(defclass |SExpressionType| (|SExpressionCategoryType|)
  ((parents :initform '(|SExpressionCategory|))
   (name :initform "SExpression")
   (marker :initform 'domain)
   (abbreviation :initform 'SEX)
   (comment :initform (list
     "This domain allows the manipulation of the usual Lisp values"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SExpression|
  (progn
    (push '|SExpression| *Domains*)
    (make-instance '|SExpressionType|)))
```

———-

1.70.9 SExpressionOf

— defclass SExpressionOfType —

```
(defclass |SExpressionOfType| (|SExpressionCategoryType|)
  ((parents :initform '(|SExpressionCategory|))
   (name :initform "SExpressionOf")
   (marker :initform 'domain)
   (abbreviation :initform 'SEXOF)
   (comment :initform (list
     "This domain allows the manipulation of Lisp values over"
     "arbitrary atomic types."
     "Allows the names of the atomic types to be chosen."
     "Warning: Although the parameters are declared only to be Sets,"
     "they must have the appropriate representations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SExpressionOf|
  (progn
    (push '|SExpressionOf| *Domains*)
    (make-instance '|SExpressionOfType|)))
```

———–

1.70.10 SimpleAlgebraicExtension

— defclass SimpleAlgebraicExtensionType —

```
(defclass |SimpleAlgebraicExtensionType| (|MonogenicAlgebraType|)
  ((parents :initform '(|MonogenicAlgebra|))
   (name :initform "SimpleAlgebraicExtension")
   (marker :initform 'domain)
   (abbreviation :initform 'SAE)
   (comment :initform (list
     "Algebraic extension of a ring by a single polynomial."
     "Domain which represents simple algebraic extensions of arbitrary"
     "rings. The first argument to the domain, R, is the underlying ring,"
     "the second argument is a domain of univariate polynomials over K,"
     "while the last argument specifies the defining minimal polynomial."
     "The elements of the domain are canonically represented as polynomials"
     "of degree less than that of the minimal polynomial with coefficients"
     "in R. The second argument is both the type of the third argument and"
     "the underlying representation used by SAE itself."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleAlgebraicExtension|
  (progn
    (push '|SimpleAlgebraicExtension| *Domains*)
    (make-instance '|SimpleAlgebraicExtensionType|)))
```

———-

1.70.11 SimpleCell

— defclass SimpleCellType —

```
(defclass |SimpleCellType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "SimpleCell")
   (marker :initform 'domain)
   (abbreviation :initform 'SCELL)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleCell|
  (progn
    (push '|SimpleCell| *Domains*)
    (make-instance '|SimpleCellType|)))
```

———-

1.70.12 SimpleFortranProgram

— defclass SimpleFortranProgramType —

```
(defclass |SimpleFortranProgramType| (|FortranProgramCategoryType|)
  ((parents :initform '(|FortranProgramCategory|))
   (name :initform "SimpleFortranProgram")
   (marker :initform 'domain)
   (abbreviation :initform 'SFORT)
   (comment :initform (list
     "SimpleFortranProgram(f,type) provides a simple model of some"
     "FORTRAN subprograms, making it possible to coerce objects of various"
     "domains into a FORTRAN subprogram called f."
     "These can then be translated into legal FORTRAN code."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleFortranProgram|
  (progn
    (push '|SimpleFortranProgram| *Domains*)
    (make-instance '|SimpleFortranProgramType|)))
```

———-

1.70.13 SingleInteger

— defclass SingleIntegerType —

```
(defclass |SingleIntegerType| (|OpenMathType| |LogicType| |IntegerNumberSystemType|)
  ((parents :initform '(|OpenMath| |Logic| |IntegerNumberSystem|))
   (name :initform "SingleInteger")
   (marker :initform 'domain)
   (abbreviation :initform 'SINT)
   (comment :initform (list
     "SingleInteger is intended to support machine integer arithmetic."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SingleInteger|
  (progn
    (push '|SingleInteger| *Domains*)
    (make-instance '|SingleIntegerType|)))
```

———-

1.70.14 SingletonAsOrderedSet

— defclass SingletonAsOrderedSetType —

```
(defclass |SingletonAsOrderedSetType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "SingletonAsOrderedSet")
   (marker :initform 'domain)
   (abbreviation :initform 'SAOS)
   (comment :initform (list
     "This trivial domain lets us build Univariate Polynomials"
     "in an anonymous variable"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SingletonAsOrderedSet|
  (progn
    (push '|SingletonAsOrderedSet| *Domains*)
    (make-instance '|SingletonAsOrderedSetType|)))
```

———-

1.70.15 SparseEchelonMatrix

— defclass SparseEchelonMatrixType —

```
(defclass |SparseEchelonMatrixType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "SparseEchelonMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'SEM)
   (comment :initform (list
     "SparseEchelonMatrix(C, D) implements sparse matrices whose columns"
     "are enumerated by the OrderedSet C and whose entries"
     "belong to the GcdDomain D. The basic operation of"
     "this domain is the computation of an row echelon form. The used algorithm"
     "tries to maintain the sparsity and is especially adapted to matrices who"
     "are already close to a row echelon form."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseEchelonMatrix|
  (progn
    (push '|SparseEchelonMatrix| *Domains*)
    (make-instance '|SparseEchelonMatrixType|)))
```

———–

1.70.16 SparseMultivariatePolynomial

— defclass SparseMultivariatePolynomialType —

```
(defclass |SparseMultivariatePolynomialType| (|PolynomialCategoryType|)
  ((parents :initform '(|PolynomialCategory|))
   (name :initform "SparseMultivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'SMP)
   (comment :initform (list
     "This type is the basic representation of sparse recursive multivariate"
     "polynomials. It is parameterized by the coefficient ring and the"
     "variable set which may be infinite. The variable ordering is determined"
     "by the variable set parameter. The coefficient ring may be non-commutative,"
     "but the variables are assumed to commute."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseMultivariatePolynomial|
  (progn
    (push '|SparseMultivariatePolynomial| *Domains*)
    (make-instance '|SparseMultivariatePolynomialType|)))
```

———-

1.70.17 SparseMultivariateTaylorSeries

— defclass SparseMultivariateTaylorSeriesType —

```
(defclass |SparseMultivariateTaylorSeriesType| (|MultivariateTaylorSeriesCategoryType|)
  ((parents :initform '(|MultivariateTaylorSeriesCategory|))
   (name :initform "SparseMultivariateTaylorSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'SMTS)
   (comment :initform (list
     "This domain provides multivariate Taylor series with variables"
     "from an arbitrary ordered set.  A Taylor series is represented"
     "by a stream of polynomials from the polynomial domain SMP."
     "The nth element of the stream is a form of degree n.  SMTS is an"
     "internal domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseMultivariateTaylorSeries|
  (progn
    (push '|SparseMultivariateTaylorSeries| *Domains*)
    (make-instance '|SparseMultivariateTaylorSeriesType|)))
```

———-

1.70.18 SparseTable

— defclass SparseTableType —

```
(defclass |SparseTableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "SparseTable")
   (marker :initform 'domain)
   (abbreviation :initform 'STBL)
   (comment :initform (list
     "A sparse table has a default entry, which is returned if no other"
     "value has been explicitly stored for a key."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseTable|
  (progn
    (push '|SparseTable| *Domains*)
    (make-instance '|SparseTableType|)))
```

———-

1.70.19 SparseUnivariateLaurentSeries

— defclass SparseUnivariateLaurentSeriesType —

```
(defclass |SparseUnivariateLaurentSeriesType| (|UnivariateLaurentSeriesConstructorCategoryType|)
  ((parents :initform '(|UnivariateLaurentSeriesConstructorCategory|))
   (name :initform "SparseUnivariateLaurentSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'SULS)
   (comment :initform (list
     "SparseUnivariateLaurentSeries is a domain representing Laurent"
     "series in one variable with coefficients in an arbitrary ring.  The"
     "parameters of the type specify the coefficient ring, the power series"
     "variable, and the center of the power series expansion.  For example,"
     "SparseUnivariateLaurentSeries(Integer,x,3) represents Laurent"
     "series in (x - 3) with integer coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariateLaurentSeries|
  (progn
    (push '|SparseUnivariateLaurentSeries| *Domains*)
    (make-instance '|SparseUnivariateLaurentSeriesType|)))
```

———-

1.70.20 SparseUnivariatePolynomial

— defclass SparseUnivariatePolynomialType —

```
(defclass |SparseUnivariatePolynomialType| (|UnivariatePolynomialCategoryType|)
  ((parents :initform '(|UnivariatePolynomialCategory|))
   (name :initform "SparseUnivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'SUP)
   (comment :initform (list
     "This domain represents univariate polynomials over arbitrary"
     "(not necessarily commutative) coefficient rings. The variable is"
     "unspecified  so that the variable displays as ? on output."
     "If it is necessary to specify the variable name,"
     "use type UnivariatePolynomial. The representation is sparse"
     "in the sense that only non-zero terms are represented."
     "Note that if the coefficient ring is a field,"
     "this domain forms a euclidean domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariatePolynomial|
  (progn
    (push '|SparseUnivariatePolynomial| *Domains*)
    (make-instance '|SparseUnivariatePolynomialType|)))
```

———-

1.70.21 SparseUnivariatePolynomialExpressions

— defclass SparseUnivariatePolynomialExpressionsType —

```
(defclass |SparseUnivariatePolynomialExpressionsType| (|UnivariatePolynomialCategoryType|
                                                        |TranscendentalFunctionCategoryType|)
  ((parents :initform '(|UnivariatePolynomialCategory| |TranscendentalFunctionCategory|))
   (name :initform "SparseUnivariatePolynomialExpressions")
   (marker :initform 'domain)
   (abbreviation :initform 'SUPEXPR)
   (comment :initform (list
     "This domain has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariatePolynomialExpressions|
  (progn
    (push '|SparseUnivariatePolynomialExpressions| *Domains*)
    (make-instance '|SparseUnivariatePolynomialExpressionsType|)))
```

———-

1.70.22 SparseUnivariatePuiseuxSeries

— defclass SparseUnivariatePuiseuxSeriesType —

```
(defclass |SparseUnivariatePuiseuxSeriesType| (|UnivariatePuiseuxSeriesConstructorCategoryType|)
  ((parents :initform '(|UnivariatePuiseuxSeriesConstructorCategory|))
   (name :initform "SparseUnivariatePuiseuxSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'SUPXS)
   (comment :initform (list
     "Sparse Puiseux series in one variable"
     "SparseUnivariatePuiseuxSeries is a domain representing Puiseux"
     "series in one variable with coefficients in an arbitrary ring.  The"
     "parameters of the type specify the coefficient ring, the power series"
     "variable, and the center of the power series expansion.  For example,"
     "SparseUnivariatePuiseuxSeries(Integer,x,3) represents Puiseux"
     "series in (x - 3) with Integer coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariatePuiseuxSeries|
  (progn
    (push '|SparseUnivariatePuiseuxSeries| *Domains*)
    (make-instance '|SparseUnivariatePuiseuxSeriesType|)))
```

———-

1.70.23 SparseUnivariateSkewPolynomial

— defclass SparseUnivariateSkewPolynomialType —

```
(defclass |SparseUnivariateSkewPolynomialType| (|UnivariateSkewPolynomialCategoryType|)
  ((parents :initform '(|UnivariateSkewPolynomialCategory|))
   (name :initform "SparseUnivariateSkewPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'ORESUP)
   (comment :initform (list
     "This is the domain of sparse univariate skew polynomials over an Ore"
     "coefficient field."
     "The multiplication is given by x a = sigma(a) x + delta a."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariateSkewPolynomial|
  (progn
    (push '|SparseUnivariateSkewPolynomial| *Domains*)
    (make-instance '|SparseUnivariateSkewPolynomialType|)))
```

———-

1.70.24 SparseUnivariateTaylorSeries

— defclass SparseUnivariateTaylorSeriesType —

```
(defclass |SparseUnivariateTaylorSeriesType| (|UnivariateTaylorSeriesCategoryType|)
  ((parents :initform '(|UnivariateTaylorSeriesCategory|))
   (name :initform "SparseUnivariateTaylorSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'SUTS)
   (comment :initform (list
     "SparseUnivariateTaylorSeries is a domain representing Taylor"
     "series in one variable with coefficients in an arbitrary ring.  The"
     "parameters of the type specify the coefficient ring, the power series"
     "variable, and the center of the power series expansion.  For example,"
     "SparseUnivariateTaylorSeries(Integer,x,3) represents Taylor"
     "series in (x - 3) with Integer coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariateTaylorSeries|
  (progn
    (push '|SparseUnivariateTaylorSeries| *Domains*)
    (make-instance '|SparseUnivariateTaylorSeriesType|)))
```

———-

1.70.25 SplitHomogeneousDirectProduct

— defclass SplitHomogeneousDirectProductType —

```
(defclass |SplitHomogeneousDirectProductType| (|DirectProductCategoryType|)
  ((parents :initform '(|DirectProductCategory|))
   (name :initform "SplitHomogeneousDirectProduct")
   (marker :initform 'domain)
   (abbreviation :initform 'SHDP)
   (comment :initform (list
     "This type represents the finite direct or cartesian product of an"
     "underlying ordered component type. The vectors are ordered as if"
     "they were split into two blocks. The dim1 parameter specifies the"
     "length of the first block. The ordering is lexicographic between"
     "the blocks but acts like HomogeneousDirectProduct"
     "within each block. This type is a suitable third argument for"
     "GeneralDistributedMultivariatePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SplitHomogeneousDirectProduct|
  (progn
    (push '|SplitHomogeneousDirectProduct| *Domains*)
    (make-instance '|SplitHomogeneousDirectProductType|)))
```

———-

1.70.26 SplittingNode

— defclass SplittingNodeType —

```
(defclass |SplittingNodeType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SplittingNode")
   (marker :initform 'domain)
   (abbreviation :initform 'SPLNODE)
   (comment :initform (list
     "This domain exports a modest implementation for the"
     "vertices of splitting trees. These vertices are called"
     "here splitting nodes. Every of these nodes store 3 informations."
     "The first one is its value, that is the current expression"
     "to evaluate. The second one is its condition, that is the"
     "hypothesis under which the value has to be evaluated."
     "The last one is its status, that is a boolean flag"
     "which is true iff the value is the result of its"
     "evaluation under its condition. Two splitting vertices"
     "are equal iff they have the sane values and the same"
     "conditions (so their status do not matter)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SplittingNode|
  (progn
    (push '|SplittingNode| *Domains*)
    (make-instance '|SplittingNodeType|)))
```

———–

1.70.27 SplittingTree

— defclass SplittingTreeType —

```
(defclass |SplittingTreeType| (|RecursiveAggregateType|)
  ((parents :initform '(|RecursiveAggregate|))
   (name :initform "SplittingTree")
   (marker :initform 'domain)
   (abbreviation :initform 'SPLTREE)
   (comment :initform (list
     "This domain exports a modest implementation of splitting"
     "trees. Spliiting trees are needed when the"
     "evaluation of some quantity under some hypothesis"
     "requires to split the hypothesis into sub-cases."
     "For instance by adding some new hypothesis on one"
     "hand and its negation on another hand. The computations"
     "are terminated is a splitting tree a when"
     "status(value(a)) is true. Thus,"
     "if for the splitting tree a the flag"
     "status(value(a)) is true, then"
     "status(value(d)) is true for any"
     "subtree d of a. This property"
     "of splitting trees is called the termination"
     "condition. If no vertex in a splitting tree a"
     "is equal to another, a is said to satisfy"
     "the no-duplicates condition. The splitting "
     "tree a will satisfy this condition"
     "if nodes are added to \axiom{a} by mean of"
     "splitNodeOf! and if construct"
     "is only used to create the root of a"
     "with no children."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SplittingTree|
  (progn
    (push '|SplittingTree| *Domains*)
    (make-instance '|SplittingTreeType|)))
```

———-

1.70.28 SquareFreeRegularTriangularSet

— defclass SquareFreeRegularTriangularSetType —

```
(defclass |SquareFreeRegularTriangularSetType| (|SquareFreeRegularTriangularSetCategoryType|)
  ((parents :initform '(|SquareFreeRegularTriangularSetCategory|))
   (name :initform "SquareFreeRegularTriangularSet")
   (marker :initform 'domain)
   (abbreviation :initform 'SREGSET)
   (comment :initform (list
     "This domain provides an implementation of square-free regular chains."
     "Moreover, the operation zeroSetSplit"
     "is an implementation of a new algorithm for solving polynomial systems by"
     "means of regular chains."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularTriangularSet|
  (progn
    (push '|SquareFreeRegularTriangularSet| *Domains*)
    (make-instance '|SquareFreeRegularTriangularSetType|)))
```

———-

1.70.29 SquareMatrix

— defclass SquareMatrixType —

```
(defclass |SquareMatrixType| (|SquareMatrixCategoryType| |ConvertibleToType|)
  ((parents :initform '(|SquareMatrixCategory| |ConvertibleTo|))
   (name :initform "SquareMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'SQMATRIX)
   (comment :initform (list
     "SquareMatrix is a matrix domain of square matrices, where the"
     "number of rows (= number of columns) is a parameter of the type."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareMatrix|
  (progn
    (push '|SquareMatrix| *Domains*)
    (make-instance '|SquareMatrixType|)))
```

———-

1.70.30 Stack

— defclass StackType —

```
(defclass |StackType| (|StackAggregateType|)
  ((parents :initform '(|StackAggregate|))
   (name :initform "Stack")
   (marker :initform 'domain)
   (abbreviation :initform 'STACK)
   (comment :initform (list
     "Linked List implementation of a Stack"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Stack|
  (progn
    (push '|Stack| *Domains*)
    (make-instance '|StackType|)))
```

———-

1.70.31 StochasticDifferential

— defclass StochasticDifferentialType —

```
(defclass |StochasticDifferentialType| (|RngType| |RetractableToType| |ModuleType|)
  ((parents :initform '(|Rng| |RetractableTo| |Module|))
   (name :initform "StochasticDifferential")
   (marker :initform 'domain)
   (abbreviation :initform 'SD)
   (comment :initform (list
     "A basic implementation of StochasticDifferential(R) using the"
     "associated domain BasicStochasticDifferential in the underlying"
     "representation as sparse multivariate polynomials. The domain is"
     "a module over Expression(R), and is a ring without identity"
     "(AXIOM term is 'Rng'). Note that separate instances, for example"
     "using R=Integer and R=Float, have different hidden structure"
     "(multiplication and drift tables)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StochasticDifferential|
  (progn
    (push '|StochasticDifferential| *Domains*)
    (make-instance '|StochasticDifferentialType|)))
```

———-

1.70.32 Stream

— defclass StreamType —

```
(defclass |StreamType| (|LazyStreamAggregateType|)
  ((parents :initform '(|LazyStreamAggregate|))
   (name :initform "Stream")
   (marker :initform 'domain)
   (abbreviation :initform 'STREAM)
   (comment :initform (list
     "A stream is an implementation of an infinite sequence using"
     "a list of terms that have been computed and a function closure"
     "to compute additional terms when needed."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Stream|
  (progn
    (push '|Stream| *Domains*)
    (make-instance '|StreamType|)))
```

———-

1.70.33 String

— defclass StringType —

```
(defclass |StringType| (|StringCategoryType|)
  ((parents :initform '(|StringCategory|))
   (name :initform "String")
   (marker :initform 'domain)
   (abbreviation :initform 'STRING)
   (comment :initform (list
     "This is the domain of character strings. Strings are 1 based."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |String|
  (progn
    (push '|String| *Domains*)
    (make-instance '|StringType|)))
```

——————-

1.70.34 StringTable

— defclass StringTableType —

```
(defclass |StringTableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "StringTable")
   (marker :initform 'domain)
   (abbreviation :initform 'STRTBL)
   (comment :initform (list
     "This domain provides tables where the keys are strings."
     "A specialized hash function for strings is used."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StringTable|
  (progn
    (push '|StringTable| *Domains*)
    (make-instance '|StringTableType|)))
```

——————-

1.70.35 SubSpace

— defclass SubSpaceType —

```
(defclass |SubSpaceType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SubSpace")
   (marker :initform 'domain)
   (abbreviation :initform 'SUBSPACE)
   (comment :initform (list
     "This domain is not documented"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SubSpace|
  (progn
    (push '|SubSpace| *Domains*)
    (make-instance '|SubSpaceType|)))
```

———-

1.70.36 SubSpaceComponentProperty

— defclass SubSpaceComponentPropertyType —

```
(defclass |SubSpaceComponentPropertyType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SubSpaceComponentProperty")
   (marker :initform 'domain)
   (abbreviation :initform 'COMPPROP)
   (comment :initform (list
     "This domain implements some global properties of subspaces."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SubSpaceComponentProperty|
  (progn
    (push '|SubSpaceComponentProperty| *Domains*)
    (make-instance '|SubSpaceComponentPropertyType|)))
```

———-

1.70.37 SuchThat

— defclass SuchThatType —

```
(defclass |SuchThatType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SuchThat")
   (marker :initform 'domain)
   (abbreviation :initform 'SUCH)
   (comment :initform (list
     "This domain implements 'such that' forms"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |SuchThat|
  (progn
    (push '|SuchThat| *Domains*)
    (make-instance '|SuchThatType|)))
```

———-

1.70.38 Switch

— defclass SwitchType —

```
(defclass |SwitchType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "Switch")
   (marker :initform 'domain)
   (abbreviation :initform 'SWITCH)
   (comment :initform (list
     "This domain builds representations of boolean expressions for use with"
     "the FortranCode domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Switch|
  (progn
    (push '|Switch| *Domains*)
    (make-instance '|SwitchType|)))
```

———-

1.70.39 Symbol

— defclass SymbolType —

```
(defclass |SymbolType| (|PatternMatchableType|
                        |OrderedSetType|
                        |OpenMathType|
                        |ConvertibleToType|)
  ((parents :initform '(|PatternMatchable|
                        |OrderedSet|
                        |OpenMath|
                        |ConvertibleTo|))
   (name :initform "Symbol")
   (marker :initform 'domain)
   (abbreviation :initform 'SYMBOL)
   (comment :initform (list
     "Basic and scripted symbols."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Symbol|
  (progn
    (push '|Symbol| *Domains*)
    (make-instance '|SymbolType|)))
```

———-

1.70.40 SymbolTable

— defclass SymbolTableType —

```
(defclass |SymbolTableType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "SymbolTable")
   (marker :initform 'domain)
   (abbreviation :initform 'SYMTAB)
   (comment :initform (list
     "Create and manipulate a symbol table for generated FORTRAN code"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SymbolTable|
  (progn
    (push '|SymbolTable| *Domains*)
    (make-instance '|SymbolTableType|)))
```

———-

1.70.41 SymmetricPolynomial

— defclass SymmetricPolynomialType —

```
(defclass |SymmetricPolynomialType| (|FiniteAbelianMonoidRingType|)
  ((parents :initform '(|FiniteAbelianMonoidRing|))
   (name :initform "SymmetricPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'SYMPOLY)
   (comment :initform (list
     "This domain implements symmetric polynomial"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |SymmetricPolynomial|
  (progn
    (push '|SymmetricPolynomial| *Domains*)
    (make-instance '|SymmetricPolynomialType|)))
```

———-

1.71 T

1.71.1 Table

— defclass TableType —

```
(defclass |TableType| (|TableAggregateType|)
  ((parents :initform '(|TableAggregate|))
   (name :initform "Table")
   (marker :initform 'domain)
   (abbreviation :initform 'TABLE)
   (comment :initform (list
     "This is the general purpose table type."
     "The keys are hashed to look up the entries."
     "This creates a HashTable if equal for the Key"
     "domain is consistent with Lisp EQUAL otherwise an"
     "AssociationList"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Table|
  (progn
    (push '|Table| *Domains*)
    (make-instance '|TableType|)))
```

———-

1.71.2 Tableau

— defclass TableauType —

```
(defclass |TableauType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Tableau")
   (marker :initform 'domain)
   (abbreviation :initform 'TABLEAU)
   (comment :initform (list
     "The tableau domain is for printing Young tableaux, and"
     "coercions to and from List List S where S is a set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Tableau|
  (progn
    (push '|Tableau| *Domains*)
    (make-instance '|TableauType|)))
```

———-

1.71.3 TaylorSerieso

— defclass TaylorSeriesoType —

```
(defclass |TaylorSeriesoType| (|MultivariateTaylorSeriesCategoryType|)
  ((parents :initform '(|MultivariateTaylorSeriesCategory|))
   (name :initform "TaylorSerieso")
   (marker :initform 'domain)
   (abbreviation :initform 'TS)
   (comment :initform (list
     "TaylorSeries is a general multivariate Taylor series domain"
     "over the ring Coef and with variables of type Symbol."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TaylorSerieso|
  (progn
    (push '|TaylorSerieso| *Domains*)
    (make-instance '|TaylorSeriesoType|)))
```

———-

1.71.4 TexFormat

— defclass TexFormatType —

```
(defclass |TexFormatType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "TexFormat")
   (marker :initform 'domain)
   (abbreviation :initform 'TEX)
   (comment :initform (list
     "TexFormat provides a coercion from OutputForm to"
     "TeX format.  The particular dialect of TeX used is LaTeX."
     "The basic object consists of three parts: a prologue, a"
     "tex part and an epilogue. The functions prologue,"
     "tex and epilogue extract these parts,"
     "respectively.  The main guts of the expression go into the tex part."
     "The other parts can be set (setPrologue!,"
     "setEpilogue!) so that contain the appropriate tags for"
     "printing. For example, the prologue and epilogue might simply"
     "contain '\\verb+\\[+' and '\\verb+\\]+', respectively, so that"
     "the TeX section will be printed in LaTeX display math mode."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TexFormat|
  (progn
    (push '|TexFormat| *Domains*)
    (make-instance '|TexFormatType|)))
```

————-

1.71.5 TextFile

— defclass TextFileType —

```
(defclass |TextFileType| (|FileCategoryType|)
  ((parents :initform '(|FileCategory|))
   (name :initform "TextFile")
   (marker :initform 'domain)
   (abbreviation :initform 'TEXTFILE)
   (comment :initform (list
     "This domain provides an implementation of text files.  Text is stored"
     "in these files using the native character set of the computer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TextFile|
  (progn
    (push '|TextFile| *Domains*)
    (make-instance '|TextFileType|)))
```

————-

1.71.6 TheSymbolTable

— defclass TheSymbolTableType —

```
(defclass |TheSymbolTableType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "TheSymbolTable")
   (marker :initform 'domain)
   (abbreviation :initform 'SYMS)
   (comment :initform (list
     "Creates and manipulates one global symbol table for FORTRAN"
     "code generation, containing details of types, dimensions, and argument"
     "lists."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TheSymbolTable|
  (progn
    (push '|TheSymbolTable| *Domains*)
    (make-instance '|TheSymbolTableType|)))
```

———-

1.71.7 ThreeDimensionalMatrix

— defclass ThreeDimensionalMatrixType —

```
(defclass |ThreeDimensionalMatrixType| (|HomogeneousAggregateType|)
  ((parents :initform '(|HomogeneousAggregate|))
   (name :initform "ThreeDimensionalMatrix")
   (marker :initform 'domain)
   (abbreviation :initform 'M3D)
   (comment :initform (list
     "This domain represents three dimensional matrices over a general object type"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ThreeDimensionalMatrix|
  (progn
    (push '|ThreeDimensionalMatrix| *Domains*)
    (make-instance '|ThreeDimensionalMatrixType|)))
```

———-

1.71.8 ThreeDimensionalViewport

— defclass ThreeDimensionalViewportType —

```
(defclass |ThreeDimensionalViewportType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "ThreeDimensionalViewport")
   (marker :initform 'domain)
   (abbreviation :initform 'VIEW3D)
   (comment :initform (list
     "ThreeDimensionalViewport creates viewports to display graphs"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ThreeDimensionalViewport|
  (progn
    (push '|ThreeDimensionalViewport| *Domains*)
    (make-instance '|ThreeDimensionalViewportType|)))
```

————-

1.71.9 ThreeSpace

— defclass ThreeSpaceType —

```
(defclass |ThreeSpaceType| (|ThreeSpaceCategoryType|)
  ((parents :initform '(|ThreeSpaceCategory|))
   (name :initform "ThreeSpace")
   (marker :initform 'domain)
   (abbreviation :initform 'SPACE3)
   (comment :initform (list
     "The domain ThreeSpace is used for creating three dimensional"
     "objects using functions for defining points, curves, polygons, constructs"
     "and the subspaces containing them."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ThreeSpace|
  (progn
    (push '|ThreeSpace| *Domains*)
    (make-instance '|ThreeSpaceType|)))
```

————-

1.71.10 Tree

— defclass TreeType —

```
(defclass |TreeType| (|RecursiveAggregateType|)
  ((parents :initform '(|RecursiveAggregate|))
   (name :initform "Tree")
   (marker :initform 'domain)
   (abbreviation :initform 'TREE)
   (comment :initform (list
     "Tree(S) is a basic domains of tree structures."
     "Each tree is either empty or else is a node consisting of a value and"
     "a list of (sub)trees."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Tree|
  (progn
    (push '|Tree| *Domains*)
    (make-instance '|TreeType|)))
```

———-

1.71.11 TubePlot

— defclass TubePlotType —

```
(defclass |TubePlotType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TubePlot")
   (marker :initform 'domain)
   (abbreviation :initform 'TUBE)
   (comment :initform (list
     "Package for constructing tubes around 3-dimensional parametric curves."
     "Domain of tubes around 3-dimensional parametric curves."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TubePlot|
  (progn
    (push '|TubePlot| *Domains*)
    (make-instance '|TubePlotType|)))
```

———-

1.71.12 Tuple

— defclass TupleType —

```
(defclass |TupleType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Tuple")
   (marker :initform 'domain)
   (abbreviation :initform 'TUPLE)
   (comment :initform (list
     "This domain is used to interface with the interpreter's notion"
     "of comma-delimited sequences of values."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Tuple|
  (progn
    (push '|Tuple| *Domains*)
    (make-instance '|TupleType|)))
```

———-

1.71.13 TwoDimensionalArray

— defclass TwoDimensionalArrayType —

```
(defclass |TwoDimensionalArrayType| (|TwoDimensionalArrayCategoryType|)
  ((parents :initform '(|TwoDimensionalArrayCategory|))
   (name :initform "TwoDimensionalArray")
   (marker :initform 'domain)
   (abbreviation :initform 'ARRAY2)
   (comment :initform (list
     "A TwoDimensionalArray is a two dimensional array with"
     "1-based indexing for both rows and columns."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoDimensionalArray|
  (progn
    (push '|TwoDimensionalArray| *Domains*)
    (make-instance '|TwoDimensionalArrayType|)))
```

———-

1.71.14 TwoDimensionalViewport

— defclass TwoDimensionalViewportType —

```
(defclass |TwoDimensionalViewportType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "TwoDimensionalViewport")
   (marker :initform 'domain)
   (abbreviation :initform 'VIEW2D)
   (comment :initform (list
     "TwoDimensionalViewport creates viewports to display graphs."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |TwoDimensionalViewport|
  (progn
    (push '|TwoDimensionalViewport| *Domains*)
    (make-instance '|TwoDimensionalViewportType|)))
```

———-

1.72 U

1.72.1 UnivariateFormalPowerSeries

— defclass UnivariateFormalPowerSeriesType —

```
(defclass |UnivariateFormalPowerSeriesType| (|UnivariateTaylorSeriesCategoryType|)
  ((parents :initform '(|UnivariateTaylorSeriesCategory|))
   (name :initform "UnivariateFormalPowerSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'UFPS)
   (comment :initform (list
     "This domain has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateFormalPowerSeries|
  (progn
    (push '|UnivariateFormalPowerSeries| *Domains*)
    (make-instance '|UnivariateFormalPowerSeriesType|)))
```

———-

1.72.2 UnivariateLaurentSeries

— defclass UnivariateLaurentSeriesType —

```
(defclass |UnivariateLaurentSeriesType| (|UnivariateLaurentSeriesConstructorCategoryType|)
  ((parents :initform '(|UnivariateLaurentSeriesConstructorCategory|))
   (name :initform "UnivariateLaurentSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'ULS)
   (comment :initform (list
     "UnivariateLaurentSeries is a domain representing Laurent"
     "series in one variable with coefficients in an arbitrary ring.  The"
     "parameters of the type specify the coefficient ring, the power series"
     "variable, and the center of the power series expansion.  For example,"
     "UnivariateLaurentSeries(Integer,x,3) represents Laurent series in"
     "(x - 3) with integer coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeries|
  (progn
    (push '|UnivariateLaurentSeries| *Domains*)
    (make-instance '|UnivariateLaurentSeriesType|)))
```

————-

1.72.3 UnivariateLaurentSeriesConstructor

— defclass UnivariateLaurentSeriesConstructorType —

```
(defclass |UnivariateLaurentSeriesConstructorType| (|UnivariateLaurentSeriesConstructorCategoryType|)
  ((parents :initform '(|UnivariateLaurentSeriesConstructorCategory|))
   (name :initform "UnivariateLaurentSeriesConstructor")
   (marker :initform 'domain)
   (abbreviation :initform 'ULSCONS)
   (comment :initform (list
     "This package enables one to construct a univariate Laurent series"
     "domain from a univariate Taylor series domain. Univariate"
     "Laurent series are represented by a pair [n,f(x)], where n is"
     "an arbitrary integer and f(x)} is a Taylor series.  This pair"
     "represents the Laurent series x**n * f(x)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeriesConstructor|
  (progn
    (push '|UnivariateLaurentSeriesConstructor| *Domains*)
    (make-instance '|UnivariateLaurentSeriesConstructorType|)))
```

————-

1.72.4 UnivariatePolynomial

— defclass UnivariatePolynomialType —

```
(defclass |UnivariatePolynomialType| (|UnivariatePolynomialCategoryType|)
  ((parents :initform '(|UnivariatePolynomialCategory|))
   (name :initform "UnivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'UP)
   (comment :initform (list
     "This domain represents univariate polynomials in some symbol"
     "over arbitrary (not necessarily commutative) coefficient rings."
     "The representation is sparse"
     "in the sense that only non-zero terms are represented."
     "Note that if the coefficient ring is a field, then this domain"
     "forms a euclidean domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomial|
  (progn
    (push '|UnivariatePolynomial| *Domains*)
    (make-instance '|UnivariatePolynomialType|)))
```

———-

1.72.5 UnivariatePuiseuxSeries

— defclass UnivariatePuiseuxSeriesType —

```
(defclass |UnivariatePuiseuxSeriesType| (|UnivariatePuiseuxSeriesConstructorCategoryType|)
  ((parents :initform '(|UnivariatePuiseuxSeriesConstructorCategory|))
   (name :initform "UnivariatePuiseuxSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'UPXS)
   (comment :initform (list
     "Dense Puiseux series in one variable"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeries|
  (progn
    (push '|UnivariatePuiseuxSeries| *Domains*)
    (make-instance '|UnivariatePuiseuxSeriesType|)))
```

———-

1.72.6 UnivariatePuiseuxSeriesConstructor

— defclass UnivariatePuiseuxSeriesConstructorType —

```
(defclass |UnivariatePuiseuxSeriesConstructorType| (|UnivariatePuiseuxSeriesConstructorCategoryType|)
  ((parents :initform '(|UnivariatePuiseuxSeriesConstructorCategory|))
   (name :initform "UnivariatePuiseuxSeriesConstructor")
   (marker :initform 'domain)
   (abbreviation :initform 'UPXSCONS)
   (comment :initform (list
     "This package enables one to construct a univariate Puiseux series"
     "domain from a univariate Laurent series domain. Univariate"
     "Puiseux series are represented by a pair [r,f(x)], where r is"
     "a positive rational number and f(x) is a Laurent series."
     "This pair represents the Puiseux series f(x^r)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesConstructor|
  (progn
    (push '|UnivariatePuiseuxSeriesConstructor| *Domains*)
    (make-instance '|UnivariatePuiseuxSeriesConstructorType|)))
```

———-

1.72.7 UnivariatePuiseuxSeriesWithExponentialSingularity

— defclass UnivariatePuiseuxSeriesWithExponentialSingularityType —

```
(defclass |UnivariatePuiseuxSeriesWithExponentialSingularityType| (|FiniteAbelianMonoidRingType|)
  ((parents :initform '(|FiniteAbelianMonoidRing|))
   (name :initform "UnivariatePuiseuxSeriesWithExponentialSingularity")
   (marker :initform 'domain)
   (abbreviation :initform 'UPXSSING)
   (comment :initform (list
     "UnivariatePuiseuxSeriesWithExponentialSingularity is a domain used to"
     "represent functions with essential singularities.  Objects in this"
     "domain are sums, where each term in the sum is a univariate Puiseux"
     "series times the exponential of a univariate Puiseux series.  Thus,"
     "the elements of this domain are sums of expressions of the form"
     "g(x) * exp(f(x)), where g(x) is a univariate Puiseux series"
     "and f(x) is a univariate Puiseux series with no terms of non-negative"
     "degree."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesWithExponentialSingularity|
  (progn
    (push '|UnivariatePuiseuxSeriesWithExponentialSingularity| *Domains*)
    (make-instance '|UnivariatePuiseuxSeriesWithExponentialSingularityType|)))
```

———-

1.72.8 UnivariateSkewPolynomial

— defclass UnivariateSkewPolynomialType —

```
(defclass |UnivariateSkewPolynomialType| (|UnivariateSkewPolynomialCategoryType|)
  ((parents :initform '(|UnivariateSkewPolynomialCategory|))
   (name :initform "UnivariateSkewPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'OREUP)
   (comment :initform (list
     "This is the domain of univariate skew polynomials over an Ore"
     "coefficient field in a named variable."
     "The multiplication is given by x a = sigma(a) x + delta a."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateSkewPolynomial|
  (progn
    (push '|UnivariateSkewPolynomial| *Domains*)
    (make-instance '|UnivariateSkewPolynomialType|)))
```

———-

1.72.9 UnivariateTaylorSeries

— defclass UnivariateTaylorSeriesType —

```
(defclass |UnivariateTaylorSeriesType| (|UnivariateTaylorSeriesCategoryType|)
  ((parents :initform '(|UnivariateTaylorSeriesCategory|))
   (name :initform "UnivariateTaylorSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'UTS)
   (comment :initform (list
     "Dense Taylor series in one variable"
     "UnivariateTaylorSeries is a domain representing Taylor"
     "series in"
     "one variable with coefficients in an arbitrary ring.  The parameters"
     "of the type specify the coefficient ring, the power series variable,"
     "and the center of the power series expansion.  For example,"
     "UnivariateTaylorSeries(Integer,x,3) represents"
     "Taylor series in"
     "(x - 3) with Integer coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeries|
  (progn
    (push '|UnivariateTaylorSeries| *Domains*)
    (make-instance '|UnivariateTaylorSeriesType|)))
```

———-

1.72.10 UnivariateTaylorSeriesCZero

— defclass UnivariateTaylorSeriesCZeroType —

```
(defclass |UnivariateTaylorSeriesCZeroType| (|UnivariateTaylorSeriesCategoryType|)
  ((parents :initform '(|UnivariateTaylorSeriesCategory|))
   (name :initform "UnivariateTaylorSeriesCZero")
   (marker :initform 'domain)
   (abbreviation :initform 'UTSZ)
   (comment :initform (list
     "Part of the Package for Algebraic Function Fields in one variable PAFF"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesCZero|
  (progn
    (push '|UnivariateTaylorSeriesCZero| *Domains*)
    (make-instance '|UnivariateTaylorSeriesCZeroType|)))
```

———-

1.72.11 UniversalSegment

— defclass UniversalSegmentType —

```
(defclass |UniversalSegmentType| (|SetCategoryType| |SegmentExpansionCategoryType|)
  ((parents :initform '(|SetCategory| |SegmentExpansionCategory|))
   (name :initform "UniversalSegment")
   (marker :initform 'domain)
   (abbreviation :initform 'UNISEG)
   (comment :initform (list
     "Part of the Package for Algebraic Function Fields in one variable PAFF"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UniversalSegment|
  (progn
    (push '|UniversalSegment| *Domains*)
    (make-instance '|UniversalSegmentType|)))
```

———-

1.72.12 U8Matrix

— defclass U8MatrixType —

```
(defclass |U8MatrixType| (|MatrixCategoryType|)
  ((parents :initform '(|MatrixCategory|))
   (name :initform "U8Matrix")
   (marker :initform 'domain)
   (abbreviation :initform 'U8MAT)
   (comment :initform (list
     "This is a low-level domain which implements matrices"
     "(two dimensional arrays) of 8-bit integers."
     "Indexing is 0 based, there is no bound checking (unless"
     "provided by lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U8Matrix|
  (progn
    (push '|U8Matrix| *Domains*)
    (make-instance '|U8MatrixType|)))
```

———-

1.72.13 U16Matrix

— defclass U16MatrixType —

```
(defclass |U16MatrixType| (|MatrixCategoryType|)
  ((parents :initform '(|MatrixCategory|))
   (name :initform "U16Matrix")
   (marker :initform 'domain)
   (abbreviation :initform 'U16MAT)
   (comment :initform (list
     "This is a low-level domain which implements matrices"
     "(two dimensional arrays) of 16-bit integers."
     "Indexing is 0 based, there is no bound checking (unless"
     "provided by lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U16Matrix|
  (progn
    (push '|U16Matrix| *Domains*)
    (make-instance '|U16MatrixType|)))
```

———-

1.72.14 U32Matrix

— defclass U32MatrixType —

```
(defclass |U32MatrixType| (|MatrixCategoryType|)
  ((parents :initform '(|MatrixCategory|))
   (name :initform "U32Matrix")
   (marker :initform 'domain)
   (abbreviation :initform 'U32MAT)
   (comment :initform (list
     "This is a low-level domain which implements matrices"
     "(two dimensional arrays) of 32-bit integers."
     "Indexing is 0 based, there is no bound checking (unless"
     "provided by lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U32Matrix|
  (progn
    (push '|U32Matrix| *Domains*)
    (make-instance '|U32MatrixType|)))
```

———-

1.72.15 U8Vector

— defclass U8VectorType —

```
(defclass |U8VectorType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "U8Vector")
   (marker :initform 'domain)
   (abbreviation :initform 'U8VEC)
   (comment :initform (list
     "This is a low-level domain which implements vectors"
     "(one dimensional arrays) of unsigned 8-bit numbers.  Indexing"
     "is 0 based, there is no bound checking (unless provided by"
     "lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U8Vector|
  (progn
    (push '|U8Vector| *Domains*)
    (make-instance '|U8VectorType|)))
```

———-

1.72.16 U16Vector

— defclass U16VectorType —

```
(defclass |U16VectorType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "U16Vector")
   (marker :initform 'domain)
   (abbreviation :initform 'U16VEC)
   (comment :initform (list
     "This is a low-level domain which implements vectors"
     "(one dimensional arrays) of unsigned 16-bit numbers.  Indexing"
     "is 0 based, there is no bound checking (unless provided by"
     "lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U16Vector|
  (progn
    (push '|U16Vector| *Domains*)
    (make-instance '|U16VectorType|)))
```

———–

1.72.17 U32Vector

— defclass U32VectorType —

```
(defclass |U32VectorType| (|OneDimensionalArrayAggregateType|)
  ((parents :initform '(|OneDimensionalArrayAggregate|))
   (name :initform "U32Vector")
   (marker :initform 'domain)
   (abbreviation :initform 'U32VEC)
   (comment :initform (list
     "This is a low-level domain which implements vectors"
     "(one dimensional arrays) of unsigned 32-bit numbers.  Indexing"
     "is 0 based, there is no bound checking (unless provided by"
     "lower level)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U32Vector|
  (progn
    (push '|U32Vector| *Domains*)
    (make-instance '|U32VectorType|)))
```

———–

1.73 V

1.73.1 Variable

— defclass VariableType —

```
(defclass |VariableType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Variable")
   (marker :initform 'domain)
   (abbreviation :initform 'VARIABLE)
   (comment :initform (list
     "This domain implements variables"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Variable|
  (progn
    (push '|Variable| *Domains*)
    (make-instance '|VariableType|)))
```

———-

1.73.2 Vector

— defclass VectorType —

```
(defclass |VectorType| (|VectorCategoryType|)
  ((parents :initform '(|VectorCategory|))
   (name :initform "Vector")
   (marker :initform 'domain)
   (abbreviation :initform 'VECTOR)
   (comment :initform (list
     "This type represents vector like objects with varying lengths"
     "and indexed by a finite segment of integers starting at 1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Vector|
  (progn
    (push '|Vector| *Domains*)
    (make-instance '|VectorType|)))
```

———-

1.73.3 Void

— defclass VoidType —

```
(defclass |VoidType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Void")
   (marker :initform 'domain)
   (abbreviation :initform 'VOID)
   (comment :initform (list
     "This type is used when no value is needed, for example, in the then"
     "part of a one armed if."
     "All values can be coerced to type Void.  Once a value has been coerced"
     "to Void, it cannot be recovered."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Void|
  (progn
    (push '|Void| *Domains*)
    (make-instance '|VoidType|)))
```

——————-

1.74 W

1.74.1 WeightedPolynomials

— defclass WeightedPolynomialsType —

```
(defclass |WeightedPolynomialsType| (|AlgebraType|)
  ((parents :initform '(|Algebra|))
   (name :initform "WeightedPolynomials")
   (marker :initform 'domain)
   (abbreviation :initform 'WP)
   (comment :initform (list
     "This domain represents truncated weighted polynomials over a general"
     "(not necessarily commutative) polynomial type. The variables must be"
     "specified, as must the weights."
     "The representation is sparse"
     "in the sense that only non-zero terms are represented."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |WeightedPolynomials|
  (progn
    (push '|WeightedPolynomials| *Domains*)
    (make-instance '|WeightedPolynomialsType|)))
```

———-

1.74.2 WuWenTsunTriangularSet

— defclass WuWenTsunTriangularSetType —

```
(defclass |WuWenTsunTriangularSetType| (|TriangularSetCategoryType|)
  ((parents :initform '(|TriangularSetCategory|))
   (name :initform "WuWenTsunTriangularSet")
   (marker :initform 'domain)
   (abbreviation :initform 'WUTSET)
   (comment :initform (list
     "A domain constructor of the category GeneralTriangularSet."
     "The only requirement for a list of polynomials to be a member of such"
     "a domain is the following: no polynomial is constant and two distinct"
     "polynomials have distinct main variables. Such a triangular set may"
     "not be auto-reduced or consistent. The construct operation"
     "does not check the previous requirement. Triangular sets are stored"
     "as sorted lists w.r.t. the main variables of their members."
     "Furthermore, this domain exports operations dealing with the"
     "characteristic set method of Wu Wen Tsun and some optimizations"
     "mainly proposed by Dong Ming Wang."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |WuWenTsunTriangularSet|
  (progn
    (push '|WuWenTsunTriangularSet| *Domains*)
    (make-instance '|WuWenTsunTriangularSetType|)))
```

———-

1.75 X

1.75.1 XDistributedPolynomial

— defclass XDistributedPolynomialType —

```
(defclass |XDistributedPolynomialType| (|FreeModuleCatType| |XPolynomialsCatType|)
  ((parents :initform '(|FreeModuleCat| |XPolynomialsCat|))
   (name :initform "XDistributedPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'XDPOLY)
   (comment :initform (list
     "This type supports distributed multivariate polynomials"
     "whose variables do not commute."
     "The coefficient ring may be non-commutative too."
     "However, coefficients and variables commute."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XDistributedPolynomial|
  (progn
    (push '|XDistributedPolynomial| *Domains*)
    (make-instance '|XDistributedPolynomialType|)))
```

———-

1.75.2 XPBWPolynomial

— defclass XPBWPolynomialType —

```
(defclass |XPBWPolynomialType| (|FreeModuleCatType| |XPolynomialsCatType|)
  ((parents :initform '(|FreeModuleCat| |XPolynomialsCat|))
   (name :initform "XPBWPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'XPBWPOLY)
   (comment :initform (list
     "This domain constructor implements polynomials in non-commutative"
     "variables written in the Poincare-Birkhoff-Witt basis from the"
     "Lyndon basis."
     "These polynomials can be used to compute Baker-Campbell-Hausdorff"
     "relations. "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPBWPolynomial|
  (progn
    (push '|XPBWPolynomial| *Domains*)
    (make-instance '|XPBWPolynomialType|)))
```

———-

1.75.3 XPolynomial

— defclass XPolynomialType —

```
(defclass |XPolynomialType| (|XPolynomialsCatType|)
  ((parents :initform '(|XPolynomialsCat|))
   (name :initform "XPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'XPOLY)
   (comment :initform (list
     "This type supports multivariate polynomials whose set of variables"
     "is Symbol. The representation is recursive."
     "The coefficient ring may be non-commutative and the variables"
     "do not commute. However, coefficients and variables commute."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPolynomial|
  (progn
    (push '|XPolynomial| *Domains*)
    (make-instance '|XPolynomialType|)))
```

———-

1.75.4 XPolynomialRing

— defclass XPolynomialRingType —

```
(defclass |XPolynomialRingType| (|FreeModuleCatType| |XAlgebraType|)
  ((parents :initform '(|FreeModuleCat| |XAlgebra|))
   (name :initform "XPolynomialRing")
   (marker :initform 'domain)
   (abbreviation :initform 'XPR)
   (comment :initform (list
     "This domain represents generalized polynomials with coefficients"
     "(from a not necessarily commutative ring), and words"
     "belonging to an arbitrary OrderedMonoid."
     "This type is used, for instance, by the XDistributedPolynomial"
     "domain constructor where the Monoid is free."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPolynomialRing|
  (progn
    (push '|XPolynomialRing| *Domains*)
    (make-instance '|XPolynomialRingType|)))
```

———-

1.75.5 XRecursivePolynomial

— defclass XRecursivePolynomialType —

```
(defclass |XRecursivePolynomialType| (|XPolynomialsCatType|)
  ((parents :initform '(|XPolynomialsCat|))
   (name :initform "XRecursivePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'XRPOLY)
   (comment :initform (list
     "This type supports multivariate polynomials whose variables do not commute."
     "The representation is recursive. The coefficient ring may be"
     "non-commutative. Coefficients and variables commute."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XRecursivePolynomial|
  (progn
    (push '|XRecursivePolynomial| *Domains*)
    (make-instance '|XRecursivePolynomialType|)))
```

——————-

The Packages

1.76 A

1.76.1 AffineAlgebraicSetComputeWithGroebnerBasis

— defclass AffineAlgebraicSetComputeWithGroebnerBasisType —

```
(defclass |AffineAlgebraicSetComputeWithGroebnerBasisType| (|AxiomClass|)
  ((parents :initform '())
   (name :initform "AffineAlgebraicSetComputeWithGroebnerBasis")
   (marker :initform 'package)
   (abbreviation :initform 'AFALGGRO)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffineAlgebraicSetComputeWithGroebnerBasis|
  (progn
    (push '|AffineAlgebraicSetComputeWithGroebnerBasis| *Packages*)
    (make-instance '|AffineAlgebraicSetComputeWithGroebnerBasisType|)))
```

———–

1.76.2 AffineAlgebraicSetComputeWithResultant

— defclass AffineAlgebraicSetComputeWithResultantType —

```
(defclass |AffineAlgebraicSetComputeWithResultantType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AffineAlgebraicSetComputeWithResultant")
   (marker :initform 'package)
   (abbreviation :initform 'AFALGRES)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
    (withlist :initform nil)
   (haslist :initform nil)
    (addlist :initform nil)))

 (defvar |AffineAlgebraicSetComputeWithResultant|
   (progn
     (push '|AffineAlgebraicSetComputeWithResultant| *Packages*)
     (make-instance '|AffineAlgebraicSetComputeWithResultantType|)))
```

———-

1.76.3 AlgebraicFunction

— (defclass AlgebraicFunctionType —

```
 (defclass |AlgebraicFunctionType| (|AxiomClass|)
   ((parents :initform ())
    (name :initform "AlgebraicFunction")
    (marker :initform 'package)
    (abbreviation :initform 'AF)
   (comment :initform (list
     "This package provides algebraic functions over an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicFunction|
  (progn
    (push '|AlgebraicFunction| *Packages*)
    (make-instance '|AlgebraicFunctionType|)))
```

———-

1.76.4 AlgebraicHermiteIntegration

— defclass AlgebraicHermiteIntegrationType —

```
(defclass |AlgebraicHermiteIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgebraicHermiteIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTHERAL)
   (comment :initform (list
     "Algebraic Hermite reduction."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicHermiteIntegration|
  (progn
    (push '|AlgebraicHermiteIntegration| *Packages*)
    (make-instance '|AlgebraicHermiteIntegrationType|)))
```

————-

1.76.5 AlgebraicIntegrate

— defclass AlgebraicIntegrateType —

```
(defclass |AlgebraicIntegrateType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgebraicIntegrate")
   (marker :initform 'package)
   (abbreviation :initform 'INTALG)
   (comment :initform (list
     "This package provides functions for integrating a function"
     "on an algebraic curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicIntegrate|
  (progn
    (push '|AlgebraicIntegrate| *Packages*)
    (make-instance '|AlgebraicIntegrateType|)))
```

————-

1.76.6 AlgebraicIntegration

— defclass AlgebraicIntegrationType —

```
(defclass |AlgebraicIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgebraicIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTAF)
   (comment :initform (list
     "This package provides functions for the integration of"
     "algebraic integrands over transcendental functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicIntegration|
  (progn
    (push '|AlgebraicIntegration| *Packages*)
    (make-instance '|AlgebraicIntegrationType|)))
```

———-

1.76.7 AlgebraicManipulations

— defclass AlgebraicManipulationsType —

```
(defclass |AlgebraicManipulationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgebraicManipulations")
   (marker :initform 'package)
   (abbreviation :initform 'ALGMANIP)
   (comment :initform (list
     "AlgebraicManipulations provides functions to simplify and expand"
     "expressions involving algebraic operators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicManipulations|
  (progn
    (push '|AlgebraicManipulations| *Packages*)
    (make-instance '|AlgebraicManipulationsType|)))
```

———-

1.76.8 AlgebraicMultFact

— defclass AlgebraicMultFactType —

```
(defclass |AlgebraicMultFactType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgebraicMultFact")
   (marker :initform 'package)
   (abbreviation :initform 'ALGMFACT)
   (comment :initform (list
     "This package factors multivariate polynomials over the"
     "domain of AlgebraicNumber by allowing the user"
     "to specify a list of algebraic numbers generating the particular"
     "extension to factor over."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicMultFact|
  (progn
    (push '|AlgebraicMultFact| *Packages*)
    (make-instance '|AlgebraicMultFactType|)))
```

———-

1.76.9 AlgebraPackage

— defclass AlgebraPackageType —

```
(defclass |AlgebraPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgebraPackage")
   (marker :initform 'package)
   (abbreviation :initform 'ALGPKG)
   (comment :initform (list
     "AlgebraPackage assembles a variety of useful functions for"
     "general algebras."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraPackage|
  (progn
    (push '|AlgebraPackage| *Packages*)
    (make-instance '|AlgebraPackageType|)))
```

———-

1.76.10 AlgFactor

— defclass AlgFactorType —

```
(defclass |AlgFactorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgFactor")
   (marker :initform 'package)
   (abbreviation :initform 'ALGFACT)
   (comment :initform (list
     "Factorization of univariate polynomials with coefficients in"
     "AlgebraicNumber."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgFactor|
  (progn
    (push '|AlgFactor| *Packages*)
    (make-instance '|AlgFactorType|)))
```

———-

1.76.11 AnnaNumericalIntegrationPackage

— defclass AnnaNumericalIntegrationPackageType —

```
(defclass |AnnaNumericalIntegrationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AnnaNumericalIntegrationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'INTPACK)
   (comment :initform (list
     "AnnaNumericalIntegrationPackage is a package"
     "of functions for the category"
     "NumericalIntegrationCategory"
     "with measure, and integrate."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaNumericalIntegrationPackage|
  (progn
    (push '|AnnaNumericalIntegrationPackage| *Packages*)
    (make-instance '|AnnaNumericalIntegrationPackageType|)))
```

———-

1.76.12 AnnaNumericalOptimizationPackage

— defclass AnnaNumericalOptimizationPackageType —

```
(defclass |AnnaNumericalOptimizationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AnnaNumericalOptimizationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'OPTPACK)
   (comment :initform (list
     "AnnaNumericalOptimizationPackage is a package of"
     "functions for the NumericalOptimizationCategory"
     "with measure and optimize."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaNumericalOptimizationPackage|
  (progn
    (push '|AnnaNumericalOptimizationPackage| *Packages*)
    (make-instance '|AnnaNumericalOptimizationPackageType|)))
```

———-

1.76.13 AnnaOrdinaryDifferentialEquationPackage

— defclass AnnaOrdinaryDifferentialEquationPackageType —

```
(defclass |AnnaOrdinaryDifferentialEquationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AnnaOrdinaryDifferentialEquationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'ODEPACK)
   (comment :initform (list
     "AnnaOrdinaryDifferentialEquationPackage is a package"
     "of functions for the category"
     "OrdinaryDifferentialEquationsSolverCategory"
     "with measure, and solve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaOrdinaryDifferentialEquationPackage|
  (progn
    (push '|AnnaOrdinaryDifferentialEquationPackage| *Packages*)
    (make-instance '|AnnaOrdinaryDifferentialEquationPackageType|)))
```

———-

1.76.14 AnnaPartialDifferentialEquationPackage

— defclass AnnaPartialDifferentialEquationPackageType —

```
(defclass |AnnaPartialDifferentialEquationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AnnaPartialDifferentialEquationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PDEPACK)
   (comment :initform (list
     "AnnaPartialDifferentialEquationPackage is an uncompleted"
     "package for the interface to NAG PDE routines.  It has been realised that"
     "a new approach to solving PDEs will need to be created."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaPartialDifferentialEquationPackage|
  (progn
    (push '|AnnaPartialDifferentialEquationPackage| *Packages*)
    (make-instance '|AnnaPartialDifferentialEquationPackageType|)))
```

———-

1.76.15 AnyFunctions1

— defclass AnyFunctions1Type —

```
(defclass |AnyFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AnyFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'ANY1)
   (comment :initform (list
     "AnyFunctions1 implements several utility functions for"
     "working with Any. These functions are used to go back"
     "and forth between objects of Any and objects of other"
     "types."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnyFunctions1|
  (progn
    (push '|AnyFunctions1| *Packages*)
    (make-instance '|AnyFunctions1Type|)))
```

———-

1.76.16 ApplicationProgramInterface

— defclass ApplicationProgramInterfaceType —

```
(defclass |ApplicationProgramInterfaceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ApplicationProgramInterface")
   (marker :initform 'package)
   (abbreviation :initform 'API)
   (comment :initform (list
     "This package contains useful functions that expose Axiom system internals"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ApplicationProgramInterface|
  (progn
    (push '|ApplicationProgramInterface| *Packages*)
    (make-instance '|ApplicationProgramInterfaceType|)))
```

———-

1.76.17 ApplyRules

— defclass ApplyRulesType —

```
(defclass |ApplyRulesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ApplyRules")
   (marker :initform 'package)
   (abbreviation :initform 'APPRULE)
   (comment :initform (list
     "This package apply rewrite rules to expressions, calling"
     "the pattern matcher."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ApplyRules|
  (progn
    (push '|ApplyRules| *Packages*)
    (make-instance '|ApplyRulesType|)))
```

———-

1.76.18 ApplyUnivariateSkewPolynomial

— defclass ApplyUnivariateSkewPolynomialType —

```
(defclass |ApplyUnivariateSkewPolynomialType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ApplyUnivariateSkewPolynomial")
   (marker :initform 'package)
   (abbreviation :initform 'APPLYORE)
   (comment :initform (list
     "ApplyUnivariateSkewPolynomial (internal) allows univariate"
     "skew polynomials to be applied to appropriate modules."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ApplyUnivariateSkewPolynomial|
  (progn
    (push '|ApplyUnivariateSkewPolynomial| *Packages*)
    (make-instance '|ApplyUnivariateSkewPolynomialType|)))
```

———-

1.76.19 AssociatedEquations

— defclass AssociatedEquationsType —

```
(defclass |AssociatedEquationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AssociatedEquations")
   (marker :initform 'package)
   (abbreviation :initform 'ASSOCEQ)
   (comment :initform (list
     "AssociatedEquations provides functions to compute the"
     "associated equations needed for factoring operators"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociatedEquations|
  (progn
    (push '|AssociatedEquations| *Packages*)
    (make-instance '|AssociatedEquationsType|)))
```

———–

1.76.20 AttachPredicates

— defclass AttachPredicatesType —

```
(defclass |AttachPredicatesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AttachPredicates")
   (marker :initform 'package)
   (abbreviation :initform 'PMPRED)
   (comment :initform (list
     "Attaching predicates to symbols for pattern matching."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AttachPredicates|
  (progn
    (push '|AttachPredicates| *Packages*)
    (make-instance '|AttachPredicatesType|)))
```

———–

1.76.21 AxiomServer

— defclass AxiomServerType —

```
(defclass |AxiomServerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AxiomServer")
   (marker :initform 'package)
   (abbreviation :initform 'AXSERV)
   (comment :initform (list
     "This package provides a functions to support a web server for the"
     "new Axiom Browser functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AxiomServer|
  (progn
    (push '|AxiomServer| *Packages*)
    (make-instance '|AxiomServerType|)))
```

————-

1.77 B

1.77.1 BalancedFactorisation

— defclass BalancedFactorisationType —

```
(defclass |BalancedFactorisationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BalancedFactorisation")
   (marker :initform 'package)
   (abbreviation :initform 'BALFACT)
   (comment :initform (list
     "This package provides balanced factorisations of polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BalancedFactorisation|
  (progn
    (push '|BalancedFactorisation| *Packages*)
    (make-instance '|BalancedFactorisationType|)))
```

————-

1.77.2 BasicOperatorFunctions1

— defclass BasicOperatorFunctions1Type —

```
(defclass |BasicOperatorFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BasicOperatorFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'BOP1)
   (comment :initform (list
     "This package exports functions to set some commonly used properties"
     "of operators, including properties which contain functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicOperatorFunctions1|
  (progn
    (push '|BasicOperatorFunctions1| *Packages*)
    (make-instance '|BasicOperatorFunctions1Type|)))
```

————-

1.77.3 Bezier

— defclass BezierType —

```
(defclass |BezierType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Bezier")
   (marker :initform 'package)
   (abbreviation :initform 'BEZIER)
   (comment :initform (list
     "Provide linear, quadratic, and cubic spline bezier curves"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Bezier|
  (progn
    (push '|Bezier| *Packages*)
    (make-instance '|BezierType|)))
```

————-

1.77.4 BezoutMatrix

— defclass BezoutMatrixType —

```
(defclass |BezoutMatrixType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BezoutMatrix")
   (marker :initform 'package)
   (abbreviation :initform 'BEZOUT)
   (comment :initform (list
     "BezoutMatrix contains functions for computing resultants and"
     "discriminants using Bezout matrices."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BezoutMatrix|
  (progn
    (push '|BezoutMatrix| *Packages*)
    (make-instance '|BezoutMatrixType|)))
```

———-

1.77.5 BlowUpPackage

— defclass BlowUpPackageType —

```
(defclass |BlowUpPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BlowUpPackage")
   (marker :initform 'package)
   (abbreviation :initform 'BLUPPACK)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BlowUpPackage|
  (progn
    (push '|BlowUpPackage| *Packages*)
    (make-instance '|BlowUpPackageType|)))
```

———-

1.77.6 BoundIntegerRoots

— defclass BoundIntegerRootsType —

```
(defclass |BoundIntegerRootsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BoundIntegerRoots")
   (marker :initform 'package)
   (abbreviation :initform 'BOUNDZRO)
   (comment :initform (list
     "BoundIntegerRoots provides functions to"
     "find lower bounds on the integer roots of a polynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BoundIntegerRoots|
  (progn
    (push '|BoundIntegerRoots| *Packages*)
    (make-instance '|BoundIntegerRootsType|)))
```

———-

1.77.7 BrillhartTests

— defclass BrillhartTestsType —

```
(defclass |BrillhartTestsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BrillhartTests")
   (marker :initform 'package)
   (abbreviation :initform 'BRILL)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BrillhartTests|
  (progn
    (push '|BrillhartTests| *Packages*)
    (make-instance '|BrillhartTestsType|)))
```

———-

1.78 C

1.78.1 CartesianTensorFunctions2

— defclass CartesianTensorFunctions2Type —

```
(defclass |CartesianTensorFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CartesianTensorFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'CARTEN2)
   (comment :initform (list
     "This package provides functions to enable conversion of tensors"
     "given conversion of the components."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CartesianTensorFunctions2|
  (progn
    (push '|CartesianTensorFunctions2| *Packages*)
    (make-instance '|CartesianTensorFunctions2Type|)))
```

———-

1.78.2 ChangeOfVariable

— defclass ChangeOfVariableType —

```
(defclass |ChangeOfVariableType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ChangeOfVariable")
   (marker :initform 'package)
   (abbreviation :initform 'CHVAR)
   (comment :initform (list
     "Tools to send a point to infinity on an algebraic curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ChangeOfVariable|
  (progn
    (push '|ChangeOfVariable| *Packages*)
    (make-instance '|ChangeOfVariableType|)))
```

———-

1.78.3 CharacteristicPolynomialInMonogenicalAlgebra

— defclass CharacteristicPolynomialInMonogenicalAlgebraType —

```
(defclass |CharacteristicPolynomialInMonogenicalAlgebraType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CharacteristicPolynomialInMonogenicalAlgebra")
   (marker :initform 'package)
   (abbreviation :initform 'CPIMA)
   (comment :initform (list
     "This package implements characteristicPolynomials for monogenic algebras"
     "using resultants"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicPolynomialInMonogenicalAlgebra|
  (progn
    (push '|CharacteristicPolynomialInMonogenicalAlgebra| *Packages*)
    (make-instance '|CharacteristicPolynomialInMonogenicalAlgebraType|)))
```

———-

1.78.4 CharacteristicPolynomialPackage

— defclass CharacteristicPolynomialPackageType —

```
(defclass |CharacteristicPolynomialPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CharacteristicPolynomialPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CHARPOL)
   (comment :initform (list
     "This package provides a characteristicPolynomial function"
     "for any matrix over a commutative ring."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicPolynomialPackage|
  (progn
    (push '|CharacteristicPolynomialPackage| *Packages*)
    (make-instance '|CharacteristicPolynomialPackageType|)))
```

———-

1.78.5 ChineseRemainderToolsForIntegralBases

— defclass ChineseRemainderToolsForIntegralBasesType —

```
(defclass |ChineseRemainderToolsForIntegralBasesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ChineseRemainderToolsForIntegralBases")
   (marker :initform 'package)
   (abbreviation :initform 'IBACHIN)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |ChineseRemainderToolsForIntegralBases|
  (progn
    (push '|ChineseRemainderToolsForIntegralBases| *Packages*)
    (make-instance '|ChineseRemainderToolsForIntegralBasesType|)))
```

———-

1.78.6 CoerceVectorMatrixPackage

— defclass CoerceVectorMatrixPackageType —

```
(defclass |CoerceVectorMatrixPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CoerceVectorMatrixPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CVMP)
   (comment :initform (list
     "CoerceVectorMatrixPackage is an unexposed, technical package"
     "for data conversions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CoerceVectorMatrixPackage|
  (progn
    (push '|CoerceVectorMatrixPackage| *Packages*)
    (make-instance '|CoerceVectorMatrixPackageType|)))
```

———-

1.78.7 CombinatorialFunction

— defclass CombinatorialFunctionType —

```
(defclass |CombinatorialFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CombinatorialFunction")
   (marker :initform 'package)
   (abbreviation :initform 'COMBF)
   (comment :initform (list
     "Provides combinatorial functions over an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CombinatorialFunction|
  (progn
    (push '|CombinatorialFunction| *Packages*)
    (make-instance '|CombinatorialFunctionType|)))
```

———-

1.78.8 CommonDenominator

— defclass CommonDenominatorType —

```
(defclass |CommonDenominatorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CommonDenominator")
   (marker :initform 'package)
   (abbreviation :initform 'CDEN)
   (comment :initform (list
     "CommonDenominator provides functions to compute the"
     "common denominator of a finite linear aggregate of elements of"
     "the quotient field of an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommonDenominator|
  (progn
    (push '|CommonDenominator| *Packages*)
    (make-instance '|CommonDenominatorType|)))
```

———-

1.78.9 CommonOperators

— defclass CommonOperatorsType —

```
(defclass |CommonOperatorsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CommonOperators")
   (marker :initform 'package)
   (abbreviation :initform 'COMMONOP)
   (comment :initform (list
     "This package exports the elementary operators, with some semantics"
     "already attached to them. The semantics that is attached here is not"
     "dependent on the set in which the operators will be applied."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommonOperators|
  (progn
    (push '|CommonOperators| *Packages*)
    (make-instance '|CommonOperatorsType|)))
```

———-

1.78.10 CommuteUnivariatePolynomialCategory

— defclass CommuteUnivariatePolynomialCategoryType —

```
(defclass |CommuteUnivariatePolynomialCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CommuteUnivariatePolynomialCategory")
   (marker :initform 'package)
   (abbreviation :initform 'COMMUPC)
   (comment :initform (list
     "A package for swapping the order of two variables in a tower of two"
     "UnivariatePolynomialCategory extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommuteUnivariatePolynomialCategory|
  (progn
    (push '|CommuteUnivariatePolynomialCategory| *Packages*)
    (make-instance '|CommuteUnivariatePolynomialCategoryType|)))
```

———-

1.78.11 ComplexFactorization

— defclass ComplexFactorizationType —

```
(defclass |ComplexFactorizationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexFactorization")
   (marker :initform 'package)
   (abbreviation :initform 'COMPFACT)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexFactorization|
  (progn
    (push '|ComplexFactorization| *Packages*)
    (make-instance '|ComplexFactorizationType|)))
```

———-

1.78.12 ComplexFunctions2

— defclass ComplexFunctions2Type —

```
(defclass |ComplexFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'COMPLEX2)
   (comment :initform (list
     "This package extends maps from underlying rings to maps between"
     "complex over those rings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexFunctions2|
  (progn
    (push '|ComplexFunctions2| *Packages*)
    (make-instance '|ComplexFunctions2Type|)))
```

———-

1.78.13 ComplexIntegerSolveLinearPolynomialEquation

— defclass ComplexIntegerSolveLinearPolynomialEquationType —

```
(defclass |ComplexIntegerSolveLinearPolynomialEquationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexIntegerSolveLinearPolynomialEquation")
   (marker :initform 'package)
   (abbreviation :initform 'CINTSLPE)
   (comment :initform (list
     "This package provides the generalized euclidean algorithm which is"
     "needed as the basic step for factoring polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexIntegerSolveLinearPolynomialEquation|
  (progn
    (push '|ComplexIntegerSolveLinearPolynomialEquation| *Packages*)
    (make-instance '|ComplexIntegerSolveLinearPolynomialEquationType|)))
```

———-

1.78.14 ComplexPattern

— defclass ComplexPatternType —

```
(defclass |ComplexPatternType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexPattern")
   (marker :initform 'package)
   (abbreviation :initform 'COMPLPAT)
   (comment :initform (list
     "This package supports converting complex expressions to patterns"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexPattern|
  (progn
    (push '|ComplexPattern| *Packages*)
    (make-instance '|ComplexPatternType|)))
```

———-

1.78.15 ComplexPatternMatch

— defclass ComplexPatternMatchType —

```
(defclass |ComplexPatternMatchType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexPatternMatch")
   (marker :initform 'package)
   (abbreviation :initform 'CPMATCH)
   (comment :initform (list
     "This package supports matching patterns involving complex expressions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexPatternMatch|
  (progn
    (push '|ComplexPatternMatch| *Packages*)
    (make-instance '|ComplexPatternMatchType|)))
```

———-

1.78.16 ComplexRootFindingPackage

— defclass ComplexRootFindingPackageType —

```
(defclass |ComplexRootFindingPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexRootFindingPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CRFP)
   (comment :initform (list
     "ComplexRootFindingPackage provides functions to"
     "find all roots of a polynomial p over the complex number by"
     "using Plesken's idea to calculate in the polynomial ring"
     "modulo f and employing the Chinese Remainder Theorem."
     "In this first version, the precision (see digits)"
     "is not increased when this is necessary to"
     "avoid rounding errors. Hence it is the user's responsibility to"
     "increase the precision if necessary."
     "Note also, if this package is called with, for example, Fraction Integer,"
     "the precise calculations could require a lot of time."
     "Also note that evaluating the zeros is not necessarily a good check"
     "whether the result is correct: already evaluation can cause"
     "rounding errors."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexRootFindingPackage|
  (progn
    (push '|ComplexRootFindingPackage| *Packages*)
    (make-instance '|ComplexRootFindingPackageType|)))
```

——————-

1.78.17 ComplexRootPackage

— defclass ComplexRootPackageType —

```
(defclass |ComplexRootPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexRootPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CMPLXRT)
   (comment :initform (list
     "This package provides functions complexZeros"
     "for finding the complex zeros"
     "of univariate polynomials with complex rational number coefficients."
     "The results are to any user specified precision and are returned"
     "as either complex rational number or complex floating point numbers"
     "depending on the type of the second argument which specifies the"
     "precision."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexRootPackage|
  (progn
    (push '|ComplexRootPackage| *Packages*)
    (make-instance '|ComplexRootPackageType|)))
```

———-

1.78.18 ComplexTrigonometricManipulations

— defclass ComplexTrigonometricManipulationsType —

```
(defclass |ComplexTrigonometricManipulationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexTrigonometricManipulations")
   (marker :initform 'package)
   (abbreviation :initform 'CTRIGMNP)
   (comment :initform (list
     "ComplexTrigonometricManipulations provides function that"
     "compute the real and imaginary parts of complex functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexTrigonometricManipulations|
  (progn
    (push '|ComplexTrigonometricManipulations| *Packages*)
    (make-instance '|ComplexTrigonometricManipulationsType|)))
```

———-

1.78.19 ConstantLODE

— defclass ConstantLODEType —

```
(defclass |ConstantLODEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ConstantLODE")
   (marker :initform 'package)
   (abbreviation :initform 'ODECONST)
   (comment :initform (list
     "Solution of linear ordinary differential equations,"
     "constant coefficient case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ConstantLODE|
  (progn
    (push '|ConstantLODE| *Packages*)
    (make-instance '|ConstantLODEType|)))
```

———-

1.78.20 CoordinateSystems

— defclass CoordinateSystemsType —

```
(defclass |CoordinateSystemsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CoordinateSystems")
   (marker :initform 'package)
   (abbreviation :initform 'COORDSYS)
   (comment :initform (list
     "CoordinateSystems provides coordinate transformation functions"
     "for plotting.  Functions in this package return conversion functions"
     "which take points expressed in other coordinate systems and return points"
     "with the corresponding Cartesian coordinates."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CoordinateSystems|
  (progn
    (push '|CoordinateSystems| *Packages*)
    (make-instance '|CoordinateSystemsType|)))
```

———-

1.78.21 CRApackage

— defclass CRApackageType —

```
(defclass |CRApackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CRApackage")
   (marker :initform 'package)
   (abbreviation :initform 'CRAPACK)
   (comment :initform (list
     "This package has no documentation"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CRApackage|
  (progn
    (push '|CRApackage| *Packages*)
    (make-instance '|CRApackageType|)))
```

———-

1.78.22 CycleIndicators

— defclass CycleIndicatorsType —

```
(defclass |CycleIndicatorsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CycleIndicators")
   (marker :initform 'package)
   (abbreviation :initform 'CYCLES)
   (comment :initform (list
     "Polya-Redfield enumeration by cycle indices."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CycleIndicators|
  (progn
    (push '|CycleIndicators| *Packages*)
    (make-instance '|CycleIndicatorsType|)))
```

———-

1.78.23 CyclicStreamTools

— defclass CyclicStreamToolsType —

```
(defclass |CyclicStreamToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CyclicStreamTools")
   (marker :initform 'package)
   (abbreviation :initform 'CSTTOOLS)
   (comment :initform (list
     "This package provides tools for working with cyclic streams."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CyclicStreamTools|
  (progn
    (push '|CyclicStreamTools| *Packages*)
    (make-instance '|CyclicStreamToolsType|)))
```

———-

1.78.24 CyclotomicPolynomialPackage

— defclass CyclotomicPolynomialPackageType —

```
(defclass |CyclotomicPolynomialPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CyclotomicPolynomialPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CYCLOTOM)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CyclotomicPolynomialPackage|
  (progn
    (push '|CyclotomicPolynomialPackage| *Packages*)
    (make-instance '|CyclotomicPolynomialPackageType|)))
```

———-

1.78.25 CylindricalAlgebraicDecompositionPackage

— defclass CylindricalAlgebraicDecompositionPackageType —

```
(defclass |CylindricalAlgebraicDecompositionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CylindricalAlgebraicDecompositionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CAD)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CylindricalAlgebraicDecompositionPackage|
  (progn
    (push '|CylindricalAlgebraicDecompositionPackage| *Packages*)
    (make-instance '|CylindricalAlgebraicDecompositionPackageType|)))
```

———-

1.78.26 CylindricalAlgebraicDecompositionUtilities

— defclass CylindricalAlgebraicDecompositionUtilitiesType —

```
(defclass |CylindricalAlgebraicDecompositionUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CylindricalAlgebraicDecompositionUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'CADU)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CylindricalAlgebraicDecompositionUtilities|
  (progn
    (push '|CylindricalAlgebraicDecompositionUtilities| *Packages*)
    (make-instance '|CylindricalAlgebraicDecompositionUtilitiesType|)))
```

———-

1.79 D

1.79.1 DefiniteIntegrationTools

— defclass DefiniteIntegrationToolsType —

```
(defclass |DefiniteIntegrationToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DefiniteIntegrationTools")
   (marker :initform 'package)
   (abbreviation :initform 'DFINTTLS)
   (comment :initform (list
     "DefiniteIntegrationTools provides common tools used"
     "by the definite integration of both rational and elementary functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DefiniteIntegrationTools|
  (progn
    (push '|DefiniteIntegrationTools| *Packages*)
    (make-instance '|DefiniteIntegrationToolsType|)))
```

———-

1.79.2 DegreeReductionPackage

— defclass DegreeReductionPackageType —

```
(defclass |DegreeReductionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DegreeReductionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'DEGRED)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DegreeReductionPackage|
  (progn
    (push '|DegreeReductionPackage| *Packages*)
    (make-instance '|DegreeReductionPackageType|)))
```

———-

1.79.3 DesingTreePackage

— defclass DesingTreePackageType —

```
(defclass |DesingTreePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DesingTreePackage")
   (marker :initform 'package)
   (abbreviation :initform 'DTP)
   (comment :initform (list
     "The following is all the categories, domains and package"
     "used for the desingularisation be means of"
     "monoidal transformation (Blowing-up)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DesingTreePackage|
  (progn
    (push '|DesingTreePackage| *Packages*)
    (make-instance '|DesingTreePackageType|)))
```

———-

1.79.4 DiophantineSolutionPackage

— defclass DiophantineSolutionPackageType —

```
(defclass |DiophantineSolutionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DiophantineSolutionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'DIOSP)
   (comment :initform (list
     "Any solution of a homogeneous linear Diophantine equation"
     "can be represented as a sum of minimal solutions, which"
     "form a 'basis' (a minimal solution cannot be represented"
     "as a nontrivial sum of solutions)"
     "in the case of an inhomogeneous linear Diophantine equation,"
     "each solution is the sum of a inhomogeneous solution and"
     "any number of homogeneous solutions"
     "therefore, it suffices to compute two sets:"
     "1. all minimal inhomogeneous solutions"
     "2. all minimal homogeneous solutions"
     "the algorithm implemented is a completion procedure, which"
     "enumerates all solutions in a recursive depth-first-search"
     "it can be seen as finding monotone paths in a graph"
     "for more details see Reference"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DiophantineSolutionPackage|
  (progn
    (push '|DiophantineSolutionPackage| *Packages*)
    (make-instance '|DiophantineSolutionPackageType|)))
```

———-

1.79.5 DirectProductFunctions2

— defclass DirectProductFunctions2Type —

```
(defclass |DirectProductFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DirectProductFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'DIRPROD2)
   (comment :initform (list
     "This package provides operations which all take as arguments direct"
     "products of elements of some type A and functions from A"
     "to another type B. The operations all iterate over their vector argument"
     "and either return a value of type B or a direct product over B."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductFunctions2|
  (progn
    (push '|DirectProductFunctions2| *Packages*)
    (make-instance '|DirectProductFunctions2Type|)))
```

———-

1.79.6 DiscreteLogarithmPackage

— defclass DiscreteLogarithmPackageType —

```
(defclass |DiscreteLogarithmPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DiscreteLogarithmPackage")
   (marker :initform 'package)
   (abbreviation :initform 'DLP)
   (comment :initform (list
     "DiscreteLogarithmPackage implements help functions for discrete logarithms"
     "in monoids using small cyclic groups."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DiscreteLogarithmPackage|
  (progn
    (push '|DiscreteLogarithmPackage| *Packages*)
    (make-instance '|DiscreteLogarithmPackageType|)))
```

———-

1.79.7 DisplayPackage

— defclass DisplayPackageType —

```
(defclass |DisplayPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DisplayPackage")
   (marker :initform 'package)
   (abbreviation :initform 'DISPLAY)
   (comment :initform (list
     "DisplayPackage allows one to print strings in a nice manner,"
     "including highlighting substrings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DisplayPackage|
  (progn
    (push '|DisplayPackage| *Packages*)
    (make-instance '|DisplayPackageType|)))
```

———-

1.79.8 DistinctDegreeFactorize

— defclass DistinctDegreeFactorizeType —

```
(defclass |DistinctDegreeFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DistinctDegreeFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'DDFACT)
   (comment :initform (list
     "Package for the factorization of a univariate polynomial with"
     "coefficients in a finite field. The algorithm used is the"
     "'distinct degree' algorithm of Cantor-Zassenhaus, modified"
     "to use trace instead of the norm and a table for computing"
     "Frobenius as suggested by Naudin and Quitte."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DistinctDegreeFactorize|
  (progn
    (push '|DistinctDegreeFactorize| *Packages*)
    (make-instance '|DistinctDegreeFactorizeType|)))
```

————-

1.79.9 DoubleFloatSpecialFunctions

— defclass DoubleFloatSpecialFunctionsType —

```
(defclass |DoubleFloatSpecialFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DoubleFloatSpecialFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'DFSFUN)
   (comment :initform (list
     "This package provides special functions for double precision"
     "real and complex floating point."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleFloatSpecialFunctions|
  (progn
    (push '|DoubleFloatSpecialFunctions| *Packages*)
    (make-instance '|DoubleFloatSpecialFunctionsType|)))
```

————-

1.79.10 DoubleResultantPackage

— defclass DoubleResultantPackageType —

```
(defclass |DoubleResultantPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DoubleResultantPackage")
   (marker :initform 'package)
   (abbreviation :initform 'DBLRESP)
   (comment :initform (list
     "This package provides functions for computing the residues"
     "of a function on an algebraic curve."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleResultantPackage|
  (progn
    (push '|DoubleResultantPackage| *Packages*)
    (make-instance '|DoubleResultantPackageType|)))
```

————-

1.79.11 DrawComplex

— defclass DrawComplexType —

```
(defclass |DrawComplexType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DrawComplex")
   (marker :initform 'package)
   (abbreviation :initform 'DRAWCX)
   (comment :initform (list
     "DrawComplex provides some facilities"
     "for drawing complex functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawComplex|
  (progn
    (push '|DrawComplex| *Packages*)
    (make-instance '|DrawComplexType|)))
```

————-

1.79.12 DrawNumericHack

— defclass DrawNumericHackType —

```
(defclass |DrawNumericHackType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DrawNumericHack")
   (marker :initform 'package)
   (abbreviation :initform 'DRAWHACK)
   (comment :initform (list
     "Hack for the draw interface. DrawNumericHack provides"
     "a 'coercion' from something of the form x = a..b where a"
     "and b are"
     "formal expressions to a binding of the form x = c..d where c and d"
     "are the numerical values of a and b. This 'coercion' fails if"
     "a and b contains symbolic variables, but is meant for expressions"
     "involving %pi."
     "Note that this package is meant for internal use only."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawNumericHack|
  (progn
    (push '|DrawNumericHack| *Packages*)
    (make-instance '|DrawNumericHackType|)))
```

———-

1.79.13 DrawOptionFunctions0

— defclass DrawOptionFunctions0Type —

```
(defclass |DrawOptionFunctions0Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DrawOptionFunctions0")
   (marker :initform 'package)
   (abbreviation :initform 'DROPT0)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawOptionFunctions0|
  (progn
    (push '|DrawOptionFunctions0| *Packages*)
    (make-instance '|DrawOptionFunctions0Type|)))
```

———-

1.79.14 DrawOptionFunctions1

— defclass DrawOptionFunctions1Type —

```
(defclass |DrawOptionFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DrawOptionFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'DROPT1)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawOptionFunctions1|
  (progn
    (push '|DrawOptionFunctions1| *Packages*)
    (make-instance '|DrawOptionFunctions1Type|)))
```

————-

1.79.15 d01AgentsPackage

— defclass d01AgentsPackageType —

```
(defclass |d01AgentsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "d01AgentsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'D01AGNT)
   (comment :initform (list
     "d01AgentsPackage is a package of numerical agents to be used"
     "to investigate attributes of an input function so as to decide the"
     "measure of an appropriate numerical integration routine."
     "It contains functions rangeIsFinite to test the input range and"
     "functionIsContinuousAtEndPoints to check for continuity at"
     "the end points of the range."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01AgentsPackage|
  (progn
    (push '|d01AgentsPackage| *Packages*)
    (make-instance '|d01AgentsPackageType|)))
```

————-

1.79.16 d01WeightsPackage

— defclass d01WeightsPackageType —

```
(defclass |d01WeightsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "d01WeightsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'D01WGTS)
   (comment :initform (list
     "d01WeightsPackage is a package for functions used to investigate"
     "whether a function can be divided into a simpler function and a weight"
     "function.  The types of weights investigated are those giving rise to"
     "end-point singularities of the algebraico-logarithmic type, and"
     "trigonometric weights."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01WeightsPackage|
  (progn
    (push '|d01WeightsPackage| *Packages*)
    (make-instance '|d01WeightsPackageType|)))
```

————-

1.79.17 d02AgentsPackage

— defclass d02AgentsPackageType —

```
(defclass |d02AgentsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "d02AgentsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'D02AGNT)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02AgentsPackage|
  (progn
    (push '|d02AgentsPackage| *Packages*)
    (make-instance '|d02AgentsPackageType|)))
```

————-

1.79.18 d03AgentsPackage

— defclass d03AgentsPackageType —

```
(defclass |d03AgentsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "d03AgentsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'D03AGNT)
   (comment :initform (list
     "d03AgentsPackage contains a set of computational agents"
     "for use with Partial Differential Equation solvers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d03AgentsPackage|
  (progn
    (push '|d03AgentsPackage| *Packages*)
    (make-instance '|d03AgentsPackageType|)))
```

———–

1.80 E

1.80.1 EigenPackage

— defclass EigenPackageType —

```
(defclass |EigenPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "EigenPackage")
   (marker :initform 'package)
   (abbreviation :initform 'EP)
   (comment :initform (list
     "This is a package for the exact computation of eigenvalues and eigenvectors."
     "This package can be made to work for matrices with coefficients which are"
     "rational functions over a ring where we can factor polynomials."
     "Rational eigenvalues are always explicitly computed while the"
     "non-rational ones are expressed in terms of their minimal polynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EigenPackage|
  (progn
    (push '|EigenPackage| *Packages*)
    (make-instance '|EigenPackageType|)))
```

———-

1.80.2 ElementaryFunction

— defclass ElementaryFunctionType —

```
(defclass |ElementaryFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunction")
   (marker :initform 'package)
   (abbreviation :initform 'EF)
   (comment :initform (list
     "Provides elementary functions over an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunction|
  (progn
    (push '|ElementaryFunction| *Packages*)
    (make-instance '|ElementaryFunctionType|)))
```

———-

1.80.3 ElementaryFunctionDefiniteIntegration

— defclass ElementaryFunctionDefiniteIntegrationType —

```
(defclass |ElementaryFunctionDefiniteIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunctionDefiniteIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'DEFINTEF)
   (comment :initform (list
     "RationalFunctionDefiniteIntegration provides functions to"
     "compute definite integrals of rational functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionDefiniteIntegration|
  (progn
    (push '|ElementaryFunctionDefiniteIntegration| *Packages*)
    (make-instance '|ElementaryFunctionDefiniteIntegrationType|)))
```

———-

1.80.4 ElementaryFunctionLODESolver

— defclass ElementaryFunctionLODESolverType —

```
(defclass |ElementaryFunctionLODESolverType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunctionLODESolver")
   (marker :initform 'package)
   (abbreviation :initform 'LODEEF)
   (comment :initform (list
     "ElementaryFunctionLODESolver provides the top-level"
     "functions for finding closed form solutions of linear ordinary"
     "differential equations and initial value problems."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionLODESolver|
  (progn
    (push '|ElementaryFunctionLODESolver| *Packages*)
    (make-instance '|ElementaryFunctionLODESolverType|)))
```

———-

1.80.5 ElementaryFunctionODESolver

— defclass ElementaryFunctionODESolverType —

```
(defclass |ElementaryFunctionODESolverType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunctionODESolver")
   (marker :initform 'package)
   (abbreviation :initform 'ODEEF)
   (comment :initform (list
     "ElementaryFunctionODESolver provides the top-level"
     "functions for finding closed form solutions of ordinary"
     "differential equations and initial value problems."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionODESolver|
  (progn
    (push '|ElementaryFunctionODESolver| *Packages*)
    (make-instance '|ElementaryFunctionODESolverType|)))
```

———-

1.80.6 ElementaryFunctionSign

— defclass ElementaryFunctionSignType —

```
(defclass |ElementaryFunctionSignType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunctionSign")
   (marker :initform 'package)
   (abbreviation :initform 'SIGNEF)
   (comment :initform (list
     "This package provides functions to determine the sign of an"
     "elementary function around a point or infinity."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionSign|
  (progn
    (push '|ElementaryFunctionSign| *Packages*)
    (make-instance '|ElementaryFunctionSignType|)))
```

———-

1.80.7 ElementaryFunctionStructurePackage

— defclass ElementaryFunctionStructurePackageType —

```
(defclass |ElementaryFunctionStructurePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunctionStructurePackage")
   (marker :initform 'package)
   (abbreviation :initform 'EFSTRUC)
   (comment :initform (list
     "ElementaryFunctionStructurePackage provides functions to test the"
     "algebraic independence of various elementary functions, using the"
     "Risch structure theorem (real and complex versions)."
     "It also provides transformations on elementary functions"
     "which are not considered simplifications."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionStructurePackage|
  (progn
    (push '|ElementaryFunctionStructurePackage| *Packages*)
    (make-instance '|ElementaryFunctionStructurePackageType|)))
```

———-

1.80.8 ElementaryIntegration

— defclass ElementaryIntegrationType —

```
(defclass |ElementaryIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTEF)
   (comment :initform (list
     "This package provides functions for integration, limited integration,"
     "extended integration and the risch differential equation for"
     "elementary functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryIntegration|
  (progn
    (push '|ElementaryIntegration| *Packages*)
    (make-instance '|ElementaryIntegrationType|)))
```

———-

1.80.9 ElementaryRischDE

— defclass ElementaryRischDEType —

```
(defclass |ElementaryRischDEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryRischDE")
   (marker :initform 'package)
   (abbreviation :initform 'RDEEF)
   (comment :initform (list
     "Risch differential equation, elementary case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryRischDE|
  (progn
    (push '|ElementaryRischDE| *Packages*)
    (make-instance '|ElementaryRischDEType|)))
```

———-

1.80.10 ElementaryRischDESystem

— defclass ElementaryRischDESystemType —

```
(defclass |ElementaryRischDESystemType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryRischDESystem")
   (marker :initform 'package)
   (abbreviation :initform 'RDEEFS)
   (comment :initform (list
     "Risch differential equation, elementary case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryRischDESystem|
  (progn
    (push '|ElementaryRischDESystem| *Packages*)
    (make-instance '|ElementaryRischDESystemType|)))
```

———-

1.80.11 EllipticFunctionsUnivariateTaylorSeries

— defclass EllipticFunctionsUnivariateTaylorSeriesType —

```
(defclass |EllipticFunctionsUnivariateTaylorSeriesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "EllipticFunctionsUnivariateTaylorSeries")
   (marker :initform 'package)
   (abbreviation :initform 'ELFUTS)
   (comment :initform (list
     "The elliptic functions sn, sc and dn are expanded as Taylor series."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EllipticFunctionsUnivariateTaylorSeries|
  (progn
    (push '|EllipticFunctionsUnivariateTaylorSeries| *Packages*)
    (make-instance '|EllipticFunctionsUnivariateTaylorSeriesType|)))
```

———-

1.80.12 EquationFunctions2

— defclass EquationFunctions2Type —

```
(defclass |EquationFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "EquationFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'EQ2)
   (comment :initform (list
     "This package provides operations for mapping the sides of equations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EquationFunctions2|
  (progn
    (push '|EquationFunctions2| *Packages*)
    (make-instance '|EquationFunctions2Type|)))
```

———-

1.80.13 ErrorFunctions

— defclass ErrorFunctionsType —

```
(defclass |ErrorFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ErrorFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'ERROR)
   (comment :initform (list
     "ErrorFunctions implements error functions callable from the system"
     "interpreter.  Typically, these functions would be called in user"
     "functions.  The simple forms of the functions take one argument"
     "which is either a string (an error message) or a list of strings"
     "which all together make up a message.  The list can contain"
     "formatting codes (see below).  The more sophisticated versions takes"
     "two arguments where the first argument is the name of the function"
     "from which the error was invoked and the second argument is either a"
     "string or a list of strings, as above.  When you use the one"
     "argument version in an interpreter function, the system will"
     "automatically insert the name of the function as the new first"
     "argument.  Thus in the user interpreter function"
     "     f x == if x < 0 then error 'negative argument' else x"
     "the call to error will actually be of the form"
     "     error('f','negative argument')"
     "because the interpreter will have created a new first argument."
     " "
     "Formatting codes:  error messages may contain the following"
     "formatting codes (they should either start or end a string or"
     "else have blanks around them):"
     "%l      start a new line"
     "%ceon   start centering message lines"
     "%ceoff  stop  centering message lines"
     "%rjon   start displaying lines 'ragged left'"
     "%rjoff  stop  displaying lines 'ragged left'"
     "%i      indent   following lines 3 additional spaces"
     "%u      unindent following lines 3 additional spaces"
     "%xN    insert N blanks (eg, %x10 inserts 10 blanks)"
     " "
     "Examples:"
     "1.error 'Whoops, you made a %l %ceon big %ceoff %l mistake!'"
     "2.error ['Whoops, you made a','%l %ceon ','big',"
     "          '%d %ceoff %l','mistake!']"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ErrorFunctions|
  (progn
    (push '|ErrorFunctions| *Packages*)
    (make-instance '|ErrorFunctionsType|)))
```

———-

1.80.14 EuclideanGroebnerBasisPackage

— defclass EuclideanGroebnerBasisPackageType —

```
(defclass |EuclideanGroebnerBasisPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "EuclideanGroebnerBasisPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GBEUCLID)
   (comment :initform (list
     "EuclideanGroebnerBasisPackage computes groebner"
     "bases for polynomial ideals over euclidean domains."
     "The basic computation provides"
     "a distinguished set of generators for these ideals."
     "This basis allows an easy test for membership: the operation"
     "euclideanNormalForm returns zero on ideal members. The string"
     "'info' and 'redcrit' can be given as additional args to provide"
     "incremental information during the computation. If 'info' is given,"
     "a computational summary is given for each s-polynomial. If 'redcrit'"
     "is given, the reduced critical pairs are printed. The term ordering"
     "is determined by the polynomial type used. Suggested types include"
     "DistributedMultivariatePolynomial,"
     "HomogeneousDistributedMultivariatePolynomial,"
     "GeneralDistributedMultivariatePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EuclideanGroebnerBasisPackage|
  (progn
    (push '|EuclideanGroebnerBasisPackage| *Packages*)
    (make-instance '|EuclideanGroebnerBasisPackageType|)))
```

———-

1.80.15 EvaluateCycleIndicators

— defclass EvaluateCycleIndicatorsType —

```
(defclass |EvaluateCycleIndicatorsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "EvaluateCycleIndicators")
   (marker :initform 'package)
   (abbreviation :initform 'EVALCYC)
   (comment :initform (list
     "This package is to be used in conjuction with the CycleIndicators package."
     "It provides an evaluation function for SymmetricPolynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EvaluateCycleIndicators|
  (progn
    (push '|EvaluateCycleIndicators| *Packages*)
    (make-instance '|EvaluateCycleIndicatorsType|)))
```

———-

1.80.16 ExpertSystemContinuityPackage

— defclass ExpertSystemContinuityPackageType —

```
(defclass |ExpertSystemContinuityPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpertSystemContinuityPackage")
   (marker :initform 'package)
   (abbreviation :initform 'ESCONT)
   (comment :initform (list
     "ExpertSystemContinuityPackage is a package of functions for the use of"
     "domains belonging to the category NumericalIntegration."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemContinuityPackage|
  (progn
    (push '|ExpertSystemContinuityPackage| *Packages*)
    (make-instance '|ExpertSystemContinuityPackageType|)))
```

———-

1.80.17 ExpertSystemContinuityPackage1

— defclass ExpertSystemContinuityPackage1Type —

```
(defclass |ExpertSystemContinuityPackage1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpertSystemContinuityPackage1")
   (marker :initform 'package)
   (abbreviation :initform 'ESCONT1)
   (comment :initform (list
     "ExpertSystemContinuityPackage1 exports a function to check range inclusion"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemContinuityPackage1|
  (progn
    (push '|ExpertSystemContinuityPackage1| *Packages*)
    (make-instance '|ExpertSystemContinuityPackage1Type|)))
```

———-

1.80.18 ExpertSystemToolsPackage

— defclass ExpertSystemToolsPackageType —

```
(defclass |ExpertSystemToolsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpertSystemToolsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'ESTOOLS)
   (comment :initform (list
     "ExpertSystemToolsPackage contains some useful functions for use"
     "by the computational agents of numerical solvers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemToolsPackage|
  (progn
    (push '|ExpertSystemToolsPackage| *Packages*)
    (make-instance '|ExpertSystemToolsPackageType|)))
```

———-

1.80.19 ExpertSystemToolsPackage1

— defclass ExpertSystemToolsPackage1Type —

```
(defclass |ExpertSystemToolsPackage1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpertSystemToolsPackage1")
   (marker :initform 'package)
   (abbreviation :initform 'ESTOOLS1)
   (comment :initform (list
     "ExpertSystemToolsPackage1 contains some useful functions for use"
     "by the computational agents of Ordinary Differential Equation solvers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemToolsPackage1|
  (progn
    (push '|ExpertSystemToolsPackage1| *Packages*)
    (make-instance '|ExpertSystemToolsPackage1Type|)))
```

———-

1.80.20 ExpertSystemToolsPackage2

— defclass ExpertSystemToolsPackage2Type —

```
(defclass |ExpertSystemToolsPackage2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpertSystemToolsPackage2")
   (marker :initform 'package)
   (abbreviation :initform 'ESTOOLS2)
   (comment :initform (list
     "ExpertSystemToolsPackage2 contains some useful functions for use"
     "by the computational agents of Ordinary Differential Equation solvers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemToolsPackage2|
  (progn
    (push '|ExpertSystemToolsPackage2| *Packages*)
    (make-instance '|ExpertSystemToolsPackage2Type|)))
```

———-

1.80.21 ExpressionFunctions2

— defclass ExpressionFunctions2Type —

```
(defclass |ExpressionFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'EXPR2)
   (comment :initform (list
     "Lifting of maps to Expressions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionFunctions2|
  (progn
    (push '|ExpressionFunctions2| *Packages*)
    (make-instance '|ExpressionFunctions2Type|)))
```

———-

1.80.22 ExpressionSolve

— defclass ExpressionSolveType —

```
(defclass |ExpressionSolveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionSolve")
   (marker :initform 'package)
   (abbreviation :initform 'EXPRSOL)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSolve|
  (progn
    (push '|ExpressionSolve| *Packages*)
    (make-instance '|ExpressionSolveType|)))
```

———-

1.80.23 ExpressionSpaceFunctions1

— defclass ExpressionSpaceFunctions1Type —

```
(defclass |ExpressionSpaceFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionSpaceFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'ES1)
   (comment :initform (list
     "This package allows a map from any expression space into any object"
     "to be lifted to a kernel over the expression set, using a given"
     "property of the operator of the kernel."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpaceFunctions1|
  (progn
    (push '|ExpressionSpaceFunctions1| *Packages*)
    (make-instance '|ExpressionSpaceFunctions1Type|)))
```

———-

1.80.24 ExpressionSpaceFunctions2

— defclass ExpressionSpaceFunctions2Type —

```
(defclass |ExpressionSpaceFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionSpaceFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ES2)
   (comment :initform (list
     "This package allows a mapping E -> F to be lifted to a kernel over E"
     "This lifting can fail if the operator of the kernel cannot be applied"
     "in F; Do not use this package with E = F, since this may"
     "drop some properties of the operators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpaceFunctions2|
  (progn
    (push '|ExpressionSpaceFunctions2| *Packages*)
    (make-instance '|ExpressionSpaceFunctions2Type|)))
```

———-

1.80.25 ExpressionSpaceODESolver

— defclass ExpressionSpaceODESolverType —

```
(defclass |ExpressionSpaceODESolverType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionSpaceODESolver")
   (marker :initform 'package)
   (abbreviation :initform 'EXPRODE)
   (comment :initform (list
     "Taylor series solutions of explicit ODE's"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpaceODESolver|
  (progn
    (push '|ExpressionSpaceODESolver| *Packages*)
    (make-instance '|ExpressionSpaceODESolverType|)))
```

———-

1.80.26 ExpressionToOpenMath

— defclass ExpressionToOpenMathType —

```
(defclass |ExpressionToOpenMathType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionToOpenMath")
   (marker :initform 'package)
   (abbreviation :initform 'OMEXPR)
   (comment :initform (list
     "ExpressionToOpenMath provides support for"
     "converting objects of type Expression into OpenMath."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionToOpenMath|
  (progn
    (push '|ExpressionToOpenMath| *Packages*)
    (make-instance '|ExpressionToOpenMathType|)))
```

———-

1.80.27 ExpressionToUnivariatePowerSeries

— defclass ExpressionToUnivariatePowerSeriesType —

```
(defclass |ExpressionToUnivariatePowerSeriesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionToUnivariatePowerSeries")
   (marker :initform 'package)
   (abbreviation :initform 'EXPR2UPS)
   (comment :initform (list
     "This package provides functions to convert functional expressions"
     "to power series."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionToUnivariatePowerSeries|
  (progn
    (push '|ExpressionToUnivariatePowerSeries| *Packages*)
    (make-instance '|ExpressionToUnivariatePowerSeriesType|)))
```

———-

1.80.28 ExpressionTubePlot

— defclass ExpressionTubePlotType —

```
(defclass |ExpressionTubePlotType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionTubePlot")
   (marker :initform 'package)
   (abbreviation :initform 'EXPRTUBE)
   (comment :initform (list
     "Package for constructing tubes around 3-dimensional parametric curves."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionTubePlot|
  (progn
    (push '|ExpressionTubePlot| *Packages*)
    (make-instance '|ExpressionTubePlotType|)))
```

———-

1.80.29 Export3D

— defclass Export3DType —

```
(defclass |Export3DType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Export3D")
   (marker :initform 'package)
   (abbreviation :initform 'EXP3D)
   (comment :initform (list
     "This package provides support for exporting SubSpace and"
     "ThreeSpace structures to files."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Export3D|
  (progn
    (push '|Export3D| *Packages*)
    (make-instance '|Export3DType|)))
```

———-

1.80.30 e04AgentsPackage

— defclass e04AgentsPackageType —

```
(defclass |e04AgentsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "e04AgentsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'E04AGNT)
   (comment :initform (list
     "e04AgentsPackage is a package of numerical agents to be used"
     "to investigate attributes of an input function so as to decide the"
     "measure of an appropriate numerical optimization routine."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04AgentsPackage|
  (progn
    (push '|e04AgentsPackage| *Packages*)
    (make-instance '|e04AgentsPackageType|)))
```

———-

1.81 F

1.81.1 FactoredFunctions

— defclass FactoredFunctionsType —

```
(defclass |FactoredFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FactoredFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'FACTFUNC)
   (comment :initform (list
     "Computes various functions on factored arguments."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoredFunctions|
  (progn
    (push '|FactoredFunctions| *Packages*)
    (make-instance '|FactoredFunctionsType|)))
```

——————-

1.81.2 FactoredFunctions2

— defclass FactoredFunctions2Type —

```
(defclass |FactoredFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FactoredFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FR2)
   (comment :initform (list
     "FactoredFunctions2 contains functions that involve"
     "factored objects whose underlying domains may not be the same."
     "For example, map might be used to coerce an object of"
     "type Factored(Integer) to Factored(Complex(Integer))."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoredFunctions2|
  (progn
    (push '|FactoredFunctions2| *Packages*)
    (make-instance '|FactoredFunctions2Type|)))
```

——————-

1.81.3 FactoredFunctionUtilities

— defclass FactoredFunctionUtilitiesType —

```
(defclass |FactoredFunctionUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FactoredFunctionUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'FRUTIL)
   (comment :initform (list
     "FactoredFunctionUtilities implements some utility"
     "functions for manipulating factored objects."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoredFunctionUtilities|
  (progn
    (push '|FactoredFunctionUtilities| *Packages*)
    (make-instance '|FactoredFunctionUtilitiesType|)))
```

———-

1.81.4 FactoringUtilities

— defclass FactoringUtilitiesType —

```
(defclass |FactoringUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FactoringUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'FACUTIL)
   (comment :initform (list
     "This package provides utilities used by the factorizers"
     "which operate on polynomials represented as univariate polynomials"
     "with multivariate coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoringUtilities|
  (progn
    (push '|FactoringUtilities| *Packages*)
    (make-instance '|FactoringUtilitiesType|)))
```

———-

1.81.5 FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

— defclass FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumberType —

```
(defclass |FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumberType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber")
   (marker :initform 'package)
   (abbreviation :initform 'FACTEXT)
   (comment :initform (list
     "Part of the Package for Algebraic Function Fields in one variable PAFF"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber|
  (progn
    (push '|FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber| *Packages*)
    (make-instance '|FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumberType|)))
```

———-

1.81.6 FactorisationOverPseudoAlgebraicClosureOfRationalNumber

— defclass FactorisationOverPseudoAlgebraicClosureOfRationalNumberType —

```
(defclass |FactorisationOverPseudoAlgebraicClosureOfRationalNumberType| (|AxiomClass|)
 ((parents :initform ())
  (name :initform "FactorisationOverPseudoAlgebraicClosureOfRationalNumber")
  (marker :initform 'package)
  (abbreviation :initform 'FACTRN)
  (comment :initform (list
    "Part of the Package for Algebraic Function Fields in one variable PAFF"))
  (argslist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |FactorisationOverPseudoAlgebraicClosureOfRationalNumber|
 (progn
   (push '|FactorisationOverPseudoAlgebraicClosureOfRationalNumber| *Packages*)
   (make-instance '|FactorisationOverPseudoAlgebraicClosureOfRationalNumberType|)))
```

———-

1.81.7 FGLMIfCanPackage

— defclass FGLMIfCanPackageType —

```
(defclass |FGLMIfCanPackageType| (|AxiomClass|)
 ((parents :initform ())
  (name :initform "FGLMIfCanPackage")
  (marker :initform 'package)
  (abbreviation :initform 'FGLMICPK)
  (comment :initform (list
    "This is just an interface between several packages and domains."
    "The goal is to compute lexicographical Groebner bases"
    "of sets of polynomial with type Polynomial R"
    "by the FGLM algorithm if this is possible"
    "(if the input system generates a zero-dimensional ideal)."))
  (argslist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
```

```
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FGLMIfCanPackage|
  (progn
    (push '|FGLMIfCanPackage| *Packages*)
    (make-instance '|FGLMIfCanPackageType|)))
```

———-

1.81.8 FindOrderFinite

— defclass FindOrderFiniteType —

```
(defclass |FindOrderFiniteType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FindOrderFinite")
   (marker :initform 'package)
   (abbreviation :initform 'FORDER)
   (comment :initform (list
     "Finds the order of a divisor over a finite field"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FindOrderFinite|
  (progn
    (push '|FindOrderFinite| *Packages*)
    (make-instance '|FindOrderFiniteType|)))
```

———-

1.81.9 FiniteAbelianMonoidRingFunctions2

— defclass FiniteAbelianMonoidRingFunctions2Type —

```
(defclass |FiniteAbelianMonoidRingFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteAbelianMonoidRingFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FAMR2)
   (comment :initform (list
     "This package provides a mapping function for FiniteAbelianMonoidRing"
     "The packages defined in this file provide fast fraction free rational"
     "interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |FiniteAbelianMonoidRingFunctions2|
  (progn
    (push '|FiniteAbelianMonoidRingFunctions2| *Packages*)
    (make-instance '|FiniteAbelianMonoidRingFunctions2Type|)))
```

———-

1.81.10 FiniteDivisorFunctions2

— defclass FiniteDivisorFunctions2Type —

```
(defclass |FiniteDivisorFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteDivisorFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FDIV2)
   (comment :initform (list
     "Lift a map to finite divisors."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteDivisorFunctions2|
  (progn
    (push '|FiniteDivisorFunctions2| *Packages*)
    (make-instance '|FiniteDivisorFunctions2Type|)))
```

———-

1.81.11 FiniteFieldFactorization

— defclass FiniteFieldFactorizationType —

```
(defclass |FiniteFieldFactorizationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldFactorization")
   (marker :initform 'package)
   (abbreviation :initform 'FFFACTOR)
   (comment :initform (list
     "Part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldFactorization|
  (progn
    (push '|FiniteFieldFactorization| *Packages*)
    (make-instance '|FiniteFieldFactorizationType|)))
```

———-

1.81.12 FiniteFieldFactorizationWithSizeParseBySideEffect

— defclass FiniteFieldFactorizationWithSizeParseBySideEffectType —

```
(defclass |FiniteFieldFactorizationWithSizeParseBySideEffectType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldFactorizationWithSizeParseBySideEffect")
   (marker :initform 'package)
   (abbreviation :initform 'FFFACTSE)
   (comment :initform (list
     "Part of the package for Algebraic Function Fields in one variable (PAFF)"
     "It has been modified (very slitely) so that each time the 'factor'"
     "function is used, the variable related to the size of the field"
     "over which the polynomial is factorized is reset. This is done in"
     "order to be used with a 'dynamic extension field' which size is not"
     "fixed but set before calling the 'factor' function and which is"
     "parse by side effect to this package via the function 'size'. See"
     "the local function initialize' of this package."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldFactorizationWithSizeParseBySideEffect|
  (progn
    (push '|FiniteFieldFactorizationWithSizeParseBySideEffect| *Packages*)
    (make-instance '|FiniteFieldFactorizationWithSizeParseBySideEffectType|)))
```

———-

1.81.13 FiniteFieldFunctions

— defclass FiniteFieldFunctionsType —

```
(defclass |FiniteFieldFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'FFF)
   (comment :initform (list
     "FiniteFieldFunctions(GF) is a package with functions"
     "concerning finite extension fields of the finite ground field GF,"
     "for example, Zech logarithms."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldFunctions|
  (progn
    (push '|FiniteFieldFunctions| *Packages*)
    (make-instance '|FiniteFieldFunctionsType|)))
```

———-

1.81.14 FiniteFieldHomomorphisms

— defclass FiniteFieldHomomorphismsType —

```
(defclass |FiniteFieldHomomorphismsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldHomomorphisms")
   (marker :initform 'package)
   (abbreviation :initform 'FFHOM)
   (comment :initform (list
     "FiniteFieldHomomorphisms(F1,GF,F2) exports coercion functions of"
     "elements between the fields F1 and F2, which both must be"
     "finite simple algebraic extensions of the finite ground field GF."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldHomomorphisms|
  (progn
    (push '|FiniteFieldHomomorphisms| *Packages*)
    (make-instance '|FiniteFieldHomomorphismsType|)))
```

———-

1.81.15 FiniteFieldPolynomialPackage

— defclass FiniteFieldPolynomialPackageType —

```
(defclass |FiniteFieldPolynomialPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldPolynomialPackage")
   (marker :initform 'package)
   (abbreviation :initform 'FFPOLY)
   (comment :initform (list
     "This package provides a number of functions for generating, counting"
     "and testing irreducible, normal, primitive, random polynomials"
     "over finite fields."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
  (addlist :initform nil)))

(defvar |FiniteFieldPolynomialPackage|
  (progn
    (push '|FiniteFieldPolynomialPackage| *Packages*)
    (make-instance '|FiniteFieldPolynomialPackageType|)))
```

———-

1.81.16 FiniteFieldPolynomialPackage2

— defclass FiniteFieldPolynomialPackage2Type —

```
(defclass |FiniteFieldPolynomialPackage2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldPolynomialPackage2")
   (marker :initform 'package)
   (abbreviation :initform 'FFPOLY2)
   (comment :initform (list
     "FiniteFieldPolynomialPackage2(F,GF) exports some functions concerning"
     "finite fields, which depend on a finite field GF and an"
     "algebraic extension F of GF, for example, a zero of a polynomial"
     "over GF in F."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldPolynomialPackage2|
  (progn
    (push '|FiniteFieldPolynomialPackage2| *Packages*)
    (make-instance '|FiniteFieldPolynomialPackage2Type|)))
```

———-

1.81.17 FiniteFieldSolveLinearPolynomialEquation

— defclass FiniteFieldSolveLinearPolynomialEquationType —

```
(defclass |FiniteFieldSolveLinearPolynomialEquationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldSolveLinearPolynomialEquation")
   (marker :initform 'package)
   (abbreviation :initform 'FFSLPE)
   (comment :initform (list
     "This package solves linear diophantine equations for Bivariate polynomials"
     "over finite fields"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldSolveLinearPolynomialEquation|
  (progn
    (push '|FiniteFieldSolveLinearPolynomialEquation| *Packages*)
    (make-instance '|FiniteFieldSolveLinearPolynomialEquationType|)))
```

———-

1.81.18 FiniteFieldSquareFreeDecomposition

— defclass FiniteFieldSquareFreeDecompositionType —

```
(defclass |FiniteFieldSquareFreeDecompositionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldSquareFreeDecomposition")
   (marker :initform 'package)
   (abbreviation :initform 'FFSQFR)
   (comment :initform (list
     "Part of the package for Algebraic Function Fields in one variable (PAFF)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldSquareFreeDecomposition|
  (progn
    (push '|FiniteFieldSquareFreeDecomposition| *Packages*)
    (make-instance '|FiniteFieldSquareFreeDecompositionType|)))
```

———-

1.81.19 FiniteLinearAggregateFunctions2

— defclass FiniteLinearAggregateFunctions2Type —

```
(defclass |FiniteLinearAggregateFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteLinearAggregateFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FLAGG2)
   (comment :initform (list
     "FiniteLinearAggregateFunctions2 provides functions involving two"
     "FiniteLinearAggregates where the underlying domains might be"
     "different. An example of this might be creating a list of rational"
     "numbers by mapping a function across a list of integers where the"
     "function divides each integer by 1000."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteLinearAggregateFunctions2|
  (progn
    (push '|FiniteLinearAggregateFunctions2| *Packages*)
    (make-instance '|FiniteLinearAggregateFunctions2Type|)))
```

———-

1.81.20 FiniteLinearAggregateSort

— defclass FiniteLinearAggregateSortType —

```
(defclass |FiniteLinearAggregateSortType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteLinearAggregateSort")
   (marker :initform 'package)
   (abbreviation :initform 'FLASORT)
   (comment :initform (list
     "This package exports 3 sorting algorithms which work over"
     "FiniteLinearAggregates."
     "Sort package (in-place) for shallowlyMutable Finite Linear Aggregates"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteLinearAggregateSort|
  (progn
    (push '|FiniteLinearAggregateSort| *Packages*)
    (make-instance '|FiniteLinearAggregateSortType|)))
```

———-

1.81.21 FiniteSetAggregateFunctions2

— defclass FiniteSetAggregateFunctions2Type —

```
(defclass |FiniteSetAggregateFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteSetAggregateFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FSAGG2)
   (comment :initform (list
     "FiniteSetAggregateFunctions2 provides functions involving two"
     "finite set aggregates where the underlying domains might be"
     "different. An example of this is to create a set of rational"
     "numbers by mapping a function across a set of integers, where the"
     "function divides each integer by 1000."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteSetAggregateFunctions2|
  (progn
    (push '|FiniteSetAggregateFunctions2| *Packages*)
    (make-instance '|FiniteSetAggregateFunctions2Type|)))
```

——–-

1.81.22 FloatingComplexPackage

— defclass FloatingComplexPackageType —

```
(defclass |FloatingComplexPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FloatingComplexPackage")
   (marker :initform 'package)
   (abbreviation :initform 'FLOATCP)
   (comment :initform (list
     "This is a package for the approximation of complex solutions for"
     "systems of equations of rational functions with complex rational"
     "coefficients. The results are expressed as either complex rational"
     "numbers or complex floats depending on the type of the precision"
     "parameter which can be either a rational number or a floating point number."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatingComplexPackage|
  (progn
    (push '|FloatingComplexPackage| *Packages*)
    (make-instance '|FloatingComplexPackageType|)))
```

——–-

1.81.23 FloatingRealPackage

— defclass FloatingRealPackageType —

```
(defclass |FloatingRealPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FloatingRealPackage")
   (marker :initform 'package)
   (abbreviation :initform 'FLOATRP)
   (comment :initform (list
     "This is a package for the approximation of real solutions for"
     "systems of polynomial equations over the rational numbers."
     "The results are expressed as either rational numbers or floats"
     "depending on the type of the precision parameter which can be"
     "either a rational number or a floating point number."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatingRealPackage|
  (progn
    (push '|FloatingRealPackage| *Packages*)
    (make-instance '|FloatingRealPackageType|)))
```

———-

1.81.24 FloatSpecialFunctions

— defclass FloatSpecialFunctionsType —

```
(defclass |FloatSpecialFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FloatSpecialFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'FSFUN)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatSpecialFunctions|
  (progn
    (push '|FloatSpecialFunctions| *Packages*)
    (make-instance '|FloatSpecialFunctionsType|)))
```

———-

1.81.25 FortranCodePackage1

— defclass FortranCodePackage1Type —

```
(defclass |FortranCodePackage1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FortranCodePackage1")
   (marker :initform 'package)
   (abbreviation :initform 'FCPAK1)
   (comment :initform (list
     "FortranCodePackage1 provides some utilities for"
     "producing useful objects in FortranCode domain."
     "The Package may be used with the FortranCode domain and its"
     "printCode or possibly via an outputAsFortran."
     "(The package provides items of use in connection with ASPs"
     "in the AXIOM-NAG link and, where appropriate, naming accords"
     "with that in IRENA.)"
     "The easy-to-use functions use Fortran loop variables I1, I2,"
     "and it is users' responsibility to check that this is sensible."
     "The advanced functions use SegmentBinding to allow users control"
     "over Fortran loop variable names."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranCodePackage1|
  (progn
    (push '|FortranCodePackage1| *Packages*)
    (make-instance '|FortranCodePackage1Type|)))
```

———-

1.81.26 FortranOutputStackPackage

— defclass FortranOutputStackPackageType —

```
(defclass |FortranOutputStackPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FortranOutputStackPackage")
   (marker :initform 'package)
   (abbreviation :initform 'FOP)
   (comment :initform (list
     "Code to manipulate Fortran Output Stack"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranOutputStackPackage|
  (progn
    (push '|FortranOutputStackPackage| *Packages*)
    (make-instance '|FortranOutputStackPackageType|)))
```

———-

1.81.27 FortranPackage

— defclass FortranPackageType —

```
(defclass |FortranPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FortranPackage")
   (marker :initform 'package)
   (abbreviation :initform 'FORT)
   (comment :initform (list
     "Provides an interface to the boot code for calling Fortran"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranPackage|
  (progn
    (push '|FortranPackage| *Packages*)
    (make-instance '|FortranPackageType|)))
```

———-

1.81.28 FractionalIdealFunctions2

— defclass FractionalIdealFunctions2Type —

```
(defclass |FractionalIdealFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FractionalIdealFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FRIDEAL2)
   (comment :initform (list
     "Lifting of morphisms to fractional ideals."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionalIdealFunctions2|
  (progn
    (push '|FractionalIdealFunctions2| *Packages*)
    (make-instance '|FractionalIdealFunctions2Type|)))
```

———-

1.81.29 FractionFreeFastGaussian

— defclass FractionFreeFastGaussianType —

```
(defclass |FractionFreeFastGaussianType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FractionFreeFastGaussian")
   (marker :initform 'package)
   (abbreviation :initform 'FFFG)
   (comment :initform (list
     "This package implements the interpolation algorithm proposed in Beckermann,"
     "Bernhard and Labahn, George, Fraction-free computation of matrix rational"
     "interpolants and matrix GCDs, SIAM Journal on Matrix Analysis and"
     "Applications 22."
     "The packages defined in this file provide fast fraction free rational"
     "interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionFreeFastGaussian|
  (progn
    (push '|FractionFreeFastGaussian| *Packages*)
    (make-instance '|FractionFreeFastGaussianType|)))
```

———-

1.81.30 FractionFreeFastGaussianFractions

— defclass FractionFreeFastGaussianFractionsType —

```
(defclass |FractionFreeFastGaussianFractionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FractionFreeFastGaussianFractions")
   (marker :initform 'package)
   (abbreviation :initform 'FFFGF)
   (comment :initform (list
     "This package lifts the interpolation functions from"
     "FractionFreeFastGaussian to fractions."
     "The packages defined in this file provide fast fraction free rational"
     "interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionFreeFastGaussianFractions|
  (progn
    (push '|FractionFreeFastGaussianFractions| *Packages*)
    (make-instance '|FractionFreeFastGaussianFractionsType|)))
```

———-

1.81.31 FractionFunctions2

— defclass FractionFunctions2Type —

```
(defclass |FractionFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FractionFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FRAC2)
   (comment :initform (list
     "This package extends a map between integral domains to"
     "a map between Fractions over those domains by applying the map to the"
     "numerators and denominators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionFunctions2|
  (progn
    (push '|FractionFunctions2| *Packages*)
    (make-instance '|FractionFunctions2Type|)))
```

———-

1.81.32 FramedNonAssociativeAlgebraFunctions2

— defclass FramedNonAssociativeAlgebraFunctions2Type —

```
(defclass |FramedNonAssociativeAlgebraFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FramedNonAssociativeAlgebraFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FRNAAF2)
   (comment :initform (list
     "FramedNonAssociativeAlgebraFunctions2 implements functions between"
     "two framed non associative algebra domains defined over different rings."
     "The function map is used to coerce between algebras over different"
     "domains having the same structural constants."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedNonAssociativeAlgebraFunctions2|
  (progn
    (push '|FramedNonAssociativeAlgebraFunctions2| *Packages*)
    (make-instance '|FramedNonAssociativeAlgebraFunctions2Type|)))
```

———-

1.81.33 FunctionalSpecialFunction

— defclass FunctionalSpecialFunctionType —

```
(defclass |FunctionalSpecialFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionalSpecialFunction")
   (marker :initform 'package)
   (abbreviation :initform 'FSPECF)
   (comment :initform (list
     "Provides some special functions over an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionalSpecialFunction|
  (progn
    (push '|FunctionalSpecialFunction| *Packages*)
    (make-instance '|FunctionalSpecialFunctionType|)))
```

———-

1.81.34 FunctionFieldCategoryFunctions2

— defclass FunctionFieldCategoryFunctions2Type —

```
(defclass |FunctionFieldCategoryFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionFieldCategoryFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FFCAT2)
   (comment :initform (list
     "Lifts a map from rings to function fields over them."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionFieldCategoryFunctions2|
  (progn
    (push '|FunctionFieldCategoryFunctions2| *Packages*)
    (make-instance '|FunctionFieldCategoryFunctions2Type|)))
```

———-

1.81.35 FunctionFieldIntegralBasis

— defclass FunctionFieldIntegralBasisType —

```
(defclass |FunctionFieldIntegralBasisType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionFieldIntegralBasis")
   (marker :initform 'package)
   (abbreviation :initform 'FFINTBAS)
   (comment :initform (list
     "Integral bases for function fields of dimension one"
     "In this package R is a Euclidean domain and F is a framed algebra"
     "over R.  The package provides functions to compute the integral"
     "closure of R in the quotient field of F.  It is assumed that"
     "char(R/P) = char(R) for any prime P of R.  A typical instance of"
     "this is when R = K[x] and F is a function field over R."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionFieldIntegralBasis|
  (progn
    (push '|FunctionFieldIntegralBasis| *Packages*)
    (make-instance '|FunctionFieldIntegralBasisType|)))
```

———-

1.81.36 FunctionSpaceAssertions

— defclass FunctionSpaceAssertionsType —

```
(defclass |FunctionSpaceAssertionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceAssertions")
   (marker :initform 'package)
   (abbreviation :initform 'PMASSFS)
   (comment :initform (list
     "Attaching assertions to symbols for pattern matching"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceAssertions|
  (progn
    (push '|FunctionSpaceAssertions| *Packages*)
    (make-instance '|FunctionSpaceAssertionsType|)))
```

———-

1.81.37 FunctionSpaceAttachPredicates

— defclass FunctionSpaceAttachPredicatesType —

```
(defclass |FunctionSpaceAttachPredicatesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceAttachPredicates")
   (marker :initform 'package)
   (abbreviation :initform 'PMPREDFS)
   (comment :initform (list
     "Attaching predicates to symbols for pattern matching."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceAttachPredicates|
  (progn
    (push '|FunctionSpaceAttachPredicates| *Packages*)
    (make-instance '|FunctionSpaceAttachPredicatesType|)))
```

———-

1.81.38 FunctionSpaceComplexIntegration

— defclass FunctionSpaceComplexIntegrationType —

```
(defclass |FunctionSpaceComplexIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceComplexIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'FSCINT)
   (comment :initform (list
     "FunctionSpaceComplexIntegration provides functions for the"
     "indefinite integration of complex-valued functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceComplexIntegration|
  (progn
    (push '|FunctionSpaceComplexIntegration| *Packages*)
    (make-instance '|FunctionSpaceComplexIntegrationType|)))
```

———-

1.81.39 FunctionSpaceFunctions2

— defclass FunctionSpaceFunctions2Type —

```
(defclass |FunctionSpaceFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FS2)
   (comment :initform (list
     "This package allows a mapping R -> S to be lifted to a mapping"
     "from a function space over R to a function space over S"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceFunctions2|
  (progn
    (push '|FunctionSpaceFunctions2| *Packages*)
    (make-instance '|FunctionSpaceFunctions2Type|)))
```

———-

1.81.40 FunctionSpaceIntegration

— defclass FunctionSpaceIntegrationType —

```
(defclass |FunctionSpaceIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'FSINT)
   (comment :initform (list
     "Top-level real function integration"
     "FunctionSpaceIntegration provides functions for the"
     "indefinite integration of real-valued functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceIntegration|
  (progn
    (push '|FunctionSpaceIntegration| *Packages*)
    (make-instance '|FunctionSpaceIntegrationType|)))
```

———-

1.81.41 FunctionSpacePrimitiveElement

— defclass FunctionSpacePrimitiveElementType —

```
(defclass |FunctionSpacePrimitiveElementType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpacePrimitiveElement")
   (marker :initform 'package)
   (abbreviation :initform 'FSPRMELT)
   (comment :initform (list
     "FunctionsSpacePrimitiveElement provides functions to compute"
     "primitive elements in functions spaces"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpacePrimitiveElement|
  (progn
    (push '|FunctionSpacePrimitiveElement| *Packages*)
    (make-instance '|FunctionSpacePrimitiveElementType|)))
```

———-

1.81.42 FunctionSpaceReduce

— defclass FunctionSpaceReduceType —

```
(defclass |FunctionSpaceReduceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceReduce")
   (marker :initform 'package)
   (abbreviation :initform 'FSRED)
   (comment :initform (list
     "Reduction from a function space to the rational numbers"
     "This package provides function which replaces transcendental kernels"
     "in a function space by random integers. The correspondence between"
     "the kernels and the integers is fixed between calls to new()."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceReduce|
  (progn
    (push '|FunctionSpaceReduce| *Packages*)
    (make-instance '|FunctionSpaceReduceType|)))
```

———-

1.81.43 FunctionSpaceSum

— defclass FunctionSpaceSumType —

```
(defclass |FunctionSpaceSumType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceSum")
   (marker :initform 'package)
   (abbreviation :initform 'SUMFS)
   (comment :initform (list
     "Computes sums of top-level expressions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceSum|
  (progn
    (push '|FunctionSpaceSum| *Packages*)
    (make-instance '|FunctionSpaceSumType|)))
```

———-

1.81.44 FunctionSpaceToExponentialExpansion

— defclass FunctionSpaceToExponentialExpansionType —

```
(defclass |FunctionSpaceToExponentialExpansionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceToExponentialExpansion")
   (marker :initform 'package)
   (abbreviation :initform 'FS2EXPXP)
   (comment :initform (list
     "This package converts expressions in some function space to exponential"
     "expansions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceToExponentialExpansion|
  (progn
    (push '|FunctionSpaceToExponentialExpansion| *Packages*)
    (make-instance '|FunctionSpaceToExponentialExpansionType|)))
```

———-

1.81.45 FunctionSpaceToUnivariatePowerSeries

— defclass FunctionSpaceToUnivariatePowerSeriesType —

```
(defclass |FunctionSpaceToUnivariatePowerSeriesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceToUnivariatePowerSeries")
   (marker :initform 'package)
   (abbreviation :initform 'FS2UPS)
   (comment :initform (list
     "This package converts expressions in some function space to power"
     "series in a variable x with coefficients in that function space."
     "The function exprToUPS converts expressions to power series"
     "whose coefficients do not contain the variable x. The function"
     "exprToGenUPS converts functional expressions to power series"
     "whose coefficients may involve functions of log(x)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceToUnivariatePowerSeries|
  (progn
    (push '|FunctionSpaceToUnivariatePowerSeries| *Packages*)
    (make-instance '|FunctionSpaceToUnivariatePowerSeriesType|)))
```

———-

1.81.46 FunctionSpaceUnivariatePolynomialFactor

— defclass FunctionSpaceUnivariatePolynomialFactorType —

```
(defclass |FunctionSpaceUnivariatePolynomialFactorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceUnivariatePolynomialFactor")
   (marker :initform 'package)
   (abbreviation :initform 'FSUPFACT)
   (comment :initform (list
     "This package is used internally by IR2F"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceUnivariatePolynomialFactor|
  (progn
    (push '|FunctionSpaceUnivariatePolynomialFactor| *Packages*)
    (make-instance '|FunctionSpaceUnivariatePolynomialFactorType|)))
```

———-

1.82 G

1.82.1 GaloisGroupFactorizationUtilities

— defclass GaloisGroupFactorizationUtilitiesType —

```
(defclass |GaloisGroupFactorizationUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GaloisGroupFactorizationUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'GALFACTU)
   (comment :initform (list
     "GaloisGroupFactorizationUtilities provides functions"
     "that will be used by the factorizer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupFactorizationUtilities|
  (progn
    (push '|GaloisGroupFactorizationUtilities| *Packages*)
    (make-instance '|GaloisGroupFactorizationUtilitiesType|)))
```

———-

1.82.2 GaloisGroupFactorizer

— defclass GaloisGroupFactorizerType —

```
(defclass |GaloisGroupFactorizerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GaloisGroupFactorizer")
   (marker :initform 'package)
   (abbreviation :initform 'GALFACT)
   (comment :initform (list
     "GaloisGroupFactorizationUtilities provides functions"
     "that will be used by the factorizer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupFactorizer|
  (progn
    (push '|GaloisGroupFactorizer| *Packages*)
    (make-instance '|GaloisGroupFactorizerType|)))
```

———-

1.82.3 GaloisGroupPolynomialUtilities

— defclass GaloisGroupPolynomialUtilitiesType —

```
(defclass |GaloisGroupPolynomialUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GaloisGroupPolynomialUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'GALPOLYU)
   (comment :initform (list
     "GaloisGroupPolynomialUtilities provides useful"
     "functions for univariate polynomials which should be added to"
     "UnivariatePolynomialCategory or to Factored"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupPolynomialUtilities|
  (progn
    (push '|GaloisGroupPolynomialUtilities| *Packages*)
    (make-instance '|GaloisGroupPolynomialUtilitiesType|)))
```

————-

1.82.4 GaloisGroupUtilities

— defclass GaloisGroupUtilitiesType —

```
(defclass |GaloisGroupUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GaloisGroupUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'GALUTIL)
   (comment :initform (list
     "GaloisGroupUtilities provides several useful functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupUtilities|
  (progn
    (push '|GaloisGroupUtilities| *Packages*)
    (make-instance '|GaloisGroupUtilitiesType|)))
```

————-

1.82.5 GaussianFactorizationPackage

— defclass GaussianFactorizationPackageType —

```
(defclass |GaussianFactorizationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GaussianFactorizationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GAUSSFAC)
   (comment :initform (list
     "Package for the factorization of complex or gaussian integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |GaussianFactorizationPackage|
  (progn
    (push '|GaussianFactorizationPackage| *Packages*)
    (make-instance '|GaussianFactorizationPackageType|)))
```

———–

1.82.6 GeneralHenselPackage

— defclass GeneralHenselPackageType —

```
(defclass |GeneralHenselPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GeneralHenselPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GHENSEL)
   (comment :initform (list
     "Used for Factorization of bivariate polynomials over a finite field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralHenselPackage|
  (progn
    (push '|GeneralHenselPackage| *Packages*)
    (make-instance '|GeneralHenselPackageType|)))
```

———–

1.82.7 GeneralizedMultivariateFactorize

— defclass GeneralizedMultivariateFactorizeType —

```
(defclass |GeneralizedMultivariateFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GeneralizedMultivariateFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'GENMFACT)
   (comment :initform (list
     "This is the top level package for doing multivariate factorization"
     "over basic domains like Integer or Fraction Integer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralizedMultivariateFactorize|
  (progn
    (push '|GeneralizedMultivariateFactorize| *Packages*)
    (make-instance '|GeneralizedMultivariateFactorizeType|)))
```

———-

1.82.8 GeneralPackageForAlgebraicFunctionField

— defclass GeneralPackageForAlgebraicFunctionFieldType —

```
(defclass |GeneralPackageForAlgebraicFunctionFieldType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GeneralPackageForAlgebraicFunctionField")
   (marker :initform 'package)
   (abbreviation :initform 'GPAFF)
   (comment :initform (list
     "A package that implements the Brill-Noether algorithm. Part of the"
     "PAFF package."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralPackageForAlgebraicFunctionField|
  (progn
    (push '|GeneralPackageForAlgebraicFunctionField| *Packages*)
    (make-instance '|GeneralPackageForAlgebraicFunctionFieldType|)))
```

———-

1.82.9 GeneralPolynomialGcdPackage

— defclass GeneralPolynomialGcdPackageType —

```
(defclass |GeneralPolynomialGcdPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GeneralPolynomialGcdPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GENPGCD)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralPolynomialGcdPackage|
  (progn
    (push '|GeneralPolynomialGcdPackage| *Packages*)
    (make-instance '|GeneralPolynomialGcdPackageType|)))
```

———-

1.82.10 GenerateUnivariatePowerSeries

— defclass GenerateUnivariatePowerSeriesType —

```
(defclass |GenerateUnivariatePowerSeriesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GenerateUnivariatePowerSeries")
   (marker :initform 'package)
   (abbreviation :initform 'GENUPS)
   (comment :initform (list
     "GenerateUnivariatePowerSeries provides functions that create"
     "power series from explicit formulas for their nth coefficient."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenerateUnivariatePowerSeries|
  (progn
    (push '|GenerateUnivariatePowerSeries| *Packages*)
    (make-instance '|GenerateUnivariatePowerSeriesType|)))
```

———-

1.82.11 GenExEuclid

— defclass GenExEuclidType —

```
(defclass |GenExEuclidType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GenExEuclid")
   (marker :initform 'package)
   (abbreviation :initform 'GENEEZ)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenExEuclid|
  (progn
    (push '|GenExEuclid| *Packages*)
    (make-instance '|GenExEuclidType|)))
```

———-

1.82.12 GenUFactorize

— defclass GenUFactorizeType —

```
(defclass |GenUFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GenUFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'GENUFACT)
   (comment :initform (list
     "This package provides operations for the factorization"
     "of univariate polynomials with integer"
     "coefficients. The factorization is done by 'lifting' the"
     "finite 'berlekamp's factorization"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenUFactorize|
  (progn
    (push '|GenUFactorize| *Packages*)
    (make-instance '|GenUFactorizeType|)))
```

———-

1.82.13 GenusZeroIntegration

— defclass GenusZeroIntegrationType —

```
(defclass |GenusZeroIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GenusZeroIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTG0)
   (comment :initform (list
     "Rationalization of several types of genus 0 integrands"
     "This internal package rationalises integrands on curves of the form"
     "     y\^2 = a x\^2 + b x + c"
     "     y\^2 = (a x + b) / (c x + d)"
     "     f(x, y) = 0 where f has degree 1 in x"
     "The rationalization is done for integration, limited integration,"
     "extended integration and the risch differential equation"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenusZeroIntegration|
  (progn
    (push '|GenusZeroIntegration| *Packages*)
    (make-instance '|GenusZeroIntegrationType|)))
```

———-

1.82.14 GnuDraw

— defclass GnuDrawType —

```
(defclass |GnuDrawType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GnuDraw")
   (marker :initform 'package)
   (abbreviation :initform 'GDRAW)
   (comment :initform (list
     "This package provides support for gnuplot. These routines"
     "generate output files contain gnuplot scripts that may be"
     "processed directly by gnuplot. This is especially convenient"
     "in the axiom-wiki environment where gnuplot is called from"
     "LaTeX via gnuplottex."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GnuDraw|
  (progn
    (push '|GnuDraw| *Packages*)
    (make-instance '|GnuDrawType|)))
```

———-

1.82.15 GosperSummationMethod

— defclass GosperSummationMethodType —

```
(defclass |GosperSummationMethodType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GosperSummationMethod")
   (marker :initform 'package)
   (abbreviation :initform 'GOSPER)
   (comment :initform (list
     "Gosper's summation algorithm."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GosperSummationMethod|
  (progn
    (push '|GosperSummationMethod| *Packages*)
    (make-instance '|GosperSummationMethodType|)))
```

———-

1.82.16 GraphicsDefaults

— defclass GraphicsDefaultsType —

```
(defclass |GraphicsDefaultsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GraphicsDefaults")
   (marker :initform 'package)
   (abbreviation :initform 'GRDEF)
   (comment :initform (list
     "TwoDimensionalPlotSettings sets global flags and constants"
     "for 2-dimensional plotting."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GraphicsDefaults|
  (progn
    (push '|GraphicsDefaults| *Packages*)
    (make-instance '|GraphicsDefaultsType|)))
```

———-

1.82.17 Graphviz

— defclass GraphvizType —

```
(defclass |GraphvizType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Graphviz")
   (marker :initform 'package)
   (abbreviation :initform 'GRAPHVIZ)
   (comment :initform (list
     "Low level tools for creating and viewing graphs using graphviz"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Graphviz|
  (progn
    (push '|Graphviz| *Packages*)
    (make-instance '|GraphvizType|)))
```

———-

1.82.18 GrayCode

— defclass GrayCodeType —

```
(defclass |GrayCodeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GrayCode")
   (marker :initform 'package)
   (abbreviation :initform 'GRAY)
   (comment :initform (list
     "GrayCode provides a function for efficiently running"
     "through all subsets of a finite set, only changing one element"
     "by another one."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GrayCode|
  (progn
    (push '|GrayCode| *Packages*)
    (make-instance '|GrayCodeType|)))
```

———–

1.82.19 GroebnerFactorizationPackage

— defclass GroebnerFactorizationPackageType —

```
(defclass |GroebnerFactorizationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GroebnerFactorizationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GBF)
   (comment :initform (list
     "GroebnerFactorizationPackage provides the function"
     "groebnerFactor which uses the factorization routines of Axiom to"
     "factor each polynomial under consideration while doing the groebner basis"
     "algorithm. Then it writes the ideal as an intersection of ideals"
     "determined by the irreducible factors. Note that the whole ring may"
     "occur as well as other redundancies. We also use the fact, that from the"
     "second factor on we can assume that the preceding factors are"
     "not equal to 0 and we divide all polynomials under considerations"
     "by the elements of this list of 'nonZeroRestrictions'."
     "The result is a list of groebner bases, whose union of solutions"
     "of the corresponding systems of equations is the solution of"
     "the system of equation corresponding to the input list."
     "The term ordering is determined by the polynomial type used."
     "Suggested types include"
     "DistributedMultivariatePolynomial,"
     "HomogeneousDistributedMultivariatePolynomial,"
     "GeneralDistributedMultivariatePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerFactorizationPackage|
  (progn
    (push '|GroebnerFactorizationPackage| *Packages*)
    (make-instance '|GroebnerFactorizationPackageType|)))
```

———-

1.82.20 GroebnerInternalPackage

— defclass GroebnerInternalPackageType —

```
(defclass |GroebnerInternalPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GroebnerInternalPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GBINTERN)
   (comment :initform (list
     "This package provides low level tools for Groebner basis computations"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerInternalPackage|
  (progn
    (push '|GroebnerInternalPackage| *Packages*)
    (make-instance '|GroebnerInternalPackageType|)))
```

———-

1.82.21 GroebnerPackage

— defclass GroebnerPackageType —

```
(defclass |GroebnerPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GroebnerPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GB)
   (comment :initform (list
     "GroebnerPackage computes groebner"
     "bases for polynomial ideals. The basic computation provides a distinguished"
     "set of generators for polynomial ideals over fields. This basis allows an"
     "easy test for membership: the operation normalForm"
     "returns zero on ideal members. When the provided coefficient domain, Dom,"
     "is not a field, the result is equivalent to considering the extended"
     "ideal with Fraction(Dom) as coefficients, but considerably more"
     "efficient since all calculations are performed in Dom. Additional"
     "argument 'info' and 'redcrit' can be given to provide incremental"
     "information during computation. Argument 'info' produces a computational"
     "summary for each s-polynomial."
     "Argument 'redcrit' prints out the reduced critical pairs. The term ordering"
     "is determined by the polynomial type used. Suggested types include"
     "DistributedMultivariatePolynomial,"
     "HomogeneousDistributedMultivariatePolynomial,"
     "GeneralDistributedMultivariatePolynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerPackage|
  (progn
    (push '|GroebnerPackage| *Packages*)
    (make-instance '|GroebnerPackageType|)))
```

———-

1.82.22 GroebnerSolve

— defclass GroebnerSolveType —

```
(defclass |GroebnerSolveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GroebnerSolve")
   (marker :initform 'package)
   (abbreviation :initform 'GROEBSOL)
   (comment :initform (list
     "Solve systems of polynomial equations using Groebner bases"
     "Total order Groebner bases are computed and then converted to lex ones"
     "This package is mostly intended for internal use."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerSolve|
  (progn
    (push '|GroebnerSolve| *Packages*)
    (make-instance '|GroebnerSolveType|)))
```

———-

1.82.23 Guess

— defclass GuessType —

```
(defclass |GuessType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Guess")
   (marker :initform 'package)
   (abbreviation :initform 'GUESS)
   (comment :initform (list
     "This package implements guessing of sequences. Packages for the"
     "most common cases are provided as GuessInteger,"
     "GuessPolynomial, etc."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Guess|
  (progn
    (push '|Guess| *Packages*)
    (make-instance '|GuessType|)))
```

———-

1.82.24 GuessAlgebraicNumber

— defclass GuessAlgebraicNumberType —

```
(defclass |GuessAlgebraicNumberType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GuessAlgebraicNumber")
   (marker :initform 'package)
   (abbreviation :initform 'GUESSAN)
   (comment :initform (list
     "This package exports guessing of sequences of rational functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessAlgebraicNumber|
  (progn
    (push '|GuessAlgebraicNumber| *Packages*)
    (make-instance '|GuessAlgebraicNumberType|)))
```

———-

1.82.25 GuessFinite

— defclass GuessFiniteType —

```
(defclass |GuessFiniteType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GuessFinite")
   (marker :initform 'package)
   (abbreviation :initform 'GUESSF)
   (comment :initform (list
     "This package exports guessing of sequences of numbers in a finite field"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessFinite|
  (progn
    (push '|GuessFinite| *Packages*)
    (make-instance '|GuessFiniteType|)))
```

———-

1.82.26 GuessFiniteFunctions

— defclass GuessFiniteFunctionsType —

```
(defclass |GuessFiniteFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GuessFiniteFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'GUESSF1)
   (comment :initform (list
     "This package exports guessing of sequences of numbers in a finite field"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessFiniteFunctions|
  (progn
    (push '|GuessFiniteFunctions| *Packages*)
    (make-instance '|GuessFiniteFunctionsType|)))
```

———-

1.82.27 GuessInteger

— defclass GuessIntegerType —

```
(defclass |GuessIntegerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GuessInteger")
   (marker :initform 'package)
   (abbreviation :initform 'GUESSINT)
   (comment :initform (list
     "This package exports guessing of sequences of rational numbers"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessInteger|
  (progn
    (push '|GuessInteger| *Packages*)
    (make-instance '|GuessIntegerType|)))
```

———-

1.82.28 GuessPolynomial

— defclass GuessPolynomialType —

```
(defclass |GuessPolynomialType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GuessPolynomial")
   (marker :initform 'package)
   (abbreviation :initform 'GUESSP)
   (comment :initform (list
     "This package exports guessing of sequences of rational functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessPolynomial|
  (progn
    (push '|GuessPolynomial| *Packages*)
    (make-instance '|GuessPolynomialType|)))
```

———-

1.82.29 GuessUnivariatePolynomial

— defclass GuessUnivariatePolynomialType —

```
(defclass |GuessUnivariatePolynomialType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GuessUnivariatePolynomial")
   (marker :initform 'package)
   (abbreviation :initform 'GUESSUP)
   (comment :initform (list
     "This package exports guessing of sequences of univariate rational functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessUnivariatePolynomial|
  (progn
    (push '|GuessUnivariatePolynomial| *Packages*)
    (make-instance '|GuessUnivariatePolynomialType|)))
```

————-

1.83 H

1.83.1 HallBasis

— defclass HallBasisType —

```
(defclass |HallBasisType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "HallBasis")
   (marker :initform 'package)
   (abbreviation :initform 'HB)
   (comment :initform (list
     "Generate a basis for the free Lie algebra on n"
     "generators over a ring R with identity up to basic commutators"
     "of length c using the algorithm of P. Hall as given in Serre's"
     "book Lie Groups -- Lie Algebras"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HallBasis|
  (progn
    (push '|HallBasis| *Packages*)
    (make-instance '|HallBasisType|)))
```

————-

1.83.2 HeuGcd

— defclass HeuGcdType —

```
(defclass |HeuGcdType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "HeuGcd")
   (marker :initform 'package)
   (abbreviation :initform 'HEUGCD)
   (comment :initform (list
     "This package provides the functions for the heuristic integer gcd."
     "Geddes's algorithm,for univariate polynomials with integer coefficients"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |HeuGcd|
  (progn
    (push '|HeuGcd| *Packages*)
    (make-instance '|HeuGcdType|)))
```

———-

1.84 I

1.84.1 IdealDecompositionPackage

— defclass IdealDecompositionPackageType —

```
(defclass |IdealDecompositionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IdealDecompositionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'IDECOMP)
   (comment :initform (list
     "This package provides functions for the primary decomposition of"
     "polynomial ideals over the rational numbers. The ideals are members"
     "of the PolynomialIdeals domain, and the polynomial generators are"
     "required to be from the DistributedMultivariatePolynomial domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IdealDecompositionPackage|
  (progn
    (push '|IdealDecompositionPackage| *Packages*)
    (make-instance '|IdealDecompositionPackageType|)))
```

———-

1.84.2 IncrementingMaps

— defclass IncrementingMapsType —

```
(defclass |IncrementingMapsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IncrementingMaps")
   (marker :initform 'package)
   (abbreviation :initform 'INCRMAPS)
   (comment :initform (list
     "This package provides operations to create incrementing functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IncrementingMaps|
  (progn
    (push '|IncrementingMaps| *Packages*)
    (make-instance '|IncrementingMapsType|)))
```

———-

1.84.3 InfiniteProductCharacteristicZero

— defclass InfiniteProductCharacteristicZeroType —

```
(defclass |InfiniteProductCharacteristicZeroType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InfiniteProductCharacteristicZero")
   (marker :initform 'package)
   (abbreviation :initform 'INFPROD0)
   (comment :initform (list
     "This package computes infinite products of univariate Taylor series"
     "over an integral domain of characteristic 0."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteProductCharacteristicZero|
  (progn
    (push '|InfiniteProductCharacteristicZero| *Packages*)
    (make-instance '|InfiniteProductCharacteristicZeroType|)))
```

———-

1.84.4 InfiniteProductFiniteField

— defclass InfiniteProductFiniteFieldType —

```
(defclass |InfiniteProductFiniteFieldType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InfiniteProductFiniteField")
   (marker :initform 'package)
   (abbreviation :initform 'INPRODFF)
   (comment :initform (list
     "This package computes infinite products of univariate Taylor series"
     "over an arbitrary finite field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |InfiniteProductFiniteField|
  (progn
    (push '|InfiniteProductFiniteField| *Packages*)
    (make-instance '|InfiniteProductFiniteFieldType|)))
```

———-

1.84.5 InfiniteProductPrimeField

— defclass InfiniteProductPrimeFieldType —

```
(defclass |InfiniteProductPrimeFieldType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InfiniteProductPrimeField")
   (marker :initform 'package)
   (abbreviation :initform 'INPRODPF)
   (comment :initform (list
     "This package computes infinite products of univariate Taylor series"
     "over a field of prime order."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteProductPrimeField|
  (progn
    (push '|InfiniteProductPrimeField| *Packages*)
    (make-instance '|InfiniteProductPrimeFieldType|)))
```

———-

1.84.6 InfiniteTupleFunctions2

— defclass InfiniteTupleFunctions2Type —

```
(defclass |InfiniteTupleFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InfiniteTupleFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ITFUN2)
   (comment :initform (list
     "Functions defined on streams with entries in two sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteTupleFunctions2|
  (progn
    (push '|InfiniteTupleFunctions2| *Packages*)
    (make-instance '|InfiniteTupleFunctions2Type|)))
```

———-

1.84.7 InfiniteTupleFunctions3

— defclass InfiniteTupleFunctions3Type —

```
(defclass |InfiniteTupleFunctions3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InfiniteTupleFunctions3")
   (marker :initform 'package)
   (abbreviation :initform 'ITFUN3)
   (comment :initform (list
     "Functions defined on streams with entries in two sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteTupleFunctions3|
  (progn
    (push '|InfiniteTupleFunctions3| *Packages*)
    (make-instance '|InfiniteTupleFunctions3Type|)))
```

———-

1.84.8 Infinity

— defclass InfinityType —

```
(defclass |InfinityType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Infinity")
   (marker :initform 'package)
   (abbreviation :initform 'INFINITY)
   (comment :initform (list
     "Default infinity signatures for the interpreter"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Infinity|
  (progn
    (push '|Infinity| *Packages*)
    (make-instance '|InfinityType|)))
```

———-

1.84.9 InnerAlgFactor

— defclass InnerAlgFactorType —

```
(defclass |InnerAlgFactorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerAlgFactor")
   (marker :initform 'package)
   (abbreviation :initform 'IALGFACT)
   (comment :initform (list
     "Factorisation in a simple algebraic extension"
     "Factorization of univariate polynomials with coefficients in an"
     "algebraic extension of a field over which we can factor UP's"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerAlgFactor|
  (progn
    (push '|InnerAlgFactor| *Packages*)
    (make-instance '|InnerAlgFactorType|)))
```

———-

1.84.10 InnerCommonDenominator

— defclass InnerCommonDenominatorType —

```
(defclass |InnerCommonDenominatorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerCommonDenominator")
   (marker :initform 'package)
   (abbreviation :initform 'ICDEN)
   (comment :initform (list
     "InnerCommonDenominator provides functions to compute"
     "the common denominator of a finite linear aggregate of elements"
     "of the quotient field of an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerCommonDenominator|
  (progn
    (push '|InnerCommonDenominator| *Packages*)
    (make-instance '|InnerCommonDenominatorType|)))
```

———-

1.84.11 InnerMatrixLinearAlgebraFunctions

— defclass InnerMatrixLinearAlgebraFunctionsType —

```
(defclass |InnerMatrixLinearAlgebraFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerMatrixLinearAlgebraFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'IMATLIN)
   (comment :initform (list
     "InnerMatrixLinearAlgebraFunctions is an internal package"
     "which provides standard linear algebra functions on domains in"
     "MatrixCategory"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerMatrixLinearAlgebraFunctions|
  (progn
    (push '|InnerMatrixLinearAlgebraFunctions| *Packages*)
    (make-instance '|InnerMatrixLinearAlgebraFunctionsType|)))
```

————-

1.84.12 InnerMatrixQuotientFieldFunctions

— defclass InnerMatrixQuotientFieldFunctionsType —

```
(defclass |InnerMatrixQuotientFieldFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerMatrixQuotientFieldFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'IMATQF)
   (comment :initform (list
     "InnerMatrixQuotientFieldFunctions provides functions on matrices"
     "over an integral domain which involve the quotient field of that integral"
     "domain.  The functions rowEchelon and inverse return matrices with"
     "entries in the quotient field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerMatrixQuotientFieldFunctions|
  (progn
    (push '|InnerMatrixQuotientFieldFunctions| *Packages*)
    (make-instance '|InnerMatrixQuotientFieldFunctionsType|)))
```

————-

1.84.13 InnerModularGcd

— defclass InnerModularGcdType —

```
(defclass |InnerModularGcdType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerModularGcd")
   (marker :initform 'package)
   (abbreviation :initform 'INMODGCD)
   (comment :initform (list
     "This file contains the functions for modular gcd algorithm"
     "for univariate polynomials with coefficients in a"
     "non-trivial euclidean domain (not a field)."
     "The package parametrised by the coefficient domain,"
     "the polynomial domain, a prime, and a function for choosing the next prime"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerModularGcd|
  (progn
    (push '|InnerModularGcd| *Packages*)
    (make-instance '|InnerModularGcdType|)))
```

———–

1.84.14 InnerMultFact

— defclass InnerMultFactType —

```
(defclass |InnerMultFactType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerMultFact")
   (marker :initform 'package)
   (abbreviation :initform 'INNMFACT)
   (comment :initform (list
     "This is an inner package for factoring multivariate polynomials"
     "over various coefficient domains in characteristic 0."
     "The univariate factor operation is passed as a parameter."
     "Multivariate hensel lifting is used to lift the univariate"
     "factorization"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerMultFact|
  (progn
    (push '|InnerMultFact| *Packages*)
    (make-instance '|InnerMultFactType|)))
```

———-

1.84.15 InnerNormalBasisFieldFunctions

— defclass InnerNormalBasisFieldFunctionsType —

```
(defclass |InnerNormalBasisFieldFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerNormalBasisFieldFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'INBFF)
   (comment :initform (list
     "InnerNormalBasisFieldFunctions(GF) (unexposed)"
     "This package has functions used by"
     "every normal basis finite field extension domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerNormalBasisFieldFunctions|
  (progn
    (push '|InnerNormalBasisFieldFunctions| *Packages*)
    (make-instance '|InnerNormalBasisFieldFunctionsType|)))
```

———-

1.84.16 InnerNumericEigenPackage

— defclass InnerNumericEigenPackageType —

```
(defclass |InnerNumericEigenPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerNumericEigenPackage")
   (marker :initform 'package)
   (abbreviation :initform 'INEP)
   (comment :initform (list
     "This package is the inner package to be used by NumericRealEigenPackage"
     "and NumericComplexEigenPackage for the computation of numeric"
     "eigenvalues and eigenvectors."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerNumericEigenPackage|
  (progn
    (push '|InnerNumericEigenPackage| *Packages*)
    (make-instance '|InnerNumericEigenPackageType|)))
```

———-

1.84.17 InnerNumericFloatSolvePackage

— defclass InnerNumericFloatSolvePackageType —

```
(defclass |InnerNumericFloatSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerNumericFloatSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'INFSP)
   (comment :initform (list
     "This is an internal package"
     "for computing approximate solutions to systems of polynomial equations."
     "The parameter K specifies the coefficient field of the input polynomials"
     "and must be either \spad{Fraction(Integer)} or"
     "Complex(Fraction Integer)."
     "The parameter F specifies where the solutions must lie and can"
     "be one of the following: Float, Fraction(Integer), Complex(Float),"
     "Complex(Fraction Integer). The last parameter specifies the type"
     "of the precision operand and must be either Fraction(Integer) or"
     "Float."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerNumericFloatSolvePackage|
  (progn
    (push '|InnerNumericFloatSolvePackage| *Packages*)
    (make-instance '|InnerNumericFloatSolvePackageType|)))
```

————-

1.84.18 InnerPolySign

— defclass InnerPolySignType —

```
(defclass |InnerPolySignType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerPolySign")
   (marker :initform 'package)
   (abbreviation :initform 'INPSIGN)
   (comment :initform (list
     "Find the sign of a polynomial around a point or infinity."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerPolySign|
  (progn
    (push '|InnerPolySign| *Packages*)
    (make-instance '|InnerPolySignType|)))
```

———-

1.84.19 InnerPolySum

— defclass InnerPolySumType —

```
(defclass |InnerPolySumType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerPolySum")
   (marker :initform 'package)
   (abbreviation :initform 'ISUMP)
   (comment :initform (list
     "Tools for the summation packages of polynomials"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerPolySum|
  (progn
    (push '|InnerPolySum| *Packages*)
    (make-instance '|InnerPolySumType|)))
```

———-

1.84.20 InnerTrigonometricManipulations

— defclass InnerTrigonometricManipulationsType —

```
(defclass |InnerTrigonometricManipulationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerTrigonometricManipulations")
   (marker :initform 'package)
   (abbreviation :initform 'ITRIGMNP)
   (comment :initform (list
     "This package provides transformations from trigonometric functions"
     "to exponentials and logarithms, and back."
     "F and FG should be the same type of function space."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerTrigonometricManipulations|
  (progn
    (push '|InnerTrigonometricManipulations| *Packages*)
    (make-instance '|InnerTrigonometricManipulationsType|)))
```

———-

1.84.21 InputFormFunctions1

— defclass InputFormFunctions1Type —

```
(defclass |InputFormFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InputFormFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'INFORM1)
   (comment :initform (list
     "Tools for manipulating input forms."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InputFormFunctions1|
  (progn
    (push '|InputFormFunctions1| *Packages*)
    (make-instance '|InputFormFunctions1Type|)))
```

———-

1.84.22 InterfaceGroebnerPackage

— defclass InterfaceGroebnerPackageType —

```
(defclass |InterfaceGroebnerPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InterfaceGroebnerPackage")
   (marker :initform 'package)
   (abbreviation :initform 'INTERGB)
   (comment :initform (list
     "Part of the Package for Algebraic Function Fields in one variable PAFF"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InterfaceGroebnerPackage|
  (progn
    (push '|InterfaceGroebnerPackage| *Packages*)
    (make-instance '|InterfaceGroebnerPackageType|)))
```

———-

1.84.23 IntegerBits

— defclass IntegerBitsType —

```
(defclass |IntegerBitsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerBits")
   (marker :initform 'package)
   (abbreviation :initform 'INTBIT)
   (comment :initform (list
     "This  package provides functions to lookup bits in integers "))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerBits|
  (progn
    (push '|IntegerBits| *Packages*)
    (make-instance '|IntegerBitsType|)))
```

———-

1.84.24 IntegerCombinatoricFunctions

— defclass IntegerCombinatoricFunctionsType —

```
(defclass |IntegerCombinatoricFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerCombinatoricFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'COMBINAT)
   (comment :initform (list
     "The IntegerCombinatoricFunctions package provides some"
     "standard functions in combinatorics."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerCombinatoricFunctions|
  (progn
    (push '|IntegerCombinatoricFunctions| *Packages*)
    (make-instance '|IntegerCombinatoricFunctionsType|)))
```

———-

1.84.25 IntegerFactorizationPackage

— defclass IntegerFactorizationPackageType —

```
(defclass |IntegerFactorizationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerFactorizationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'INTFACT)
   (comment :initform (list
     "This Package contains basic methods for integer factorization."
     "The factor operation employs trial division up to 10,000.  It"
     "then tests to see if n is a perfect power before using Pollards"
     "rho method.  Because Pollards method may fail, the result"
     "of factor may contain composite factors.  We should also employ"
     "Lenstra's eliptic curve method."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerFactorizationPackage|
  (progn
    (push '|IntegerFactorizationPackage| *Packages*)
    (make-instance '|IntegerFactorizationPackageType|)))
```

———-

1.84.26 IntegerLinearDependence

— defclass IntegerLinearDependenceType —

```
(defclass |IntegerLinearDependenceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerLinearDependence")
   (marker :initform 'package)
   (abbreviation :initform 'ZLINDEP)
   (comment :initform (list
     "Test for linear dependence over the integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerLinearDependence|
  (progn
    (push '|IntegerLinearDependence| *Packages*)
    (make-instance '|IntegerLinearDependenceType|)))
```

———-

1.84.27 IntegerNumberTheoryFunctions

— defclass IntegerNumberTheoryFunctionsType —

```
(defclass |IntegerNumberTheoryFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerNumberTheoryFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'INTHEORY)
   (comment :initform (list
     "This package provides various number theoretic functions on the integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerNumberTheoryFunctions|
  (progn
    (push '|IntegerNumberTheoryFunctions| *Packages*)
    (make-instance '|IntegerNumberTheoryFunctionsType|)))
```

———-

1.84.28 IntegerPrimesPackage

— defclass IntegerPrimesPackageType —

```
(defclass |IntegerPrimesPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerPrimesPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PRIMES)
   (comment :initform (list
     "The IntegerPrimesPackage implements a modification of"
     "Rabin's probabilistic"
     "primality test and the utility functions nextPrime,"
     "prevPrime and primes."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerPrimesPackage|
  (progn
    (push '|IntegerPrimesPackage| *Packages*)
    (make-instance '|IntegerPrimesPackageType|)))
```

———-

1.84.29 IntegerRetractions

— defclass IntegerRetractionsType —

```
(defclass |IntegerRetractionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerRetractions")
   (marker :initform 'package)
   (abbreviation :initform 'INTRET)
   (comment :initform (list
     "Provides integer testing and retraction functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerRetractions|
  (progn
    (push '|IntegerRetractions| *Packages*)
    (make-instance '|IntegerRetractionsType|)))
```

———-

1.84.30 IntegerRoots

— defclass IntegerRootsType —

```
(defclass |IntegerRootsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerRoots")
   (marker :initform 'package)
   (abbreviation :initform 'IROOT)
   (comment :initform (list
     "The IntegerRoots package computes square roots and"
     "nth roots of integers efficiently."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerRoots|
  (progn
    (push '|IntegerRoots| *Packages*)
    (make-instance '|IntegerRootsType|)))
```

———-

1.84.31 IntegerSolveLinearPolynomialEquation

— defclass IntegerSolveLinearPolynomialEquationType —

```
(defclass |IntegerSolveLinearPolynomialEquationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerSolveLinearPolynomialEquation")
   (marker :initform 'package)
   (abbreviation :initform 'INTSLPE)
   (comment :initform (list
     "This package provides the implementation for the"
     "solveLinearPolynomialEquation"
     "operation over the integers. It uses a lifting technique"
     "from the package GenExEuclid"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerSolveLinearPolynomialEquation|
  (progn
    (push '|IntegerSolveLinearPolynomialEquation| *Packages*)
    (make-instance '|IntegerSolveLinearPolynomialEquationType|)))
```

———-

1.84.32 IntegralBasisTools

— defclass IntegralBasisToolsType —

```
(defclass |IntegralBasisToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegralBasisTools")
   (marker :initform 'package)
   (abbreviation :initform 'IBATOOL)
   (comment :initform (list
     "This package contains functions used in the packages"
     "FunctionFieldIntegralBasis and NumberFieldIntegralBasis."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegralBasisTools|
  (progn
    (push '|IntegralBasisTools| *Packages*)
    (make-instance '|IntegralBasisToolsType|)))
```

———-

1.84.33 IntegralBasisPolynomialTools

— defclass IntegralBasisPolynomialToolsType —

```
(defclass |IntegralBasisPolynomialToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegralBasisPolynomialTools")
   (marker :initform 'package)
   (abbreviation :initform 'IBPTOOLS)
   (comment :initform (list
     "IntegralBasisPolynomialTools provides functions for mapping functions"
     "on the coefficients of univariate and bivariate polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegralBasisPolynomialTools|
  (progn
    (push '|IntegralBasisPolynomialTools| *Packages*)
    (make-instance '|IntegralBasisPolynomialToolsType|)))
```

———-

1.84.34 IntegrationResultFunctions2

— defclass IntegrationResultFunctions2Type —

```
(defclass |IntegrationResultFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegrationResultFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'IR2)
   (comment :initform (list
     "Internally used by the integration packages"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResultFunctions2|
  (progn
    (push '|IntegrationResultFunctions2| *Packages*)
    (make-instance '|IntegrationResultFunctions2Type|)))
```

———-

1.84.35 IntegrationResultRFToFunction

— defclass IntegrationResultRFToFunctionType —

```
(defclass |IntegrationResultRFToFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegrationResultRFToFunction")
   (marker :initform 'package)
   (abbreviation :initform 'IRRF2F)
   (comment :initform (list
     "Conversion of integration results to top-level expressions."
     "This package allows a sum of logs over the roots of a polynomial"
     "to be expressed as explicit logarithms and arc tangents, provided"
     "that the indexing polynomial can be factored into quadratics."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResultRFToFunction|
  (progn
    (push '|IntegrationResultRFToFunction| *Packages*)
    (make-instance '|IntegrationResultRFToFunctionType|)))
```

———-

1.84.36 IntegrationResultToFunction

— defclass IntegrationResultToFunctionType —

```
(defclass |IntegrationResultToFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegrationResultToFunction")
   (marker :initform 'package)
   (abbreviation :initform 'IR2F)
   (comment :initform (list
     "Conversion of integration results to top-level expressions"
     "This package allows a sum of logs over the roots of a polynomial"
     "to be expressed as explicit logarithms and arc tangents, provided"
     "that the indexing polynomial can be factored into quadratics."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResultToFunction|
  (progn
    (push '|IntegrationResultToFunction| *Packages*)
    (make-instance '|IntegrationResultToFunctionType|)))
```

———-

1.84.37 IntegrationTools

— defclass IntegrationToolsType —

```
(defclass |IntegrationToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegrationTools")
   (marker :initform 'package)
   (abbreviation :initform 'INTTOOLS)
   (comment :initform (list
     "Tools for the integrator"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationTools|
  (progn
    (push '|IntegrationTools| *Packages*)
    (make-instance '|IntegrationToolsType|)))
```

————-

1.84.38 InternalPrintPackage

— defclass InternalPrintPackageType —

```
(defclass |InternalPrintPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InternalPrintPackage")
   (marker :initform 'package)
   (abbreviation :initform 'IPRNTPK)
   (comment :initform (list
     "A package to print strings without line-feed nor carriage-return."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InternalPrintPackage|
  (progn
    (push '|InternalPrintPackage| *Packages*)
    (make-instance '|InternalPrintPackageType|)))
```

————-

1.84.39 InternalRationalUnivariateRepresentationPackage

— defclass InternalRationalUnivariateRepresentationPackageType —

```
(defclass |InternalRationalUnivariateRepresentationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InternalRationalUnivariateRepresentationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'IRURPK)
   (comment :initform nil)
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |InternalRationalUnivariateRepresentationPackage|
  (progn
    (push '|InternalRationalUnivariateRepresentationPackage| *Packages*)
    (make-instance '|InternalRationalUnivariateRepresentationPackageType|)))
```

———-

1.84.40 InterpolateFormsPackage

— defclass InterpolateFormsPackageType —

```
(defclass |InterpolateFormsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InterpolateFormsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'INTFRSP)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InterpolateFormsPackage|
  (progn
    (push '|InterpolateFormsPackage| *Packages*)
    (make-instance '|InterpolateFormsPackageType|)))
```

———-

1.84.41 IntersectionDivisorPackage

— defclass IntersectionDivisorPackageType —

```
(defclass |IntersectionDivisorPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntersectionDivisorPackage")
   (marker :initform 'package)
   (abbreviation :initform 'INTDIVP)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntersectionDivisorPackage|
  (progn
    (push '|IntersectionDivisorPackage| *Packages*)
    (make-instance '|IntersectionDivisorPackageType|)))
```

———-

1.84.42 IrredPolyOverFiniteField

— defclass IrredPolyOverFiniteFieldType —

```
(defclass |IrredPolyOverFiniteFieldType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IrredPolyOverFiniteField")
   (marker :initform 'package)
   (abbreviation :initform 'IRREDFFX)
   (comment :initform (list
     "This package exports the function generateIrredPoly that computes"
     "a monic irreducible polynomial of degree n over a finite field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IrredPolyOverFiniteField|
  (progn
    (push '|IrredPolyOverFiniteField| *Packages*)
    (make-instance '|IrredPolyOverFiniteFieldType|)))
```

———-

1.84.43 IrrRepSymNatPackage

— defclass IrrRepSymNatPackageType —

```
(defclass |IrrRepSymNatPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IrrRepSymNatPackage")
   (marker :initform 'package)
   (abbreviation :initform 'IRSN)
   (comment :initform (list
     "IrrRepSymNatPackage contains functions for computing"
     "the ordinary irreducible representations of symmetric groups on"
     "n letters {1,2,...,n} in Young's natural form and their dimensions."
     "These representations can be labelled by number partitions of n,"
     "a weakly decreasing sequence of integers summing up to n, for"
     "example, [3,3,3,1] labels an irreducible representation for n equals 10."
     "Note that whenever a List Integer appears in a signature,"
     "a partition required."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |IrrRepSymNatPackage|
  (progn
    (push '|IrrRepSymNatPackage| *Packages*)
    (make-instance '|IrrRepSymNatPackageType|)))
```

————-

1.84.44 InverseLaplaceTransform

— defclass InverseLaplaceTransformType —

```
(defclass |InverseLaplaceTransformType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InverseLaplaceTransform")
   (marker :initform 'package)
   (abbreviation :initform 'INVLAPLA)
   (comment :initform (list
     "This package computes the inverse Laplace Transform."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InverseLaplaceTransform|
  (progn
    (push '|InverseLaplaceTransform| *Packages*)
    (make-instance '|InverseLaplaceTransformType|)))
```

————-

1.85 K

1.85.1 KernelFunctions2

— defclass KernelFunctions2Type —

```
(defclass |KernelFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "KernelFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'KERNEL2)
   (comment :initform (list
     "This package exports some auxiliary functions on kernels"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |KernelFunctions2|
  (progn
    (push '|KernelFunctions2| *Packages*)
    (make-instance '|KernelFunctions2Type|)))
```

———-

1.85.2 Kovacic

— defclass KovacicType —

```
(defclass |KovacicType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Kovacic")
   (marker :initform 'package)
   (abbreviation :initform 'KOVACIC)
   (comment :initform (list
     "Kovacic provides a modified Kovacic's algorithm for"
     "solving explicitely irreducible 2nd order linear ordinary"
     "differential equations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Kovacic|
  (progn
    (push '|Kovacic| *Packages*)
    (make-instance '|KovacicType|)))
```

———-

1.86 L

1.86.1 LaplaceTransform

— defclass LaplaceTransformType —

```
(defclass |LaplaceTransformType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LaplaceTransform")
   (marker :initform 'package)
   (abbreviation :initform 'LAPLACE)
   (comment :initform (list
     "This package computes the forward Laplace Transform."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |LaplaceTransform|
  (progn
    (push '|LaplaceTransform| *Packages*)
    (make-instance '|LaplaceTransformType|)))
```

———-

1.86.2 LazardSetSolvingPackage

— defclass LazardSetSolvingPackageType —

```
(defclass |LazardSetSolvingPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LazardSetSolvingPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LAZM3PK)
   (comment :initform (list
     "A package for solving polynomial systems by means of Lazard triangular"
     "sets. This package provides two operations. One for solving in the sense"
     "of the regular zeros, and the other for solving in the sense of"
     "the Zariski closure. Both produce square-free regular sets."
     "Moreover, the decompositions do not contain any redundant component."
     "However, only zero-dimensional regular sets are normalized, since"
     "normalization may be time consumming in positive dimension."
     "The decomposition process is that of [2]."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LazardSetSolvingPackage|
  (progn
    (push '|LazardSetSolvingPackage| *Packages*)
    (make-instance '|LazardSetSolvingPackageType|)))
```

———-

1.86.3 LeadingCoefDetermination

— defclass LeadingCoefDeterminationType —

```
(defclass |LeadingCoefDeterminationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LeadingCoefDetermination")
   (marker :initform 'package)
   (abbreviation :initform 'LEADCDET)
   (comment :initform (list
     "Package for leading coefficient determination in the lifting step."
     "Package working for every R euclidean with property 'F'."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeadingCoefDetermination|
  (progn
    (push '|LeadingCoefDetermination| *Packages*)
    (make-instance '|LeadingCoefDeterminationType|)))
```

———-

1.86.4 LexTriangularPackage

— defclass LexTriangularPackageType —

```
(defclass |LexTriangularPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LexTriangularPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LEXTRIPK)
   (comment :initform (list
     "A package for solving polynomial systems with finitely many solutions."
     "The decompositions are given by means of regular triangular sets."
     "The computations use lexicographical Groebner bases."
     "The main operations are lexTriangular"
     "and squareFreeLexTriangular. The second one provide decompositions by"
     "means of square-free regular triangular sets."
     "Both are based on the lexTriangular method described in [1]."
     "They differ from the algorithm described in [2] by the fact that"
     "multiplicities of the roots are not kept."
     "With the squareFreeLexTriangular operation all multiciplities are removed."
     "With the other operation some multiciplities may remain. Both operations"
     "admit an optional argument to produce normalized triangular sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LexTriangularPackage|
  (progn
    (push '|LexTriangularPackage| *Packages*)
    (make-instance '|LexTriangularPackageType|)))
```

———-

1.86.5 LinearDependence

— defclass LinearDependenceType —

```
(defclass |LinearDependenceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearDependence")
   (marker :initform 'package)
   (abbreviation :initform 'LINDEP)
   (comment :initform (list
     "Test for linear dependence."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearDependence|
  (progn
    (push '|LinearDependence| *Packages*)
    (make-instance '|LinearDependenceType|)))
```

———-

1.86.6 LinearOrdinaryDifferentialOperatorFactorizer

— defclass LinearOrdinaryDifferentialOperatorFactorizerType —

```
(defclass |LinearOrdinaryDifferentialOperatorFactorizerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearOrdinaryDifferentialOperatorFactorizer")
   (marker :initform 'package)
   (abbreviation :initform 'LODOF)
   (comment :initform (list
     "LinearOrdinaryDifferentialOperatorFactorizer provides a"
     "factorizer for linear ordinary differential operators whose coefficients"
     "are rational functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperatorFactorizer|
  (progn
    (push '|LinearOrdinaryDifferentialOperatorFactorizer| *Packages*)
    (make-instance '|LinearOrdinaryDifferentialOperatorFactorizerType|)))
```

———-

1.86.7 LinearOrdinaryDifferentialOperatorsOps

— defclass LinearOrdinaryDifferentialOperatorsOpsType —

```
(defclass |LinearOrdinaryDifferentialOperatorsOpsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearOrdinaryDifferentialOperatorsOps")
   (marker :initform 'package)
   (abbreviation :initform 'LODOOPS)
   (comment :initform (list
     "LinearOrdinaryDifferentialOperatorsOps provides symmetric"
     "products and sums for linear ordinary differential operators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperatorsOps|
  (progn
    (push '|LinearOrdinaryDifferentialOperatorsOps| *Packages*)
    (make-instance '|LinearOrdinaryDifferentialOperatorsOpsType|)))
```

———-

1.86.8 LinearPolynomialEquationByFractions

— defclass LinearPolynomialEquationByFractionsType —

```
(defclass |LinearPolynomialEquationByFractionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearPolynomialEquationByFractions")
   (marker :initform 'package)
   (abbreviation :initform 'LPEFRAC)
   (comment :initform (list
     "Given a PolynomialFactorizationExplicit ring, this package"
     "provides a defaulting rule for the solveLinearPolynomialEquation"
     "operation, by moving into the field of fractions, and solving it there"
     "via the multiEuclidean operation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearPolynomialEquationByFractions|
  (progn
    (push '|LinearPolynomialEquationByFractions| *Packages*)
    (make-instance '|LinearPolynomialEquationByFractionsType|)))
```

———-

1.86.9 LinearSystemFromPowerSeriesPackage

— defclass LinearSystemFromPowerSeriesPackageType —

```
(defclass |LinearSystemFromPowerSeriesPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearSystemFromPowerSeriesPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LISYSER)
   (comment :initform (list
     "Part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemFromPowerSeriesPackage|
  (progn
    (push '|LinearSystemFromPowerSeriesPackage| *Packages*)
    (make-instance '|LinearSystemFromPowerSeriesPackageType|)))
```

———–

1.86.10 LinearSystemMatrixPackage

— defclass LinearSystemMatrixPackageType —

```
(defclass |LinearSystemMatrixPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearSystemMatrixPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LSMP)
   (comment :initform (list
     "This package solves linear system in the matrix form AX = B."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemMatrixPackage|
  (progn
    (push '|LinearSystemMatrixPackage| *Packages*)
    (make-instance '|LinearSystemMatrixPackageType|)))
```

———–

1.86.11 LinearSystemMatrixPackage1

— defclass LinearSystemMatrixPackage1Type —

```
(defclass |LinearSystemMatrixPackage1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearSystemMatrixPackage1")
   (marker :initform 'package)
   (abbreviation :initform 'LSMP1)
   (comment :initform (list
     "This package solves linear system in the matrix form AX = B."
     "It is essentially a particular instantiation of the package"
     "LinearSystemMatrixPackage for Matrix and Vector. This"
     "package's existence makes it easier to use solve in the"
     "AXIOM interpreter."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemMatrixPackage1|
  (progn
    (push '|LinearSystemMatrixPackage1| *Packages*)
    (make-instance '|LinearSystemMatrixPackage1Type|)))
```

———-

1.86.12 LinearSystemPolynomialPackage

— defclass LinearSystemPolynomialPackageType —

```
(defclass |LinearSystemPolynomialPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearSystemPolynomialPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LSPP)
   (comment :initform (list
     "This package finds the solutions of linear systems presented as a"
     "list of polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemPolynomialPackage|
  (progn
    (push '|LinearSystemPolynomialPackage| *Packages*)
    (make-instance '|LinearSystemPolynomialPackageType|)))
```

———-

1.86.13 LinGroebnerPackage

— defclass LinGroebnerPackageType —

```
(defclass |LinGroebnerPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinGroebnerPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LGROBP)
   (comment :initform (list
     "Given a Groebner basis B with respect to the total degree ordering for"
     "a zero-dimensional ideal I, compute"
     "a Groebner basis with respect to the lexicographical ordering by using"
     "linear algebra."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinGroebnerPackage|
  (progn
    (push '|LinGroebnerPackage| *Packages*)
    (make-instance '|LinGroebnerPackageType|)))
```

———-

1.86.14 LinesOpPack

— defclass LinesOpPackType —

```
(defclass |LinesOpPackType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinesOpPack")
   (marker :initform 'package)
   (abbreviation :initform 'LOP)
   (comment :initform (list
     "A package that exports several linear algebra operations over lines"
     "of matrices. Part of the PAFF package."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinesOpPack|
  (progn
    (push '|LinesOpPack| *Packages*)
    (make-instance '|LinesOpPackType|)))
```

———-

1.86.15 LiouvillianFunction

— defclass LiouvillianFunctionType —

```
(defclass |LiouvillianFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LiouvillianFunction")
   (marker :initform 'package)
   (abbreviation :initform 'LF)
   (comment :initform (list
     "This package provides liouvillian functions over an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LiouvillianFunction|
  (progn
    (push '|LiouvillianFunction| *Packages*)
    (make-instance '|LiouvillianFunctionType|)))
```

———-

1.86.16 ListFunctions2

— defclass ListFunctions2Type —

```
(defclass |ListFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ListFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'LIST2)
   (comment :initform (list
     "ListFunctions2 implements utility functions that"
     "operate on two kinds of lists, each with a possibly different"
     "type of element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListFunctions2|
  (progn
    (push '|ListFunctions2| *Packages*)
    (make-instance '|ListFunctions2Type|)))
```

———-

1.86.17 ListFunctions3

— defclass ListFunctions3Type —

```
(defclass |ListFunctions3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ListFunctions3")
   (marker :initform 'package)
   (abbreviation :initform 'LIST3)
   (comment :initform (list
     "ListFunctions3 implements utility functions that"
     "operate on three kinds of lists, each with a possibly different"
```

```
    "type of element."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListFunctions3|
  (progn
    (push '|ListFunctions3| *Packages*)
    (make-instance '|ListFunctions3Type|)))
```

———-

1.86.18 ListToMap

— defclass ListToMapType —

```
(defclass |ListToMapType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ListToMap")
   (marker :initform 'package)
   (abbreviation :initform 'LIST2MAP)
   (comment :initform (list
     "ListToMap allows mappings to be described by a pair of"
     "lists of equal lengths.  The image of an element x,"
     "which appears in position n in the first list, is then"
     "the nth element of the second list.  A default value or"
     "default function can be specified to be used when x"
     "does not appear in the first list.  In the absence of defaults,"
     "an error will occur in that case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListToMap|
  (progn
    (push '|ListToMap| *Packages*)
    (make-instance '|ListToMapType|)))
```

———-

1.86.19 LocalParametrizationOfSimplePointPackage

— defclass LocalParametrizationOfSimplePointPackageType —

```
(defclass |LocalParametrizationOfSimplePointPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LocalParametrizationOfSimplePointPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LPARSPT)
   (comment :initform (list
     "This package is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LocalParametrizationOfSimplePointPackage|
  (progn
    (push '|LocalParametrizationOfSimplePointPackage| *Packages*)
    (make-instance '|LocalParametrizationOfSimplePointPackageType|)))
```

———-

1.87 M

1.87.1 MakeBinaryCompiledFunction

— defclass MakeBinaryCompiledFunctionType —

```
(defclass |MakeBinaryCompiledFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MakeBinaryCompiledFunction")
   (marker :initform 'package)
   (abbreviation :initform 'MKBCFUNC)
   (comment :initform (list
     "Tools and transforms for making compiled functions from"
     "top-level expressions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeBinaryCompiledFunction|
  (progn
    (push '|MakeBinaryCompiledFunction| *Packages*)
    (make-instance '|MakeBinaryCompiledFunctionType|)))
```

———-

1.87.2 MakeFloatCompiledFunction

— defclass MakeFloatCompiledFunctionType —

```
(defclass |MakeFloatCompiledFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MakeFloatCompiledFunction")
   (marker :initform 'package)
   (abbreviation :initform 'MKFLCFN)
   (comment :initform (list
     "Tools for making compiled functions from top-level expressions"
     "MakeFloatCompiledFunction transforms top-level objects into"
     "compiled Lisp functions whose arguments are Lisp floats."
     "This by-passes the Axiom compiler and interpreter,"
     "thereby gaining several orders of magnitude."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeFloatCompiledFunction|
  (progn
    (push '|MakeFloatCompiledFunction| *Packages*)
    (make-instance '|MakeFloatCompiledFunctionType|)))
```

———-

1.87.3 MakeFunction

— defclass MakeFunctionType —

```
(defclass |MakeFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MakeFunction")
   (marker :initform 'package)
   (abbreviation :initform 'MKFUNC)
   (comment :initform (list
     "Tools for making interpreter functions from top-level expressions"
     "Transforms top-level objects into interpreter functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeFunction|
  (progn
    (push '|MakeFunction| *Packages*)
    (make-instance '|MakeFunctionType|)))
```

———-

1.87.4 MakeRecord

— defclass MakeRecordType —

```
(defclass |MakeRecordType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MakeRecord")
   (marker :initform 'package)
   (abbreviation :initform 'MKRECORD)
   (comment :initform (list
     "MakeRecord is used internally by the interpreter to create record"
     "types which are used for doing parallel iterations on streams."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeRecord|
  (progn
    (push '|MakeRecord| *Packages*)
    (make-instance '|MakeRecordType|)))
```

———-

1.87.5 MakeUnaryCompiledFunction

— defclass MakeUnaryCompiledFunctionType —

```
(defclass |MakeUnaryCompiledFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MakeUnaryCompiledFunction")
   (marker :initform 'package)
   (abbreviation :initform 'MKUCFUNC)
   (comment :initform (list
     "Tools for making compiled functions from top-level expressions"
     "Transforms top-level objects into compiled functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeUnaryCompiledFunction|
  (progn
    (push '|MakeUnaryCompiledFunction| *Packages*)
    (make-instance '|MakeUnaryCompiledFunctionType|)))
```

———-

1.87.6 MappingPackageInternalHacks1

— defclass MappingPackageInternalHacks1Type —

```
(defclass |MappingPackageInternalHacks1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackageInternalHacks1")
   (marker :initform 'package)
   (abbreviation :initform 'MAPHACK1)
   (comment :initform (list
     "Various Currying operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackageInternalHacks1|
  (progn
    (push '|MappingPackageInternalHacks1| *Packages*)
    (make-instance '|MappingPackageInternalHacks1Type|)))
```

———–

1.87.7 MappingPackageInternalHacks2

— defclass MappingPackageInternalHacks2Type —

```
(defclass |MappingPackageInternalHacks2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackageInternalHacks2")
   (marker :initform 'package)
   (abbreviation :initform 'MAPHACK2)
   (comment :initform (list
     "Various Currying operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackageInternalHacks2|
  (progn
    (push '|MappingPackageInternalHacks2| *Packages*)
    (make-instance '|MappingPackageInternalHacks2Type|)))
```

———–

1.87.8 MappingPackageInternalHacks3

— defclass MappingPackageInternalHacks3Type —

```
(defclass |MappingPackageInternalHacks3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackageInternalHacks3")
   (marker :initform 'package)
   (abbreviation :initform 'MAPHACK3)
   (comment :initform (list
     "Various Currying operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackageInternalHacks3|
  (progn
    (push '|MappingPackageInternalHacks3| *Packages*)
    (make-instance '|MappingPackageInternalHacks3Type|)))
```

———-

1.87.9 MappingPackage1

— defclass MappingPackage1Type —

```
(defclass |MappingPackage1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackage1")
   (marker :initform 'package)
   (abbreviation :initform 'MAPPKG1)
   (comment :initform (list
     "Various Currying operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackage1|
  (progn
    (push '|MappingPackage1| *Packages*)
    (make-instance '|MappingPackage1Type|)))
```

———-

1.87.10 MappingPackage2

— defclass MappingPackage2Type —

```
(defclass |MappingPackage2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackage2")
   (marker :initform 'package)
   (abbreviation :initform 'MAPPKG2)
   (comment :initform (list
     "Various Currying operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |MappingPackage2|
  (progn
    (push '|MappingPackage2| *Packages*)
    (make-instance '|MappingPackage2Type|)))
```

———-

1.87.11 MappingPackage3

— defclass MappingPackage3Type —

```
(defclass |MappingPackage3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackage3")
   (marker :initform 'package)
   (abbreviation :initform 'MAPPKG3)
   (comment :initform (list
     "Various Currying operations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackage3|
  (progn
    (push '|MappingPackage3| *Packages*)
    (make-instance '|MappingPackage3Type|)))
```

———-

1.87.12 MappingPackage4

— defclass MappingPackage4Type —

```
(defclass |MappingPackage4Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackage4")
   (marker :initform 'package)
   (abbreviation :initform 'MAPPKG4)
   (comment :initform (list
     "Functional Composition."
     "Given functions f and g, returns the applicable closure"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackage4|
  (progn
    (push '|MappingPackage4| *Packages*)
    (make-instance '|MappingPackage4Type|)))
```

———-

1.87.13 MatrixCategoryFunctions2

— defclass MatrixCategoryFunctions2Type —

```
(defclass |MatrixCategoryFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MatrixCategoryFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'MATCAT2)
   (comment :initform (list
     "MatrixCategoryFunctions2 provides functions between two matrix"
     "domains.  The functions provided are map and reduce."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixCategoryFunctions2|
  (progn
    (push '|MatrixCategoryFunctions2| *Packages*)
    (make-instance '|MatrixCategoryFunctions2Type|)))
```

———-

1.87.14 MatrixCommonDenominator

— defclass MatrixCommonDenominatorType —

```
(defclass |MatrixCommonDenominatorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MatrixCommonDenominator")
   (marker :initform 'package)
   (abbreviation :initform 'MCDEN)
   (comment :initform (list
     "MatrixCommonDenominator provides functions to"
     "compute the common denominator of a matrix of elements of the"
     "quotient field of an integral domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixCommonDenominator|
  (progn
    (push '|MatrixCommonDenominator| *Packages*)
    (make-instance '|MatrixCommonDenominatorType|)))
```

———-

1.87.15 MatrixLinearAlgebraFunctions

— defclass MatrixLinearAlgebraFunctionsType —

```
(defclass |MatrixLinearAlgebraFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MatrixLinearAlgebraFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'MATLIN)
   (comment :initform (list
     "MatrixLinearAlgebraFunctions provides functions to compute"
     "inverses and canonical forms."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixLinearAlgebraFunctions|
  (progn
    (push '|MatrixLinearAlgebraFunctions| *Packages*)
    (make-instance '|MatrixLinearAlgebraFunctionsType|)))
```

———-

1.87.16 MatrixManipulation

— defclass MatrixManipulationType —

```
(defclass |MatrixManipulationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MatrixManipulation")
   (marker :initform 'package)
   (abbreviation :initform 'MAMA)
   (comment :initform (list
     "Some functions for manipulating (dense) matrices."
     "Supported are various kinds of slicing, splitting and stacking of"
     "matrices. The functions resemble operations often used in numerical"
     "linear algebra algorithms."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixManipulation|
  (progn
    (push '|MatrixManipulation| *Packages*)
    (make-instance '|MatrixManipulationType|)))
```

———-

1.87.17 MergeThing

— defclass MergeThingType —

```
(defclass |MergeThingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MergeThing")
   (marker :initform 'package)
   (abbreviation :initform 'MTHING)
   (comment :initform (list
     "This package exports tools for merging lists"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MergeThing|
  (progn
    (push '|MergeThing| *Packages*)
    (make-instance '|MergeThingType|)))
```

———-

1.87.18 MeshCreationRoutinesForThreeDimensions

— defclass MeshCreationRoutinesForThreeDimensionsType —

```
(defclass |MeshCreationRoutinesForThreeDimensionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MeshCreationRoutinesForThreeDimensions")
   (marker :initform 'package)
   (abbreviation :initform 'MESH)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MeshCreationRoutinesForThreeDimensions|
  (progn
    (push '|MeshCreationRoutinesForThreeDimensions| *Packages*)
    (make-instance '|MeshCreationRoutinesForThreeDimensionsType|)))
```

———-

1.87.19 ModularDistinctDegreeFactorizer

— defclass ModularDistinctDegreeFactorizerType —

```
(defclass |ModularDistinctDegreeFactorizerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ModularDistinctDegreeFactorizer")
   (marker :initform 'package)
   (abbreviation :initform 'MDDFACT)
   (comment :initform (list
     "This package supports factorization and gcds"
     "of univariate polynomials over the integers modulo different"
     "primes. The inputs are given as polynomials over the integers"
     "with the prime passed explicitly as an extra argument."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularDistinctDegreeFactorizer|
  (progn
    (push '|ModularDistinctDegreeFactorizer| *Packages*)
    (make-instance '|ModularDistinctDegreeFactorizerType|)))
```

————-

1.87.20 ModularHermitianRowReduction

— defclass ModularHermitianRowReductionType —

```
(defclass |ModularHermitianRowReductionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ModularHermitianRowReduction")
   (marker :initform 'package)
   (abbreviation :initform 'MHROWRED)
   (comment :initform (list
     "Modular hermitian row reduction."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularHermitianRowReduction|
  (progn
    (push '|ModularHermitianRowReduction| *Packages*)
    (make-instance '|ModularHermitianRowReductionType|)))
```

————-

1.87.21 MonoidRingFunctions2

— defclass MonoidRingFunctions2Type —

```
(defclass |MonoidRingFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MonoidRingFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'MRF2)
   (comment :initform (list
     "MonoidRingFunctions2 implements functions between"
     "two monoid rings defined with the same monoid over different rings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonoidRingFunctions2|
  (progn
    (push '|MonoidRingFunctions2| *Packages*)
    (make-instance '|MonoidRingFunctions2Type|)))
```

————-

1.87.22 MonomialExtensionTools

— defclass MonomialExtensionToolsType —

```
(defclass |MonomialExtensionToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MonomialExtensionTools")
   (marker :initform 'package)
   (abbreviation :initform 'MONOTOOL)
   (comment :initform (list
     "Tools for handling monomial extensions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonomialExtensionTools|
  (progn
    (push '|MonomialExtensionTools| *Packages*)
    (make-instance '|MonomialExtensionToolsType|)))
```

————-

1.87.23 MoreSystemCommands

— defclass MoreSystemCommandsType —

```
(defclass |MoreSystemCommandsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MoreSystemCommands")
   (marker :initform 'package)
   (abbreviation :initform 'MSYSCMD)
   (comment :initform (list
     "MoreSystemCommands implements an interface with the"
     "system command facility. These are the commands that are issued"
     "from source files or the system interpreter and they start with"
     "a close parenthesis, for example, the 'what' commands."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MoreSystemCommands|
  (progn
    (push '|MoreSystemCommands| *Packages*)
    (make-instance '|MoreSystemCommandsType|)))
```

———-

1.87.24 MPolyCatPolyFactorizer

— defclass MPolyCatPolyFactorizerType —

```
(defclass |MPolyCatPolyFactorizerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MPolyCatPolyFactorizer")
   (marker :initform 'package)
   (abbreviation :initform 'MPCPF)
   (comment :initform (list
     "This package exports a factor operation for multivariate polynomials"
     "with coefficients which are polynomials over"
     "some ring R over which we can factor. It is used internally by packages"
     "such as the solve package which need to work with polynomials in a specific"
     "set of variables with coefficients which are polynomials in all the other"
     "variables."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MPolyCatPolyFactorizer|
  (progn
    (push '|MPolyCatPolyFactorizer| *Packages*)
    (make-instance '|MPolyCatPolyFactorizerType|)))
```

———-

1.87.25 MPolyCatRationalFunctionFactorizer

— defclass MPolyCatRationalFunctionFactorizerType —

```
(defclass |MPolyCatRationalFunctionFactorizerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MPolyCatRationalFunctionFactorizer")
   (marker :initform 'package)
   (abbreviation :initform 'MPRFF)
   (comment :initform (list
     "This package exports a factor operation for multivariate polynomials"
     "with coefficients which are rational functions over"
     "some ring R over which we can factor. It is used internally by packages"
     "such as primary decomposition which need to work with polynomials"
     "with rational function coefficients, themselves fractions of"
     "polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MPolyCatRationalFunctionFactorizer|
  (progn
    (push '|MPolyCatRationalFunctionFactorizer| *Packages*)
    (make-instance '|MPolyCatRationalFunctionFactorizerType|)))
```

———-

1.87.26 MPolyCatFunctions2

— defclass MPolyCatFunctions2Type —

```
(defclass |MPolyCatFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MPolyCatFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'MPC2)
   (comment :initform (list
     "Utilities for MPolyCat"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MPolyCatFunctions2|
  (progn
    (push '|MPolyCatFunctions2| *Packages*)
    (make-instance '|MPolyCatFunctions2Type|)))
```

———-

1.87.27 MPolyCatFunctions3

— defclass MPolyCatFunctions3Type —

```
(defclass |MPolyCatFunctions3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MPolyCatFunctions3")
   (marker :initform 'package)
   (abbreviation :initform 'MPC3)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MPolyCatFunctions3|
  (progn
    (push '|MPolyCatFunctions3| *Packages*)
    (make-instance '|MPolyCatFunctions3Type|)))
```

——–

1.87.28 MRationalFactorize

— defclass MRationalFactorizeType —

```
(defclass |MRationalFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MRationalFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'MRATFAC)
   (comment :initform (list
     "MRationalFactorize contains the factor function for multivariate"
     "polynomials over the quotient field of a ring R such that the package"
     "MultivariateFactorize can factor multivariate polynomials over R."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MRationalFactorize|
  (progn
    (push '|MRationalFactorize| *Packages*)
    (make-instance '|MRationalFactorizeType|)))
```

——–

1.87.29 MultFiniteFactorize

— defclass MultFiniteFactorizeType —

```
(defclass |MultFiniteFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MultFiniteFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'MFINFACT)
   (comment :initform (list
     "Package for factorization of multivariate polynomials over finite fields."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultFiniteFactorize|
  (progn
    (push '|MultFiniteFactorize| *Packages*)
    (make-instance '|MultFiniteFactorizeType|)))
```

———-

1.87.30 MultipleMap

— defclass MultipleMapType —

```
(defclass |MultipleMapType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MultipleMap")
   (marker :initform 'package)
   (abbreviation :initform 'MMAP)
   (comment :initform (list
     "Lifting of a map through 2 levels of polynomials"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultipleMap|
  (progn
    (push '|MultipleMap| *Packages*)
    (make-instance '|MultipleMapType|)))
```

———-

1.87.31 MultiVariableCalculusFunctions

— defclass MultiVariableCalculusFunctionsType —

```
(defclass |MultiVariableCalculusFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MultiVariableCalculusFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'MCALCFN)
   (comment :initform (list
     "MultiVariableCalculusFunctions Package provides several"
     "functions for multivariable calculus."
     "These include gradient, hessian and jacobian, divergence and laplacian."
     "Various forms for banded and sparse storage of matrices are included."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultiVariableCalculusFunctions|
  (progn
    (push '|MultiVariableCalculusFunctions| *Packages*)
    (make-instance '|MultiVariableCalculusFunctionsType|)))
```

————-

1.87.32 MultivariateFactorize

— defclass MultivariateFactorizeType —

```
(defclass |MultivariateFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MultivariateFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'MULTFACT)
   (comment :initform (list
     "This is the top level package for doing multivariate factorization"
     "over basic domains like Integer or Fraction Integer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariateFactorize|
  (progn
    (push '|MultivariateFactorize| *Packages*)
    (make-instance '|MultivariateFactorizeType|)))
```

————-

1.87.33 MultivariateLifting

— defclass MultivariateLiftingType —

```
(defclass |MultivariateLiftingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MultivariateLifting")
   (marker :initform 'package)
   (abbreviation :initform 'MLIFT)
   (comment :initform (list
     "This package provides the functions for the multivariate 'lifting', using"
     "an algorithm of Paul Wang."
     "This package will work for every euclidean domain R which has property"
     "F, there exists a factor operation in R[x]."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariateLifting|
  (progn
    (push '|MultivariateLifting| *Packages*)
    (make-instance '|MultivariateLiftingType|)))
```

———-

1.87.34 MultivariateSquareFree

— defclass MultivariateSquareFreeType —

```
(defclass |MultivariateSquareFreeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MultivariateSquareFree")
   (marker :initform 'package)
   (abbreviation :initform 'MULTSQFR)
   (comment :initform (list
     "This package provides the functions for the computation of the square"
     "free decomposition of a multivariate polynomial."
     "It uses the package GenExEuclid for the resolution of"
     "the equation Af + Bg = h and its generalization to n polynomials"
     "over an integral domain and the package MultivariateLifting"
     "for the 'multivariate' lifting."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariateSquareFree|
  (progn
    (push '|MultivariateSquareFree| *Packages*)
    (make-instance '|MultivariateSquareFreeType|)))
```

———-

1.88 N

1.88.1 NagEigenPackage

— defclass NagEigenPackageType —

```
(defclass |NagEigenPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagEigenPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGF02)
   (comment :initform (list
     "This package uses the NAG Library to compute"
     "     eigenvalues and eigenvectors of a matrix"
     "     eigenvalues and eigenvectors of generalized matrix"
     "eigenvalue problems"
     "     singular values and singular vectors of a matrix."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagEigenPackage|
  (progn
    (push '|NagEigenPackage| *Packages*)
    (make-instance '|NagEigenPackageType|)))
```

———-

1.88.2 NagFittingPackage

— defclass NagFittingPackageType —

```
(defclass |NagFittingPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagFittingPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGE02)
   (comment :initform (list
     "This package uses the NAG Library to find a"
     "function which approximates a set of data points. Typically the"
     "data contain random errors, as of experimental measurement, which"
     "need to be smoothed out. To seek an approximation to the data, it"
     "is first necessary to specify for the approximating function a"
     "mathematical form (a polynomial, for example) which contains a"
     "number of unspecified coefficients: the appropriate fitting"
     "routine then derives for the coefficients the values which"
     "provide the best fit of that particular form. The package deals"
     "mainly with curve and surface fitting (fitting with"
     "functions of one and of two variables) when a polynomial or a"
     "cubic spline is used as the fitting function, since these cover"
     "the most common needs. However, fitting with other functions"
     "and/or more variables can be undertaken by means of general"
     "linear or nonlinear routines (some of which are contained in"
     "other packages) depending on whether the coefficients in the"
     "function occur linearly or nonlinearly. Cases where a graph"
     "rather than a set of data points is given can be treated simply"
     "by first reading a suitable set of points from the graph."
     "The package also contains routines for evaluating,"
     "differentiating and integrating polynomial and spline curves and"
     "surfaces, once the numerical values of their coefficients have"
     "been determined."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagFittingPackage|
  (progn
    (push '|NagFittingPackage| *Packages*)
    (make-instance '|NagFittingPackageType|)))
```

———-

1.88.3 NagLinearEquationSolvingPackage

— defclass NagLinearEquationSolvingPackageType —

```
(defclass |NagLinearEquationSolvingPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagLinearEquationSolvingPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGF04)
   (comment :initform (list
     "This package uses the NAG Library to solve the matrix equation"
     "     AX=B, where B"
     "may be a single vector or a matrix of multiple right-hand sides."
     "The matrix A may be real, complex, symmetric, Hermitian positive-"
     "definite, or sparse. It may also be rectangular, in which case a"
     "least-squares solution is obtained."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagLinearEquationSolvingPackage|
  (progn
    (push '|NagLinearEquationSolvingPackage| *Packages*)
    (make-instance '|NagLinearEquationSolvingPackageType|)))
```

———-

1.88.4 NAGLinkSupportPackage

— defclass NAGLinkSupportPackageType —

```
(defclass |NAGLinkSupportPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NAGLinkSupportPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGSP)
   (comment :initform (list
     "Support functions for the NAG Library Link functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NAGLinkSupportPackage|
  (progn
    (push '|NAGLinkSupportPackage| *Packages*)
    (make-instance '|NAGLinkSupportPackageType|)))
```

———-

1.88.5 NagIntegrationPackage

— defclass NagIntegrationPackageType —

```
(defclass |NagIntegrationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagIntegrationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGD01)
   (comment :initform (list
     "This package uses the NAG Library to calculate the numerical value of"
     "definite integrals in one or more dimensions and to evaluate"
     "weights and abscissae of integration rules."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagIntegrationPackage|
  (progn
    (push '|NagIntegrationPackage| *Packages*)
    (make-instance '|NagIntegrationPackageType|)))
```

———-

1.88.6 NagInterpolationPackage

— defclass NagInterpolationPackageType —

```
(defclass |NagInterpolationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagInterpolationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGE01)
   (comment :initform (list
     "This package uses the NAG Library to calculate the interpolation of a"
     "function of one or two variables. When provided with the value of the"
     "function (and possibly one or more of its lowest-order"
     "derivatives) at each of a number of values of the variable(s),"
     "the routines provide either an interpolating function or an"
     "interpolated value. For some of the interpolating functions,"
     "there are supporting routines to evaluate, differentiate or"
     "integrate them."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagInterpolationPackage|
  (progn
    (push '|NagInterpolationPackage| *Packages*)
    (make-instance '|NagInterpolationPackageType|)))
```

———-

1.88.7 NagLapack

— defclass NagLapackType —

```
(defclass |NagLapackType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagLapack")
   (marker :initform 'package)
   (abbreviation :initform 'NAGF07)
   (comment :initform (list
     "This package uses the NAG Library to compute matrix"
     "factorizations, and to solve systems of linear equations"
     "following the matrix factorizations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagLapack|
  (progn
    (push '|NagLapack| *Packages*)
    (make-instance '|NagLapackType|)))
```

———-

1.88.8 NagMatrixOperationsPackage

— defclass NagMatrixOperationsPackageType —

```
(defclass |NagMatrixOperationsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagMatrixOperationsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGF01)
   (comment :initform (list
     "This package uses the NAG Library to provide facilities for matrix"
     "factorizations and associated transformations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagMatrixOperationsPackage|
  (progn
    (push '|NagMatrixOperationsPackage| *Packages*)
    (make-instance '|NagMatrixOperationsPackageType|)))
```

———-

1.88.9 NagOptimisationPackage

— defclass NagOptimisationPackageType —

```
(defclass |NagOptimisationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagOptimisationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGE04)
   (comment :initform (list
     "This package uses the NAG Library to perform optimization."
     "An optimization problem involves minimizing a function (called"
     "the objective function) of several variables, possibly subject to"
     "restrictions on the values of the variables defined by a set of"
     "constraint functions. The routines in the NAG Foundation Library"
     "are concerned with function minimization only, since the problem"
     "of maximizing a given function can be transformed into a"
     "minimization problem simply by multiplying the function by -1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagOptimisationPackage|
  (progn
    (push '|NagOptimisationPackage| *Packages*)
    (make-instance '|NagOptimisationPackageType|)))
```

———-

1.88.10 NagOrdinaryDifferentialEquationsPackage

— defclass NagOrdinaryDifferentialEquationsPackageType —

```
(defclass |NagOrdinaryDifferentialEquationsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagOrdinaryDifferentialEquationsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGD02)
   (comment :initform (list
     "This package uses the NAG Library to calculate the numerical solution of"
     "ordinary differential equations. There are two main types of problem,"
     "those in which all boundary conditions are specified at one point"
     "(initial-value problems), and those in which the boundary"
     "conditions are distributed between two or more points (boundary-"
     "value problems and eigenvalue problems). Routines are available"
     "for initial-value problems, two-point boundary-value problems and"
     "Sturm-Liouville eigenvalue problems."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagOrdinaryDifferentialEquationsPackage|
  (progn
    (push '|NagOrdinaryDifferentialEquationsPackage| *Packages*)
    (make-instance '|NagOrdinaryDifferentialEquationsPackageType|)))
```

———–

1.88.11 NagPartialDifferentialEquationsPackage

— defclass NagPartialDifferentialEquationsPackageType —

```
(defclass |NagPartialDifferentialEquationsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagPartialDifferentialEquationsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGD03)
   (comment :initform (list
     "This package uses the NAG Library to solve partial"
     "differential equations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagPartialDifferentialEquationsPackage|
  (progn
    (push '|NagPartialDifferentialEquationsPackage| *Packages*)
    (make-instance '|NagPartialDifferentialEquationsPackageType|)))
```

———–

1.88.12 NagPolynomialRootsPackage

— defclass NagPolynomialRootsPackageType —

```
(defclass |NagPolynomialRootsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagPolynomialRootsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGC02)
   (comment :initform (list
     "This package uses the NAG Library to compute the zeros of a"
     "polynomial with real or complex coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagPolynomialRootsPackage|
  (progn
    (push '|NagPolynomialRootsPackage| *Packages*)
    (make-instance '|NagPolynomialRootsPackageType|)))
```

———-

1.88.13 NagRootFindingPackage

— defclass NagRootFindingPackageType —

```
(defclass |NagRootFindingPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagRootFindingPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGC05)
   (comment :initform (list
     "This package uses the NAG Library to calculate real zeros of"
     "continuous real functions of one or more variables. (Complex"
     "equations must be expressed in terms of the equivalent larger"
     "system of real equations.)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagRootFindingPackage|
  (progn
    (push '|NagRootFindingPackage| *Packages*)
    (make-instance '|NagRootFindingPackageType|)))
```

———-

1.88.14 NagSeriesSummationPackage

— defclass NagSeriesSummationPackageType —

```
(defclass |NagSeriesSummationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagSeriesSummationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGC06)
   (comment :initform (list
     "This package uses the NAG Library to calculate the discrete Fourier"
     "transform of a sequence of real or complex data values, and"
     "applies it to calculate convolutions and correlations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagSeriesSummationPackage|
  (progn
    (push '|NagSeriesSummationPackage| *Packages*)
    (make-instance '|NagSeriesSummationPackageType|)))
```

————-

1.88.15 NagSpecialFunctionsPackage

— defclass NagSpecialFunctionsPackageType —

```
(defclass |NagSpecialFunctionsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NagSpecialFunctionsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NAGS)
   (comment :initform (list
     "This package uses the NAG Library to compute some commonly"
     "occurring physical and mathematical functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagSpecialFunctionsPackage|
  (progn
    (push '|NagSpecialFunctionsPackage| *Packages*)
    (make-instance '|NagSpecialFunctionsPackageType|)))
```

————-

1.88.16 NewSparseUnivariatePolynomialFunctions2

— defclass NewSparseUnivariatePolynomialFunctions2Type —

```
(defclass |NewSparseUnivariatePolynomialFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NewSparseUnivariatePolynomialFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'NSUP2)
   (comment :initform (list
     "This package lifts a mapping from coefficient rings R to S to"
     "a mapping from sparse univariate polynomial over R to"
     "a sparse univariate polynomial over S."
     "Note that the mapping is assumed"
     "to send zero to zero, since it will only be applied to the non-zero"
     "coefficients of the polynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewSparseUnivariatePolynomialFunctions2|
  (progn
    (push '|NewSparseUnivariatePolynomialFunctions2| *Packages*)
    (make-instance '|NewSparseUnivariatePolynomialFunctions2Type|)))
```

————-

1.88.17 NewtonInterpolation

— defclass NewtonInterpolationType —

```
(defclass |NewtonInterpolationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NewtonInterpolation")
   (marker :initform 'package)
   (abbreviation :initform 'NEWTON)
   (comment :initform (list
     "This package exports Newton interpolation for the special case where the"
     "result is known to be in the original integral domain"
     "The packages defined in this file provide fast fraction free rational"
     "interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewtonInterpolation|
  (progn
    (push '|NewtonInterpolation| *Packages*)
    (make-instance '|NewtonInterpolationType|)))
```

———-

1.88.18 NewtonPolygon

— defclass NewtonPolygonType —

```
(defclass |NewtonPolygonType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NewtonPolygon")
   (marker :initform 'package)
   (abbreviation :initform 'NPOLYGON)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewtonPolygon|
  (progn
    (push '|NewtonPolygon| *Packages*)
    (make-instance '|NewtonPolygonType|)))
```

———-

1.88.19 NonCommutativeOperatorDivision

— defclass NonCommutativeOperatorDivisionType —

```
(defclass |NonCommutativeOperatorDivisionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NonCommutativeOperatorDivision")
   (marker :initform 'package)
   (abbreviation :initform 'NCODIV)
   (comment :initform (list
     "This package provides a division and related operations for"
     "MonogenicLinearOperators over a Field."
     "Since the multiplication is in general non-commutative,"
     "these operations all have left- and right-hand versions."
     "This package provides the operations based on left-division."
     "  [q,r] = leftDivide(a,b) means a=b*q+r"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonCommutativeOperatorDivision|
  (progn
    (push '|NonCommutativeOperatorDivision| *Packages*)
    (make-instance '|NonCommutativeOperatorDivisionType|)))
```

———-

1.88.20 NoneFunctions1

— defclass NoneFunctions1Type —

```
(defclass |NoneFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NoneFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'NONE1)
   (comment :initform (list
     "NoneFunctions1 implements functions on None."
     "It particular it includes a particulary dangerous coercion from"
     "any other type to None."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NoneFunctions1|
  (progn
    (push '|NoneFunctions1| *Packages*)
    (make-instance '|NoneFunctions1Type|)))
```

———-

1.88.21 NonLinearFirstOrderODESolver

— defclass NonLinearFirstOrderODESolverType —

```
(defclass |NonLinearFirstOrderODESolverType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NonLinearFirstOrderODESolver")
   (marker :initform 'package)
   (abbreviation :initform 'NODE1)
   (comment :initform (list
     "NonLinearFirstOrderODESolver provides a function"
     "for finding closed form first integrals of nonlinear ordinary"
     "differential equations of order 1."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonLinearFirstOrderODESolver|
  (progn
    (push '|NonLinearFirstOrderODESolver| *Packages*)
    (make-instance '|NonLinearFirstOrderODESolverType|)))
```

———-

1.88.22 NonLinearSolvePackage

— defclass NonLinearSolvePackageType —

```
(defclass |NonLinearSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NonLinearSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'NLINSOL)
   (comment :initform (list
     "NonLinearSolvePackage is an interface to SystemSolvePackage"
     "that attempts to retract the coefficients of the equations before"
     "solving. The solutions are given in the algebraic closure of R whenever"
     "possible."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonLinearSolvePackage|
  (progn
    (push '|NonLinearSolvePackage| *Packages*)
    (make-instance '|NonLinearSolvePackageType|)))
```

———-

1.88.23 NormalizationPackage

— defclass NormalizationPackageType —

```
(defclass |NormalizationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NormalizationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NORMPK)
   (comment :initform (list
     "A package for computing normalized assocites of univariate polynomials"
     "with coefficients in a tower of simple extensions of a field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormalizationPackage|
  (progn
    (push '|NormalizationPackage| *Packages*)
    (make-instance '|NormalizationPackageType|)))
```

———-

1.88.24 NormInMonogenicAlgebra

— defclass NormInMonogenicAlgebraType —

```
(defclass |NormInMonogenicAlgebraType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NormInMonogenicAlgebra")
   (marker :initform 'package)
   (abbreviation :initform 'NORMMA)
   (comment :initform (list
     "This package implements the norm of a polynomial with coefficients"
     "in a monogenic algebra (using resultants)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormInMonogenicAlgebra|
  (progn
    (push '|NormInMonogenicAlgebra| *Packages*)
    (make-instance '|NormInMonogenicAlgebraType|)))
```

———-

1.88.25 NormRetractPackage

— defclass NormRetractPackageType —

```
(defclass |NormRetractPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NormRetractPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NORMRETR)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormRetractPackage|
  (progn
    (push '|NormRetractPackage| *Packages*)
    (make-instance '|NormRetractPackageType|)))
```

———-

1.88.26 NPCoef

— defclass NPCoefType —

```
(defclass |NPCoefType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NPCoef")
   (marker :initform 'package)
   (abbreviation :initform 'NPCOEF)
   (comment :initform (list
     "Package for the determination of the coefficients in the lifting"
     "process. Used by MultivariateLifting."
     "This package will work for every euclidean domain R which has property"
     "F, there exists a factor operation in R[x]."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NPCoef|
  (progn
    (push '|NPCoef| *Packages*)
    (make-instance '|NPCoefType|)))
```

———-

1.88.27 NumberFieldIntegralBasis

— defclass NumberFieldIntegralBasisType —

```
(defclass |NumberFieldIntegralBasisType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumberFieldIntegralBasis")
   (marker :initform 'package)
   (abbreviation :initform 'NFINTBAS)
   (comment :initform (list
     "In this package F is a framed algebra over the integers (typically"
     "F = Z[a] for some algebraic integer a).  The package provides"
     "functions to compute the integral closure of Z in the quotient"
     "quotient field of F."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumberFieldIntegralBasis|
  (progn
    (push '|NumberFieldIntegralBasis| *Packages*)
    (make-instance '|NumberFieldIntegralBasisType|)))
```

———-

1.88.28 NumberFormats

— defclass NumberFormatsType —

```
(defclass |NumberFormatsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumberFormats")
   (marker :initform 'package)
   (abbreviation :initform 'NUMFMT)
   (comment :initform (list
     "NumberFormats provides function to format and read arabic and"
     "roman numbers, to convert numbers to strings and to read"
     "floating-point numbers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumberFormats|
  (progn
    (push '|NumberFormats| *Packages*)
    (make-instance '|NumberFormatsType|)))
```

———-

1.88.29 NumberTheoreticPolynomialFunctions

— defclass NumberTheoreticPolynomialFunctionsType —

```
(defclass |NumberTheoreticPolynomialFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumberTheoreticPolynomialFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'NTPOLFN)
   (comment :initform (list
     "This package provides polynomials as functions on a ring."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumberTheoreticPolynomialFunctions|
  (progn
    (push '|NumberTheoreticPolynomialFunctions| *Packages*)
    (make-instance '|NumberTheoreticPolynomialFunctionsType|)))
```

———-

1.88.30 Numeric

— defclass NumericType —

```
(defclass |NumericType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Numeric")
   (marker :initform 'package)
   (abbreviation :initform 'NUMERIC)
   (comment :initform (list
     "Numeric provides real and complex numerical evaluation"
     "functions for various symbolic types."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Numeric|
  (progn
    (push '|Numeric| *Packages*)
    (make-instance '|NumericType|)))
```

———-

1.88.31 NumericalOrdinaryDifferentialEquations

— defclass NumericalOrdinaryDifferentialEquationsType —

```
(defclass |NumericalOrdinaryDifferentialEquationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumericalOrdinaryDifferentialEquations")
   (marker :initform 'package)
   (abbreviation :initform 'NUMODE)
   (comment :initform (list
     "This package is a suite of functions for the numerical integration of an"
     "ordinary differential equation of n variables:"
     "     dy/dx = f(y,x)\tab{5}y is an n-vector"
     "All the routines are based on a 4-th order Runge-Kutta kernel."
     "These routines generally have as arguments:"
     "n, the number of dependent variables"
     "x1, the initial point"
     "h, the step size"
     "y, a vector of initial conditions of length n"
     "which upon exit contains the solution at x1 + h"
     " "
     "derivs, a function which computes the right hand side of the"
     "ordinary differential equation: derivs(dydx,y,x) computes"
     "dydx, a vector which contains the derivative information."
     " "
     "In order of increasing complexity"
     "     rk4(y,n,x1,h,derivs) advances the solution vector to"
     "     {x1 + h} and return the values in y."
     " "
     "     {rk4(y,n,x1,h,derivs,t1,t2,t3,t4)} is the same as"
     "     {rk4(y,n,x1,h,derivs)} except that you must provide 4 scratch"
     "     arrays t1-t4 of size n."
     " "
     "     Starting with y at x1, rk4f(y,n,x1,x2,ns,derivs)"
     "     uses ns fixed steps of a 4-th order Runge-Kutta"
     "     integrator to advance the solution vector to x2 and return"
     "     the values in y.  Argument x2, is the final point, and"
```

```
    "     ns, the number of steps to take."
    " "
    "rk4qc(y,n,x1,step,eps,yscal,derivs) takes a 5-th order"
    "Runge-Kutta step with monitoring of local truncation to ensure"
    "accuracy and adjust stepsize."
    "The function takes two half steps and one full step and scales"
    "the difference in solutions at the final point. If the error is"
    "within eps, the step is taken and the result is returned."
    "If the error is not within eps, the stepsize if decreased"
    "and the procedure is tried again until the desired accuracy is"
    "reached. Upon input, an trial step size must be given and upon"
    "return, an estimate of the next step size to use is returned as"
    "well as the step size which produced the desired accuracy."
    "The scaled error is computed as"
    "     error = MAX(ABS((y2steps(i) - y1step(i))/yscal(i)))"
    "and this is compared against eps. If this is greater"
    "than eps, the step size is reduced accordingly to"
    "     hnew = 0.9 * hdid * (error/eps)**(-1/4)"
    "If the error criterion is satisfied, then we check if the"
    "step size was too fine and return a more efficient one. If"
    "error > \spad{eps} * (6.0E-04) then the next step size should be"
    "     hnext = 0.9 * hdid * (error/\spad{eps})**(-1/5)"
    "Otherwise hnext = 4.0 * hdid is returned."
    "A more detailed discussion of this and related topics can be"
    "found in the book 'Numerical Recipies' by W.Press, B.P. Flannery,"
    "S.A. Teukolsky, W.T. Vetterling published by Cambridge University Press."
    " "
    "Argument step is a record of 3 floating point"
    "numbers (try , did , next),"
    "eps is the required accuracy,"
    "yscal is the scaling vector for the difference in solutions."
    "On input, step.try should be the guess at a step"
    "size to achieve the accuracy."
    "On output, step.did contains the step size which achieved the"
    "accuracy and step.next is the next step size to use."
    " "
    "rk4qc(y,n,x1,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,t7) is the"
    "same as rk4qc(y,n,x1,step,eps,yscal,derivs) except that the user"
    "must provide the 7 scratch arrays t1-t7 of size n."
    " "
    "rk4a(y,n,x1,x2,eps,h,ns,derivs)"
    "is a driver program which uses rk4qc to integrate n ordinary"
    "differential equations starting at x1 to x2, keeping the local"
    "truncation error to within eps by changing the local step size."
    "The scaling vector is defined as"
    "     yscal(i) = abs(y(i)) + abs(h*dydx(i)) + tiny"
    "where y(i) is the solution at location x, dydx is the"
    "ordinary differential equation's right hand side, h is the current"
    "step size and tiny is 10 times the"
    "smallest positive number representable."
    " "
    "The user must supply an estimate for a trial step size and"
    "the maximum number of calls to rk4qc to use."
    "Argument x2 is the final point,"
    "eps is local truncation,"
    "ns is the maximum number of call to rk4qc to use."))
  (argslist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
```

```
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalOrdinaryDifferentialEquations|
  (progn
    (push '|NumericalOrdinaryDifferentialEquations| *Packages*)
    (make-instance '|NumericalOrdinaryDifferentialEquationsType|)))
```

———-

1.88.32 NumericalQuadrature

— defclass NumericalQuadratureType —

```
(defclass |NumericalQuadratureType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumericalQuadrature")
   (marker :initform 'package)
   (abbreviation :initform 'NUMQUAD)
   (comment :initform (list
     "This suite of routines performs numerical quadrature using"
     "algorithms derived from the basic trapezoidal rule. Because"
     "the error term of this rule contains only even powers of the"
     "step size (for open and closed versions), fast convergence"
     "can be obtained if the integrand is sufficiently smooth."
     " "
     "Each routine returns a Record of type TrapAns, which contains"
     "value Float: estimate of the integral"
     "error Float: estimate of the error in the computation"
     "totalpts Integer: total number of function evaluations"
     "success Boolean: if the integral was computed within the user"
     "specified error criterion"
     "To produce this estimate, each routine generates an internal"
     "sequence of sub-estimates, denoted by S(i), depending on the"
     "routine, to which the various convergence criteria are applied."
     "The user must supply a relative accuracy, eps_r, and an absolute"
     "accuracy, \spad{eps_a}. Convergence is obtained when either"
     "      ABS(S(i) - S(i-1)) < eps_r * ABS(S(i-1))"
     "      or ABS(S(i) - S(i-1)) < eps_a"
     "are true statements."
     " "
     "The routines come in three families and three flavors:"
     "closed: romberg, simpson, trapezoidal"
     "open: rombergo, simpsono, trapezoidalo"
     "adaptive closed: aromberg, asimpson, atrapezoidal"
     " "
     "The S(i) for the trapezoidal family is the value of the"
     "integral using an equally spaced absicca trapezoidal rule for"
     "that level of refinement."
     " "
     "The S(i) for the simpson family is the value of the integral"
     "using an equally spaced absicca simpson rule for that level of"
     "refinement."
     " "
     "The S(i) for the romberg family is the estimate of the integral"
     "using an equally spaced absicca romberg method. For"
     "the i-th level, this is an appropriate combination of all the"
     "previous trapezodial estimates so that the error term starts"
     "with the 2*(i+1) power only."
     " "
     "The three families come in a closed version, where the formulas"
     "include the endpoints, an open version where the formulas do not"
     "include the endpoints and an adaptive version, where the user"
     "is required to input the number of subintervals over which the"
     "appropriate closed family integrator will apply with the usual"
     "convergence parmeters for each subinterval. This is useful"
     "where a large number of points are needed only in a small fraction"
     "of the entire domain."
     " "
     "Each routine takes as arguments:"
     "f integrand"
     "a starting point"
     "b ending point"
     "eps_r relative error"
     "eps_a absolute error"
     "nmin refinement level when to start checking for convergence (> 1)"
     "nmax maximum level of refinement"
     " "
     "The adaptive routines take as an additional parameter,"
     "nint, the number of independent intervals to apply a closed"
     "family integrator of the same name."
     " "
     "Notes:"
     "Closed family level i uses 1 + 2**i points."
     "Open family level i uses 1 + 3**i points."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalQuadrature|
  (progn
    (push '|NumericalQuadrature| *Packages*)
    (make-instance '|NumericalQuadratureType|)))
```

————-

1.88.33 NumericComplexEigenPackage

— defclass NumericComplexEigenPackageType —

```
(defclass |NumericComplexEigenPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumericComplexEigenPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NCEP)
   (comment :initform (list
     "This package computes explicitly eigenvalues and eigenvectors of"
     "matrices with entries over the complex rational numbers."
     "The results are expressed either as complex floating numbers or as"
     "complex rational numbers depending on the type of the precision parameter."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericComplexEigenPackage|
  (progn
    (push '|NumericComplexEigenPackage| *Packages*)
    (make-instance '|NumericComplexEigenPackageType|)))
```

———-

1.88.34 NumericContinuedFraction

— defclass NumericContinuedFractionType —

```
(defclass |NumericContinuedFractionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumericContinuedFraction")
   (marker :initform 'package)
   (abbreviation :initform 'NCNTFRAC)
   (comment :initform (list
     "NumericContinuedFraction provides functions"
     "for converting floating point numbers to continued fractions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericContinuedFraction|
  (progn
    (push '|NumericContinuedFraction| *Packages*)
    (make-instance '|NumericContinuedFractionType|)))
```

———-

1.88.35 NumericRealEigenPackage

— defclass NumericRealEigenPackageType —

```
(defclass |NumericRealEigenPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumericRealEigenPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NREP)
   (comment :initform (list
     "This package computes explicitly eigenvalues and eigenvectors of"
     "matrices with entries over the Rational Numbers."
     "The results are expressed as floating numbers or as rational numbers"
     "depending on the type of the parameter Par."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericRealEigenPackage|
  (progn
    (push '|NumericRealEigenPackage| *Packages*)
    (make-instance '|NumericRealEigenPackageType|)))
```

———-

1.88.36 NumericTubePlot

— defclass NumericTubePlotType —

```
(defclass |NumericTubePlotType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NumericTubePlot")
   (marker :initform 'package)
   (abbreviation :initform 'NUMTUBE)
   (comment :initform (list
     "Package for constructing tubes around 3-dimensional parametric curves."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericTubePlot|
  (progn
    (push '|NumericTubePlot| *Packages*)
    (make-instance '|NumericTubePlotType|)))
```

———-

1.89 O

1.89.1 OctonionCategoryFunctions2

— defclass OctonionCategoryFunctions2Type —

```
(defclass |OctonionCategoryFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OctonionCategoryFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'OCTCT2)
   (comment :initform (list
     "OctonionCategoryFunctions2 implements functions between"
     "two octonion domains defined over different rings."
```

```
    "The function map is used to coerce between octonion types."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OctonionCategoryFunctions2|
  (progn
    (push '|OctonionCategoryFunctions2| *Packages*)
    (make-instance '|OctonionCategoryFunctions2Type|)))
```

———-

1.89.2 ODEIntegration

— defclass ODEIntegrationType —

```
(defclass |ODEIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ODEIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'ODEINT)
   (comment :initform (list
     "ODEIntegration provides an interface to the integrator."
     "This package is intended for use"
     "by the differential equations solver but not at top-level."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ODEIntegration|
  (progn
    (push '|ODEIntegration| *Packages*)
    (make-instance '|ODEIntegrationType|)))
```

———-

1.89.3 ODETools

— defclass ODEToolsType —

```
(defclass |ODEToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ODETools")
   (marker :initform 'package)
   (abbreviation :initform 'ODETOOLS)
   (comment :initform (list
     "ODETools provides tools for the linear ODE solver."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ODETools|
  (progn
    (push '|ODETools| *Packages*)
    (make-instance '|ODEToolsType|)))
```

———-

1.89.4 OneDimensionalArrayFunctions2

— defclass OneDimensionalArrayFunctions2Type —

```
(defclass |OneDimensionalArrayFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OneDimensionalArrayFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ARRAY12)
   (comment :initform (list
     "This package provides tools for operating on one-dimensional arrays"
     "with unary and binary functions involving different underlying types"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OneDimensionalArrayFunctions2|
  (progn
    (push '|OneDimensionalArrayFunctions2| *Packages*)
    (make-instance '|OneDimensionalArrayFunctions2Type|)))
```

———-

1.89.5 OnePointCompletionFunctions2

— defclass OnePointCompletionFunctions2Type —

```
(defclass |OnePointCompletionFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OnePointCompletionFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ONECOMP2)
   (comment :initform (list
     "Lifting of maps to one-point completions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OnePointCompletionFunctions2|
  (progn
    (push '|OnePointCompletionFunctions2| *Packages*)
    (make-instance '|OnePointCompletionFunctions2Type|)))
```

———-

1.89.6 OpenMathPackage

— defclass OpenMathPackageType —

```
(defclass |OpenMathPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OpenMathPackage")
   (marker :initform 'package)
   (abbreviation :initform 'OMPKG)
   (comment :initform (list
     "OpenMathPackage provides some simple utilities"
     "to make reading OpenMath objects easier."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathPackage|
  (progn
    (push '|OpenMathPackage| *Packages*)
    (make-instance '|OpenMathPackageType|)))
```

———-

1.89.7 OpenMathServerPackage

— defclass OpenMathServerPackageType —

```
(defclass |OpenMathServerPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OpenMathServerPackage")
   (marker :initform 'package)
   (abbreviation :initform 'OMSERVER)
   (comment :initform (list
     "OpenMathServerPackage provides the necessary"
     "operations to run AXIOM as an OpenMath server, reading/writing objects"
     "to/from a port.  Please note the facilities available here are very basic."
     "The idea is that a user calls, for example, Omserve(4000,60) and then"
     "another process sends OpenMath objects to port 4000 and reads the result."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathServerPackage|
  (progn
    (push '|OpenMathServerPackage| *Packages*)
    (make-instance '|OpenMathServerPackageType|)))
```

———-

1.89.8 OperationsQuery

— defclass OperationsQueryType —

```
(defclass |OperationsQueryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OperationsQuery")
   (marker :initform 'package)
   (abbreviation :initform 'OPQUERY)
   (comment :initform (list
     "This package exports tools to create AXIOM Library information databases."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OperationsQuery|
  (progn
    (push '|OperationsQuery| *Packages*)
    (make-instance '|OperationsQueryType|)))
```

———-

1.89.9 OrderedCompletionFunctions2

— defclass OrderedCompletionFunctions2Type —

```
(defclass |OrderedCompletionFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OrderedCompletionFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ORDCOMP2)
   (comment :initform (list
     "Lifting of maps to ordered completions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedCompletionFunctions2|
  (progn
    (push '|OrderedCompletionFunctions2| *Packages*)
    (make-instance '|OrderedCompletionFunctions2Type|)))
```

———-

1.89.10 OrderingFunctions

— defclass OrderingFunctionsType —

```
(defclass |OrderingFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OrderingFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'ORDFUNS)
   (comment :initform (list
     "This package provides ordering functions on vectors which"
     "are suitable parameters for OrderedDirectProduct."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderingFunctions|
  (progn
    (push '|OrderingFunctions| *Packages*)
    (make-instance '|OrderingFunctionsType|)))
```

———-

1.89.11 OrthogonalPolynomialFunctions

— defclass OrthogonalPolynomialFunctionsType —

```
(defclass |OrthogonalPolynomialFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OrthogonalPolynomialFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'ORTHPOL)
   (comment :initform (list
     "This package provides orthogonal polynomials as functions on a ring."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrthogonalPolynomialFunctions|
  (progn
    (push '|OrthogonalPolynomialFunctions| *Packages*)
    (make-instance '|OrthogonalPolynomialFunctionsType|)))
```

———-

1.89.12 OutputPackage

— defclass OutputPackageType —

```
(defclass |OutputPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OutputPackage")
   (marker :initform 'package)
   (abbreviation :initform 'OUT)
   (comment :initform (list
     "OutPackage allows pretty-printing from programs."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OutputPackage|
  (progn
    (push '|OutputPackage| *Packages*)
    (make-instance '|OutputPackageType|)))
```

———-

1.90 P

1.90.1 PackageForAlgebraicFunctionField

— defclass PackageForAlgebraicFunctionFieldType —

```
(defclass |PackageForAlgebraicFunctionFieldType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PackageForAlgebraicFunctionField")
   (marker :initform 'package)
   (abbreviation :initform 'PAFF)
   (comment :initform (list
     "A package that implements the Brill-Noether algorithm."
     "Part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PackageForAlgebraicFunctionField|
  (progn
    (push '|PackageForAlgebraicFunctionField| *Packages*)
    (make-instance '|PackageForAlgebraicFunctionFieldType|)))
```

———-

1.90.2 PackageForAlgebraicFunctionFieldOverFiniteField

— defclass PackageForAlgebraicFunctionFieldOverFiniteFieldType —

```
(defclass |PackageForAlgebraicFunctionFieldOverFiniteFieldType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PackageForAlgebraicFunctionFieldOverFiniteField")
   (marker :initform 'package)
   (abbreviation :initform 'PAFFFF)
   (comment :initform (list
     "A package that implements the Brill-Noether algorithm."
     "Part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PackageForAlgebraicFunctionFieldOverFiniteField|
  (progn
    (push '|PackageForAlgebraicFunctionFieldOverFiniteField| *Packages*)
    (make-instance '|PackageForAlgebraicFunctionFieldOverFiniteFieldType|)))
```

———-

1.90.3 PackageForPoly

— defclass PackageForPolyType —

```
(defclass |PackageForPolyType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PackageForPoly")
   (marker :initform 'package)
   (abbreviation :initform 'PFORP)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PackageForPoly|
  (progn
    (push '|PackageForPoly| *Packages*)
    (make-instance '|PackageForPolyType|)))
```

———-

1.90.4 PadeApproximantPackage

— defclass PadeApproximantPackageType —

```
(defclass |PadeApproximantPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PadeApproximantPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PADEPAC)
   (comment :initform (list
     "This package computes reliable Pad&ea. approximants using"
     "a generalized Viskovatov continued fraction algorithm."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PadeApproximantPackage|
  (progn
    (push '|PadeApproximantPackage| *Packages*)
    (make-instance '|PadeApproximantPackageType|)))
```

———-

1.90.5 PadeApproximants

— defclass PadeApproximantsType —

```
(defclass |PadeApproximantsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PadeApproximants")
   (marker :initform 'package)
   (abbreviation :initform 'PADE)
   (comment :initform (list
     "This package computes reliable Pad&ea. approximants using"
     "a generalized Viskovatov continued fraction algorithm."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PadeApproximants|
  (progn
    (push '|PadeApproximants| *Packages*)
    (make-instance '|PadeApproximantsType|)))
```

———-

1.90.6 PAdicWildFunctionFieldIntegralBasis

— defclass PAdicWildFunctionFieldIntegralBasisType —

```
(defclass |PAdicWildFunctionFieldIntegralBasisType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PAdicWildFunctionFieldIntegralBasis")
   (marker :initform 'package)
   (abbreviation :initform 'PWFFINTB)
   (comment :initform (list
     "In this package K is a finite field, R is a ring of univariate"
     "polynomials over K, and F is a monogenic algebra over R."
     "We require that F is monogenic, that F = K[x,y]/(f(x,y)),"
     "because the integral basis algorithm used will factor the polynomial"
     "f(x,y).  The package provides a function to compute the integral"
     "closure of R in the quotient field of F as well as a function to compute"
     "a 'local integral basis' at a specific prime."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicWildFunctionFieldIntegralBasis|
  (progn
    (push '|PAdicWildFunctionFieldIntegralBasis| *Packages*)
    (make-instance '|PAdicWildFunctionFieldIntegralBasisType|)))
```

———-

1.90.7 ParadoxicalCombinatorsForStreams

— defclass ParadoxicalCombinatorsForStreamsType —

```
(defclass |ParadoxicalCombinatorsForStreamsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParadoxicalCombinatorsForStreams")
   (marker :initform 'package)
   (abbreviation :initform 'YSTREAM)
   (comment :initform (list
     "Computation of fixed points of mappings on streams"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParadoxicalCombinatorsForStreams|
  (progn
    (push '|ParadoxicalCombinatorsForStreams| *Packages*)
    (make-instance '|ParadoxicalCombinatorsForStreamsType|)))
```

———-

1.90.8 ParametricLinearEquations

— defclass ParametricLinearEquationsType —

```
(defclass |ParametricLinearEquationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricLinearEquations")
   (marker :initform 'package)
   (abbreviation :initform 'PLEQN)
   (comment :initform (list
     "This package completely solves a parametric linear system of equations"
     "by decomposing the set of all parametric values for which the linear"
     "system is consistent into a union of quasi-algebraic  sets (which need"
     "not be irredundant, but most of the time is). Each quasi-algebraic"
     "set is described by a list of polynomials that vanish on the set, and"
     "a list of polynomials that vanish at no point of the set."
     "For each quasi-algebraic set, the solution of the linear system"
     "is given, as a particular solution and  a basis of the homogeneous"
     "system."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricLinearEquations|
  (progn
    (push '|ParametricLinearEquations| *Packages*)
    (make-instance '|ParametricLinearEquationsType|)))
```

————-

1.90.9 ParametricPlaneCurveFunctions2

— defclass ParametricPlaneCurveFunctions2Type —

```
(defclass |ParametricPlaneCurveFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricPlaneCurveFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PARPC2)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricPlaneCurveFunctions2|
  (progn
    (push '|ParametricPlaneCurveFunctions2| *Packages*)
    (make-instance '|ParametricPlaneCurveFunctions2Type|)))
```

————-

1.90.10 ParametricSpaceCurveFunctions2

— defclass ParametricSpaceCurveFunctions2Type —

```
(defclass |ParametricSpaceCurveFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricSpaceCurveFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PARSC2)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSpaceCurveFunctions2|
  (progn
    (push '|ParametricSpaceCurveFunctions2| *Packages*)
    (make-instance '|ParametricSpaceCurveFunctions2Type|)))
```

————-

1.90.11 ParametricSurfaceFunctions2

— defclass ParametricSurfaceFunctions2Type —

```
(defclass |ParametricSurfaceFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricSurfaceFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PARSU2)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSurfaceFunctions2|
  (progn
    (push '|ParametricSurfaceFunctions2| *Packages*)
    (make-instance '|ParametricSurfaceFunctions2Type|)))
```

————-

1.90.12 ParametrizationPackage

— defclass ParametrizationPackageType —

```
(defclass |ParametrizationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametrizationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PARAMP)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametrizationPackage|
  (progn
    (push '|ParametrizationPackage| *Packages*)
    (make-instance '|ParametrizationPackageType|)))
```

———-

1.90.13 PartialFractionPackage

— defclass PartialFractionPackageType —

```
(defclass |PartialFractionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PartialFractionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PFRPAC)
   (comment :initform (list
     "The package PartialFractionPackage gives an easier"
     "to use interfact the domain PartialFraction."
     "The user gives a fraction of polynomials, and a variable and"
     "the package converts it to the proper datatype for the"
     "PartialFraction domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialFractionPackage|
  (progn
    (push '|PartialFractionPackage| *Packages*)
    (make-instance '|PartialFractionPackageType|)))
```

———-

1.90.14 PartitionsAndPermutations

— defclass PartitionsAndPermutationsType —

```
(defclass |PartitionsAndPermutationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PartitionsAndPermutations")
   (marker :initform 'package)
   (abbreviation :initform 'PARTPERM)
   (comment :initform (list
     "PartitionsAndPermutations contains functions for generating streams of"
     "integer partitions, and streams of sequences of integers"
     "composed from a multi-set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartitionsAndPermutations|
  (progn
    (push '|PartitionsAndPermutations| *Packages*)
    (make-instance '|PartitionsAndPermutationsType|)))
```

———-

1.90.15 PatternFunctions1

— defclass PatternFunctions1Type —

```
(defclass |PatternFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'PATTERN1)
   (comment :initform (list
     "Utilities for handling patterns"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternFunctions1|
  (progn
    (push '|PatternFunctions1| *Packages*)
    (make-instance '|PatternFunctions1Type|)))
```

———-

1.90.16 PatternFunctions2

— defclass PatternFunctions2Type —

```
(defclass |PatternFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PATTERN2)
   (comment :initform (list
     "Lifts maps to patterns"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternFunctions2|
  (progn
    (push '|PatternFunctions2| *Packages*)
    (make-instance '|PatternFunctions2Type|)))
```

———-

1.90.17 PatternMatch

— defclass PatternMatchType —

```
(defclass |PatternMatchType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatch")
   (marker :initform 'package)
   (abbreviation :initform 'PATMATCH)
   (comment :initform (list
     "This package provides the top-level pattern macthing functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatch|
  (progn
    (push '|PatternMatch| *Packages*)
    (make-instance '|PatternMatchType|)))
```

———-

1.90.18 PatternMatchAssertions

— defclass PatternMatchAssertionsType —

```
(defclass |PatternMatchAssertionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchAssertions")
   (marker :initform 'package)
   (abbreviation :initform 'PMASS)
   (comment :initform (list
     "Attaching assertions to symbols for pattern matching;"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchAssertions|
  (progn
    (push '|PatternMatchAssertions| *Packages*)
    (make-instance '|PatternMatchAssertionsType|)))
```

————-

1.90.19 PatternMatchFunctionSpace

— defclass PatternMatchFunctionSpaceType —

```
(defclass |PatternMatchFunctionSpaceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchFunctionSpace")
   (marker :initform 'package)
   (abbreviation :initform 'PMFS)
   (comment :initform (list
     "This package provides pattern matching functions on function spaces."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchFunctionSpace|
  (progn
    (push '|PatternMatchFunctionSpace| *Packages*)
    (make-instance '|PatternMatchFunctionSpaceType|)))
```

————-

1.90.20 PatternMatchIntegerNumberSystem

— defclass PatternMatchIntegerNumberSystemType —

```
(defclass |PatternMatchIntegerNumberSystemType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchIntegerNumberSystem")
   (marker :initform 'package)
   (abbreviation :initform 'PMINS)
   (comment :initform (list
     "This package provides pattern matching functions on integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchIntegerNumberSystem|
  (progn
    (push '|PatternMatchIntegerNumberSystem| *Packages*)
    (make-instance '|PatternMatchIntegerNumberSystemType|)))
```

———-

1.90.21 PatternMatchIntegration

— defclass PatternMatchIntegrationType —

```
(defclass |PatternMatchIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTPM)
   (comment :initform (list
     "PatternMatchIntegration provides functions that use"
     "the pattern matcher to find some indefinite and definite integrals"
     "involving special functions and found in the litterature."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchIntegration|
  (progn
    (push '|PatternMatchIntegration| *Packages*)
    (make-instance '|PatternMatchIntegrationType|)))
```

———-

1.90.22 PatternMatchKernel

— defclass PatternMatchKernelType —

```
(defclass |PatternMatchKernelType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchKernel")
   (marker :initform 'package)
   (abbreviation :initform 'PMKERNEL)
   (comment :initform (list
     "This package provides pattern matching functions on kernels."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PatternMatchKernel|
  (progn
    (push '|PatternMatchKernel| *Packages*)
    (make-instance '|PatternMatchKernelType|)))
```

———-

1.90.23 PatternMatchListAggregate

— defclass PatternMatchListAggregateType —

```
(defclass |PatternMatchListAggregateType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchListAggregate")
   (marker :initform 'package)
   (abbreviation :initform 'PMLSAGG)
   (comment :initform (list
     "This package provides pattern matching functions on lists."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchListAggregate|
  (progn
    (push '|PatternMatchListAggregate| *Packages*)
    (make-instance '|PatternMatchListAggregateType|)))
```

———-

1.90.24 PatternMatchPolynomialCategory

— defclass PatternMatchPolynomialCategoryType —

```
(defclass |PatternMatchPolynomialCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchPolynomialCategory")
   (marker :initform 'package)
   (abbreviation :initform 'PMPLCAT)
   (comment :initform (list
     "This package provides pattern matching functions on polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchPolynomialCategory|
  (progn
    (push '|PatternMatchPolynomialCategory| *Packages*)
    (make-instance '|PatternMatchPolynomialCategoryType|)))
```

———-

1.90.25 PatternMatchPushDown

— defclass PatternMatchPushDownType —

```
(defclass |PatternMatchPushDownType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchPushDown")
   (marker :initform 'package)
   (abbreviation :initform 'PMDOWN)
   (comment :initform (list
     "This packages provides tools for matching recursively in type towers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchPushDown|
  (progn
    (push '|PatternMatchPushDown| *Packages*)
    (make-instance '|PatternMatchPushDownType|)))
```

———-

1.90.26 PatternMatchQuotientFieldCategory

— defclass PatternMatchQuotientFieldCategoryType —

```
(defclass |PatternMatchQuotientFieldCategoryType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchQuotientFieldCategory")
   (marker :initform 'package)
   (abbreviation :initform 'PMQFCAT)
   (comment :initform (list
     "This package provides pattern matching functions on quotients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchQuotientFieldCategory|
  (progn
    (push '|PatternMatchQuotientFieldCategory| *Packages*)
    (make-instance '|PatternMatchQuotientFieldCategoryType|)))
```

———-

1.90.27 PatternMatchResultFunctions2

— defclass PatternMatchResultFunctions2Type —

```
(defclass |PatternMatchResultFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchResultFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PATRES2)
   (comment :initform (list
     "Lifts maps to pattern matching results."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchResultFunctions2|
  (progn
    (push '|PatternMatchResultFunctions2| *Packages*)
    (make-instance '|PatternMatchResultFunctions2Type|)))
```

————-

1.90.28 PatternMatchSymbol

— defclass PatternMatchSymbolType —

```
(defclass |PatternMatchSymbolType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchSymbol")
   (marker :initform 'package)
   (abbreviation :initform 'PMSYM)
   (comment :initform (list
     "This package provides pattern matching functions on symbols."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchSymbol|
  (progn
    (push '|PatternMatchSymbol| *Packages*)
    (make-instance '|PatternMatchSymbolType|)))
```

————-

1.90.29 PatternMatchTools

— defclass PatternMatchToolsType —

```
(defclass |PatternMatchToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchTools")
   (marker :initform 'package)
   (abbreviation :initform 'PMTOOLS)
   (comment :initform (list
     "This package provides tools for the pattern matcher."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchTools|
  (progn
    (push '|PatternMatchTools| *Packages*)
    (make-instance '|PatternMatchToolsType|)))
```

———-

1.90.30 Permanent

— defclass PermanentType —

```
(defclass |PermanentType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Permanent")
   (marker :initform 'package)
   (abbreviation :initform 'PERMAN)
   (comment :initform (list
     "Permanent implements the functions permanent, the"
     "permanent for square matrices."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Permanent|
  (progn
    (push '|Permanent| *Packages*)
    (make-instance '|PermanentType|)))
```

———-

1.90.31 PermutationGroupExamples

— defclass PermutationGroupExamplesType —

```
(defclass |PermutationGroupExamplesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PermutationGroupExamples")
   (marker :initform 'package)
   (abbreviation :initform 'PGE)
   (comment :initform (list
     "PermutationGroupExamples provides permutation groups for"
     "some classes of groups: symmetric, alternating, dihedral, cyclic,"
     "direct products of cyclic, which are in fact the finite abelian groups"
     "of symmetric groups called Young subgroups."
     "Furthermore, Rubik's group as permutation group of 48 integers and a list"
     "of sporadic simple groups derived from the atlas of finite groups."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PermutationGroupExamples|
  (progn
    (push '|PermutationGroupExamples| *Packages*)
    (make-instance '|PermutationGroupExamplesType|)))
```

———-

1.90.32 PiCoercions

— defclass PiCoercionsType —

```
(defclass |PiCoercionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PiCoercions")
   (marker :initform 'package)
   (abbreviation :initform 'PICOERCE)
   (comment :initform (list
     "Provides a coercion from the symbolic fractions in %pi with"
     "integer coefficients to any Expression type."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PiCoercions|
  (progn
    (push '|PiCoercions| *Packages*)
    (make-instance '|PiCoercionsType|)))
```

———-

1.90.33 PlotFunctions1

— defclass PlotFunctions1Type —

```
(defclass |PlotFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PlotFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'PLOT1)
   (comment :initform (list
     "PlotFunctions1 provides facilities for plotting curves"
     "where functions SF -> SF are specified by giving an expression"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlotFunctions1|
  (progn
    (push '|PlotFunctions1| *Packages*)
    (make-instance '|PlotFunctions1Type|)))
```

———-

1.90.34 PlotTools

— defclass PlotToolsType —

```
(defclass |PlotToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PlotTools")
   (marker :initform 'package)
   (abbreviation :initform 'PLOTTOOL)
   (comment :initform (list
     "This package exports plotting tools"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlotTools|
  (progn
    (push '|PlotTools| *Packages*)
    (make-instance '|PlotToolsType|)))
```

———-

1.90.35 ProjectiveAlgebraicSetPackage

— defclass ProjectiveAlgebraicSetPackageType —

```
(defclass |ProjectiveAlgebraicSetPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ProjectiveAlgebraicSetPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PRJALGPK)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectiveAlgebraicSetPackage|
  (progn
    (push '|ProjectiveAlgebraicSetPackage| *Packages*)
    (make-instance '|ProjectiveAlgebraicSetPackageType|)))
```

——————-

1.90.36 PointFunctions2

— defclass PointFunctions2Type —

```
(defclass |PointFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PTFUNC2)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointFunctions2|
  (progn
    (push '|PointFunctions2| *Packages*)
    (make-instance '|PointFunctions2Type|)))
```

——————-

1.90.37 PointPackage

— defclass PointPackageType —

```
(defclass |PointPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PTPACK)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointPackage|
  (progn
    (push '|PointPackage| *Packages*)
    (make-instance '|PointPackageType|)))
```

———-

1.90.38 PointsOfFiniteOrder

— defclass PointsOfFiniteOrderType —

```
(defclass |PointsOfFiniteOrderType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointsOfFiniteOrder")
   (marker :initform 'package)
   (abbreviation :initform 'PFO)
   (comment :initform (list
     "This package provides function for testing whether a divisor on a"
     "curve is a torsion divisor."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointsOfFiniteOrder|
  (progn
    (push '|PointsOfFiniteOrder| *Packages*)
    (make-instance '|PointsOfFiniteOrderType|)))
```

———-

1.90.39 PointsOfFiniteOrderRational

— defclass PointsOfFiniteOrderRationalType —

```
(defclass |PointsOfFiniteOrderRationalType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointsOfFiniteOrderRational")
   (marker :initform 'package)
   (abbreviation :initform 'PFOQ)
   (comment :initform (list
     "This package provides function for testing whether a divisor on a"
     "curve is a torsion divisor."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointsOfFiniteOrderRational|
  (progn
    (push '|PointsOfFiniteOrderRational| *Packages*)
    (make-instance '|PointsOfFiniteOrderRationalType|)))
```

————-

1.90.40 PointsOfFiniteOrderTools

— defclass PointsOfFiniteOrderToolsType —

```
(defclass |PointsOfFiniteOrderToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointsOfFiniteOrderTools")
   (marker :initform 'package)
   (abbreviation :initform 'PFOTOOLS)
   (comment :initform (list
     "Utilities for PFOQ and PFO"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointsOfFiniteOrderTools|
  (progn
    (push '|PointsOfFiniteOrderTools| *Packages*)
    (make-instance '|PointsOfFiniteOrderToolsType|)))
```

————-

1.90.41 PolynomialPackageForCurve

— defclass PolynomialPackageForCurveType —

```
(defclass |PolynomialPackageForCurveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialPackageForCurve")
   (marker :initform 'package)
   (abbreviation :initform 'PLPKCRV)
   (comment :initform (list
     "The following is part of the PAFF package"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialPackageForCurve|
  (progn
    (push '|PolynomialPackageForCurve| *Packages*)
    (make-instance '|PolynomialPackageForCurveType|)))
```

———-

1.90.42 PolToPol

— defclass PolToPolType —

```
(defclass |PolToPolType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolToPol")
   (marker :initform 'package)
   (abbreviation :initform 'POLTOPOL)
   (comment :initform (list
     "Package with the conversion functions among different kind of polynomials"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolToPol|
  (progn
    (push '|PolToPol| *Packages*)
    (make-instance '|PolToPolType|)))
```

———-

1.90.43 PolyGroebner

— defclass PolyGroebnerType —

```
(defclass |PolyGroebnerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolyGroebner")
   (marker :initform 'package)
   (abbreviation :initform 'PGROEB)
   (comment :initform (list
     "Groebner functions for P F"
     "This package is an interface package to the groebner basis"
     "package which allows you to compute groebner bases for polynomials"
     "in either lexicographic ordering or total degree ordering refined"
     "by reverse lex. The input is the ordinary polynomial type which"
     "is internally converted to a type with the required ordering."
     "The resulting grobner basis is converted back to ordinary polynomials."
     "The ordering among the variables is controlled by an explicit list"
     "of variables which is passed as a second argument. The coefficient"
     "domain is allowed to be any gcd domain, but the groebner basis is"
     "computed as if the polynomials were over a field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolyGroebner|
  (progn
    (push '|PolyGroebner| *Packages*)
    (make-instance '|PolyGroebnerType|)))
```

———-

1.90.44 PolynomialAN2Expression

— defclass PolynomialAN2ExpressionType —

```
(defclass |PolynomialAN2ExpressionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialAN2Expression")
   (marker :initform 'package)
   (abbreviation :initform 'PAN2EXPR)
   (comment :initform (list
     "This package provides a coerce from polynomials over"
     "algebraic numbers to Expression AlgebraicNumber."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialAN2Expression|
  (progn
    (push '|PolynomialAN2Expression| *Packages*)
    (make-instance '|PolynomialAN2ExpressionType|)))
```

———-

1.90.45 PolynomialCategoryLifting

— defclass PolynomialCategoryLiftingType —

```
(defclass |PolynomialCategoryLiftingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialCategoryLifting")
   (marker :initform 'package)
   (abbreviation :initform 'POLYLIFT)
   (comment :initform (list
     "This package provides a very general map function, which"
     "given a set S and polynomials over R with maps from the"
     "variables into S and the coefficients into S, maps polynomials"
     "into S. S is assumed to support +, * and **."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialCategoryLifting|
  (progn
    (push '|PolynomialCategoryLifting| *Packages*)
    (make-instance '|PolynomialCategoryLiftingType|)))
```

———-

1.90.46 PolynomialCategoryQuotientFunctions

— defclass PolynomialCategoryQuotientFunctionsType —

```
(defclass |PolynomialCategoryQuotientFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialCategoryQuotientFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'POLYCATQ)
   (comment :initform (list
     "Manipulations on polynomial quotients"
     "This package transforms multivariate polynomials or fractions into"
     "univariate polynomials or fractions, and back."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialCategoryQuotientFunctions|
  (progn
    (push '|PolynomialCategoryQuotientFunctions| *Packages*)
    (make-instance '|PolynomialCategoryQuotientFunctionsType|)))
```

———-

1.90.47 PolynomialComposition

— defclass PolynomialCompositionType —

```
(defclass |PolynomialCompositionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialComposition")
   (marker :initform 'package)
   (abbreviation :initform 'PCOMP)
   (comment :initform (list
     "Polynomial composition and decomposition functions"
     "If f = g o h then g=leftFactor(f,h) and h=rightFactor(f,g)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialComposition|
  (progn
    (push '|PolynomialComposition| *Packages*)
    (make-instance '|PolynomialCompositionType|)))
```

———-

1.90.48 PolynomialDecomposition

— defclass PolynomialDecompositionType —

```
(defclass |PolynomialDecompositionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialDecomposition")
   (marker :initform 'package)
   (abbreviation :initform 'PDECOMP)
   (comment :initform (list
     "Polynomial composition and decomposition functions"
     "If f = g o h then g=leftFactor(f,h) and h=rightFactor(f,g)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialDecomposition|
  (progn
    (push '|PolynomialDecomposition| *Packages*)
    (make-instance '|PolynomialDecompositionType|)))
```

———-

1.90.49 PolynomialFactorizationByRecursion

— defclass PolynomialFactorizationByRecursionType —

```
(defclass |PolynomialFactorizationByRecursionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialFactorizationByRecursion")
   (marker :initform 'package)
   (abbreviation :initform 'PFBR)
   (comment :initform (list
     "PolynomialFactorizationByRecursion(R,E,VarSet,S)"
     "is used for factorization of sparse univariate polynomials over"
     "a domain S of multivariate polynomials over R."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFactorizationByRecursion|
  (progn
    (push '|PolynomialFactorizationByRecursion| *Packages*)
    (make-instance '|PolynomialFactorizationByRecursionType|)))
```

———-

1.90.50 PolynomialFactorizationByRecursionUnivariate

— defclass PolynomialFactorizationByRecursionUnivariateType —

```
(defclass |PolynomialFactorizationByRecursionUnivariateType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialFactorizationByRecursionUnivariate")
   (marker :initform 'package)
   (abbreviation :initform 'PFBRU)
   (comment :initform (list
     "PolynomialFactorizationByRecursionUnivariate"
     "R is a PolynomialFactorizationExplicit domain,"
     "S is univariate polynomials over R"
     "We are interested in handling SparseUnivariatePolynomials over"
     "S, is a variable we shall call z"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFactorizationByRecursionUnivariate|
  (progn
    (push '|PolynomialFactorizationByRecursionUnivariate| *Packages*)
    (make-instance '|PolynomialFactorizationByRecursionUnivariateType|)))
```

———-

1.90.51 PolynomialFunctions2

— defclass PolynomialFunctions2Type —

```
(defclass |PolynomialFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'POLY2)
   (comment :initform (list
     "This package takes a mapping between coefficient rings, and lifts"
     "it to a mapping between polynomials over those rings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFunctions2|
  (progn
    (push '|PolynomialFunctions2| *Packages*)
    (make-instance '|PolynomialFunctions2Type|)))
```

———-

1.90.52 PolynomialGcdPackage

— defclass PolynomialGcdPackageType —

```
(defclass |PolynomialGcdPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialGcdPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PGCD)
   (comment :initform (list
     "This package computes multivariate polynomial gcd's using"
     "a hensel lifting strategy. The constraint on the coefficient"
     "domain is imposed by the lifting strategy. It is assumed that"
     "the coefficient domain has the property that almost all specializations"
     "preserve the degree of the gcd."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialGcdPackage|
  (progn
    (push '|PolynomialGcdPackage| *Packages*)
    (make-instance '|PolynomialGcdPackageType|)))
```

———-

1.90.53 PolynomialInterpolation

— defclass PolynomialInterpolationType —

```
(defclass |PolynomialInterpolationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialInterpolation")
   (marker :initform 'package)
   (abbreviation :initform 'PINTERP)
   (comment :initform (list
     "This package exports interpolation algorithms"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialInterpolation|
  (progn
    (push '|PolynomialInterpolation| *Packages*)
    (make-instance '|PolynomialInterpolationType|)))
```

———-

1.90.54 PolynomialInterpolationAlgorithms

— defclass PolynomialInterpolationAlgorithmsType —

```
(defclass |PolynomialInterpolationAlgorithmsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialInterpolationAlgorithms")
   (marker :initform 'package)
   (abbreviation :initform 'PINTERPA)
   (comment :initform (list
     "This package exports interpolation algorithms"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialInterpolationAlgorithms|
  (progn
    (push '|PolynomialInterpolationAlgorithms| *Packages*)
    (make-instance '|PolynomialInterpolationAlgorithmsType|)))
```

———-

1.90.55 PolynomialNumberTheoryFunctions

— defclass PolynomialNumberTheoryFunctionsType —

```
(defclass |PolynomialNumberTheoryFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialNumberTheoryFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'PNTHEORY)
   (comment :initform (list
     "This package provides various polynomial number theoretic functions"
     "over the integers."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PolynomialNumberTheoryFunctions|
  (progn
    (push '|PolynomialNumberTheoryFunctions| *Packages*)
    (make-instance '|PolynomialNumberTheoryFunctionsType|)))
```

——————-

1.90.56 PolynomialRoots

— defclass PolynomialRootsType —

```
(defclass |PolynomialRootsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialRoots")
   (marker :initform 'package)
   (abbreviation :initform 'POLYROOT)
   (comment :initform (list
     "Computes n-th roots of quotients of multivariate polynomials"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialRoots|
  (progn
    (push '|PolynomialRoots| *Packages*)
    (make-instance '|PolynomialRootsType|)))
```

——————-

1.90.57 PolynomialSetUtilitiesPackage

— defclass PolynomialSetUtilitiesPackageType —

```
(defclass |PolynomialSetUtilitiesPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialSetUtilitiesPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PSETPK)
   (comment :initform (list
     "This package provides modest routines for polynomial system solving."
     "The aim of many of the operations of this package is to remove certain"
     "factors in some polynomials in order to avoid unnecessary computations"
     "in algorithms involving splitting techniques by partial factorization."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PolynomialSetUtilitiesPackage|
  (progn
    (push '|PolynomialSetUtilitiesPackage| *Packages*)
    (make-instance '|PolynomialSetUtilitiesPackageType|)))
```

———-

1.90.58 PolynomialSolveByFormulas

— defclass PolynomialSolveByFormulasType —

```
(defclass |PolynomialSolveByFormulasType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialSolveByFormulas")
   (marker :initform 'package)
   (abbreviation :initform 'SOLVEFOR)
   (comment :initform (list
     "This package factors the formulas out of the general solve code,"
     "allowing their recursive use over different domains."
     "Care is taken to introduce few radicals so that radical extension"
     "domains can more easily simplify the results."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialSolveByFormulas|
  (progn
    (push '|PolynomialSolveByFormulas| *Packages*)
    (make-instance '|PolynomialSolveByFormulasType|)))
```

———-

1.90.59 PolynomialSquareFree

— defclass PolynomialSquareFreeType —

```
(defclass |PolynomialSquareFreeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialSquareFree")
   (marker :initform 'package)
   (abbreviation :initform 'PSQFR)
   (comment :initform (list
     "This package computes square-free decomposition of multivariate"
     "polynomials over a coefficient ring which is an arbitrary gcd domain."
     "The requirement on the coefficient domain guarantees that the"
     "content can be"
     "removed so that factors will be primitive as well as square-free."
     "Over an infinite ring of finite characteristic,it may not be possible to"
     "guarantee that the factors are square-free."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialSquareFree|
  (progn
    (push '|PolynomialSquareFree| *Packages*)
    (make-instance '|PolynomialSquareFreeType|)))
```

————-

1.90.60 PolynomialToUnivariatePolynomial

— defclass PolynomialToUnivariatePolynomialType —

```
(defclass |PolynomialToUnivariatePolynomialType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialToUnivariatePolynomial")
   (marker :initform 'package)
   (abbreviation :initform 'POLY2UP)
   (comment :initform (list
     "This package is primarily to help the interpreter do coercions."
     "It allows you to view a polynomial as a"
     "univariate polynomial in one of its variables with"
     "coefficients which are again a polynomial in all the"
     "other variables."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialToUnivariatePolynomial|
  (progn
    (push '|PolynomialToUnivariatePolynomial| *Packages*)
    (make-instance '|PolynomialToUnivariatePolynomialType|)))
```

————-

1.90.61 PowerSeriesLimitPackage

— defclass PowerSeriesLimitPackageType —

```
(defclass |PowerSeriesLimitPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PowerSeriesLimitPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LIMITPS)
   (comment :initform (list
     "PowerSeriesLimitPackage implements limits of expressions"
     "in one or more variables as one of the variables approaches a"
```

```
    "limiting value.  Included are two-sided limits, left- and right-"
    "hand limits, and limits at plus or minus infinity."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PowerSeriesLimitPackage|
  (progn
    (push '|PowerSeriesLimitPackage| *Packages*)
    (make-instance '|PowerSeriesLimitPackageType|)))
```

————-

1.90.62 PrecomputedAssociatedEquations

— defclass PrecomputedAssociatedEquationsType —

```
(defclass |PrecomputedAssociatedEquationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PrecomputedAssociatedEquations")
   (marker :initform 'package)
   (abbreviation :initform 'PREASSOC)
   (comment :initform (list
     "PrecomputedAssociatedEquations stores some generic"
     "precomputations which speed up the computations of the"
     "associated equations needed for factoring operators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrecomputedAssociatedEquations|
  (progn
    (push '|PrecomputedAssociatedEquations| *Packages*)
    (make-instance '|PrecomputedAssociatedEquationsType|)))
```

————-

1.90.63 PrimitiveArrayFunctions2

— defclass PrimitiveArrayFunctions2Type —

```
(defclass |PrimitiveArrayFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PrimitiveArrayFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PRIMARR2)
   (comment :initform (list
     "This package provides tools for operating on primitive arrays"
```

```
    "with unary and binary functions involving different underlying types"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveArrayFunctions2|
  (progn
    (push '|PrimitiveArrayFunctions2| *Packages*)
    (make-instance '|PrimitiveArrayFunctions2Type|)))
```

———-

1.90.64 PrimitiveElement

— defclass PrimitiveElementType —

```
(defclass |PrimitiveElementType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PrimitiveElement")
   (marker :initform 'package)
   (abbreviation :initform 'PRIMELT)
   (comment :initform (list
     "PrimitiveElement provides functions to compute primitive elements"
     "in algebraic extensions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveElement|
  (progn
    (push '|PrimitiveElement| *Packages*)
    (make-instance '|PrimitiveElementType|)))
```

———-

1.90.65 PrimitiveRatDE

— defclass PrimitiveRatDEType —

```
(defclass |PrimitiveRatDEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PrimitiveRatDE")
   (marker :initform 'package)
   (abbreviation :initform 'ODEPRIM)
   (comment :initform (list
     "PrimitiveRatDE provides functions for in-field solutions of linear"
     "ordinary differential equations, in the transcendental case."
     "The derivation to use is given by the parameter L."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveRatDE|
  (progn
    (push '|PrimitiveRatDE| *Packages*)
    (make-instance '|PrimitiveRatDEType|)))
```

———-

1.90.66 PrimitiveRatRicDE

— defclass PrimitiveRatRicDEType —

```
(defclass |PrimitiveRatRicDEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PrimitiveRatRicDE")
   (marker :initform 'package)
   (abbreviation :initform 'ODEPRRIC)
   (comment :initform (list
     "In-field solution of Riccati equations, primitive case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveRatRicDE|
  (progn
    (push '|PrimitiveRatRicDE| *Packages*)
    (make-instance '|PrimitiveRatRicDEType|)))
```

———-

1.90.67 PrintPackage

— defclass PrintPackageType —

```
(defclass |PrintPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PrintPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PRINT)
   (comment :initform (list
     "PrintPackage provides a print function for output forms."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrintPackage|
  (progn
    (push '|PrintPackage| *Packages*)
    (make-instance '|PrintPackageType|)))
```

———–

1.90.68 PseudoLinearNormalForm

— defclass PseudoLinearNormalFormType —

```
(defclass |PseudoLinearNormalFormType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PseudoLinearNormalForm")
   (marker :initform 'package)
   (abbreviation :initform 'PSEUDLIN)
   (comment :initform (list
     "PseudoLinearNormalForm provides a function for computing a block-companion"
     "form for pseudo-linear operators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoLinearNormalForm|
  (progn
    (push '|PseudoLinearNormalForm| *Packages*)
    (make-instance '|PseudoLinearNormalFormType|)))
```

———–

1.90.69 PseudoRemainderSequence

— defclass PseudoRemainderSequenceType —

```
(defclass |PseudoRemainderSequenceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PseudoRemainderSequence")
   (marker :initform 'package)
   (abbreviation :initform 'PRS)
   (comment :initform (list
     "This package contains some functions: discriminant, resultant,"
     "subResultantGcd, chainSubResultants, degreeSubResultant, lastSubResultant,"
     "resultantEuclidean, subResultantGcdEuclidean, semiSubResultantGcdEuclidean1,"
     "semiSubResultantGcdEuclidean2"
     "These procedures come from improvements of the subresultants algorithm."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoRemainderSequence|
  (progn
    (push '|PseudoRemainderSequence| *Packages*)
    (make-instance '|PseudoRemainderSequenceType|)))
```

———-

1.90.70 PureAlgebraicIntegration

— defclass PureAlgebraicIntegrationType —

```
(defclass |PureAlgebraicIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PureAlgebraicIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTPAF)
   (comment :initform (list
     "Integration of pure algebraic functions."
     "This package provides functions for integration, limited integration,"
     "extended integration and the risch differential equation for"
     "pure algebraic integrands."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PureAlgebraicIntegration|
  (progn
    (push '|PureAlgebraicIntegration| *Packages*)
    (make-instance '|PureAlgebraicIntegrationType|)))
```

———-

1.90.71 PureAlgebraicLODE

— defclass PureAlgebraicLODEType —

```
(defclass |PureAlgebraicLODEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PureAlgebraicLODE")
   (marker :initform 'package)
   (abbreviation :initform 'ODEPAL)
   (comment :initform (list
     "In-field solution of an linear ordinary differential equation,"
     "pure algebraic case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PureAlgebraicLODE|
  (progn
    (push '|PureAlgebraicLODE| *Packages*)
    (make-instance '|PureAlgebraicLODEType|)))
```

———-

1.90.72 PushVariables

— defclass PushVariablesType —

```
(defclass |PushVariablesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PushVariables")
   (marker :initform 'package)
   (abbreviation :initform 'PUSHVAR)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PushVariables|
  (progn
    (push '|PushVariables| *Packages*)
    (make-instance '|PushVariablesType|)))
```

———-

1.91 Q

1.91.1 QuasiAlgebraicSet2

— defclass QuasiAlgebraicSet2Type —

```
(defclass |QuasiAlgebraicSet2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "QuasiAlgebraicSet2")
   (marker :initform 'package)
   (abbreviation :initform 'QALGSET2)
   (comment :initform (list
     "QuasiAlgebraicSet2 adds a function radicalSimplify"
     "which uses IdealDecompositionPackage to simplify"
     "the representation of a quasi-algebraic set.  A quasi-algebraic set"
     "is the intersection of a Zariski"
     "closed set, defined as the common zeros of a given list of"
```

```
    "polynomials (the defining polynomials for equations), and a principal"
    "Zariski open set, defined as the complement of the common"
    "zeros of a polynomial f (the defining polynomial for the inequation)."
    "Quasi-algebraic sets are implemented in the domain"
    "QuasiAlgebraicSet, where two simplification routines are"
    "provided:"
    "idealSimplify and simplify."
    "The function"
    "radicalSimplify is added"
    "for comparison study only.  Because the domain"
    "IdealDecompositionPackage provides facilities for"
    "computing with radical ideals, it is necessary to restrict"
    "the ground ring to the domain Fraction Integer,"
    "and the polynomial ring to be of type"
    "DistributedMultivariatePolynomial."
    "The routine radicalSimplify uses these to compute groebner"
    "basis of radical ideals and"
    "is inefficient and restricted when compared to the"
    "two in QuasiAlgebraicSet."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuasiAlgebraicSet2|
  (progn
    (push '|QuasiAlgebraicSet2| *Packages*)
    (make-instance '|QuasiAlgebraicSet2Type|)))
```

———-

1.91.2 QuasiComponentPackage

— defclass QuasiComponentPackageType —

```
(defclass |QuasiComponentPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "QuasiComponentPackage")
   (marker :initform 'package)
   (abbreviation :initform 'QCMPACK)
   (comment :initform (list
     "A package for removing redundant quasi-components and redundant"
     "branches when decomposing a variety by means of quasi-components"
     "of regular triangular sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuasiComponentPackage|
  (progn
    (push '|QuasiComponentPackage| *Packages*)
    (make-instance '|QuasiComponentPackageType|)))
```

———-

1.91.3 QuotientFieldCategoryFunctions2

— defclass QuotientFieldCategoryFunctions2Type —

```
(defclass |QuotientFieldCategoryFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "QuotientFieldCategoryFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'QFCAT2)
   (comment :initform (list
     "This package extends a function between integral domains"
     "to a mapping between their quotient fields."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuotientFieldCategoryFunctions2|
  (progn
    (push '|QuotientFieldCategoryFunctions2| *Packages*)
    (make-instance '|QuotientFieldCategoryFunctions2Type|)))
```

———-

1.91.4 QuaternionCategoryFunctions2

— defclass QuaternionCategoryFunctions2Type —

```
(defclass |QuaternionCategoryFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "QuaternionCategoryFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'QUATCT2)
   (comment :initform (list
     "QuaternionCategoryFunctions2 implements functions between"
     "two quaternion domains.  The function map is used by"
     "the system interpreter to coerce between quaternion types."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuaternionCategoryFunctions2|
  (progn
    (push '|QuaternionCategoryFunctions2| *Packages*)
    (make-instance '|QuaternionCategoryFunctions2Type|)))
```

———-

1.92 R

1.92.1 RadicalEigenPackage

— defclass RadicalEigenPackageType —

```
(defclass |RadicalEigenPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RadicalEigenPackage")
   (marker :initform 'package)
   (abbreviation :initform 'REP)
   (comment :initform (list
     "Package for the computation of eigenvalues and eigenvectors."
     "This package works for matrices with coefficients which are"
     "rational functions over the integers."
     "(see Fraction Polynomial Integer)."
     "The eigenvalues and eigenvectors are expressed in terms of radicals."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadicalEigenPackage|
  (progn
    (push '|RadicalEigenPackage| *Packages*)
    (make-instance '|RadicalEigenPackageType|)))
```

———-

1.92.2 RadicalSolvePackage

— defclass RadicalSolvePackageType —

```
(defclass |RadicalSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RadicalSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'SOLVERAD)
   (comment :initform (list
     "This package tries to find solutions"
     "expressed in terms of radicals for systems of equations"
     "of rational functions with coefficients in an integral domain R."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadicalSolvePackage|
  (progn
    (push '|RadicalSolvePackage| *Packages*)
    (make-instance '|RadicalSolvePackageType|)))
```

———-

1.92.3 RadixUtilities

— defclass RadixUtilitiesType —

```
(defclass |RadixUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RadixUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'RADUTIL)
   (comment :initform (list
     "This package provides tools for creating radix expansions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadixUtilities|
  (progn
    (push '|RadixUtilities| *Packages*)
    (make-instance '|RadixUtilitiesType|)))
```

———-

1.92.4 RandomDistributions

— defclass RandomDistributionsType —

```
(defclass |RandomDistributionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RandomDistributions")
   (marker :initform 'package)
   (abbreviation :initform 'RDIST)
   (comment :initform (list
     "This package exports random distributions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RandomDistributions|
  (progn
    (push '|RandomDistributions| *Packages*)
    (make-instance '|RandomDistributionsType|)))
```

———-

1.92.5 RandomFloatDistributions

— defclass RandomFloatDistributionsType —

```
(defclass |RandomFloatDistributionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RandomFloatDistributions")
   (marker :initform 'package)
   (abbreviation :initform 'RFDIST)
   (comment :initform (list
     "This package exports random floating-point distributions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RandomFloatDistributions|
  (progn
    (push '|RandomFloatDistributions| *Packages*)
    (make-instance '|RandomFloatDistributionsType|)))
```

———-

1.92.6 RandomIntegerDistributions

— defclass RandomIntegerDistributionsType —

```
(defclass |RandomIntegerDistributionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RandomIntegerDistributions")
   (marker :initform 'package)
   (abbreviation :initform 'RIDIST)
   (comment :initform (list
     "This package exports integer distributions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RandomIntegerDistributions|
  (progn
    (push '|RandomIntegerDistributions| *Packages*)
    (make-instance '|RandomIntegerDistributionsType|)))
```

———-

1.92.7 RandomNumberSource

— defclass RandomNumberSourceType —

```
(defclass |RandomNumberSourceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RandomNumberSource")
   (marker :initform 'package)
   (abbreviation :initform 'RANDSRC)
   (comment :initform (list
     "Random number generators."
     "All random numbers used in the system should originate from"
     "the same generator.  This package is intended to be the source."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RandomNumberSource|
  (progn
    (push '|RandomNumberSource| *Packages*)
    (make-instance '|RandomNumberSourceType|)))
```

————-

1.92.8 RationalFactorize

— defclass RationalFactorizeType —

```
(defclass |RationalFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'RATFACT)
   (comment :initform (list
     "Factorization of extended polynomials with rational coefficients."
     "This package implements factorization of extended polynomials"
     "whose coefficients are rational numbers. It does this by taking the"
     "lcm of the coefficients of the polynomial and creating a polynomial"
     "with integer coefficients. The algorithm in"
     "GaloisGroupFactorizer is then"
     "used to factor the integer polynomial. The result is normalized"
     "with respect to the original lcm of the denominators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFactorize|
  (progn
    (push '|RationalFactorize| *Packages*)
    (make-instance '|RationalFactorizeType|)))
```

————-

1.92.9 RationalFunction

— defclass RationalFunctionType —

```
(defclass |RationalFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunction")
   (marker :initform 'package)
   (abbreviation :initform 'RF)
   (comment :initform (list
     "Utilities that provide the same top-level manipulations on"
     "fractions than on polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunction|
  (progn
    (push '|RationalFunction| *Packages*)
    (make-instance '|RationalFunctionType|)))
```

———-

1.92.10 RationalFunctionDefiniteIntegration

— defclass RationalFunctionDefiniteIntegrationType —

```
(defclass |RationalFunctionDefiniteIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionDefiniteIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'DEFINTRF)
   (comment :initform (list
     "Definite integration of rational functions."
     "RationalFunctionDefiniteIntegration provides functions to"
     "compute definite integrals of rational functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionDefiniteIntegration|
  (progn
    (push '|RationalFunctionDefiniteIntegration| *Packages*)
    (make-instance '|RationalFunctionDefiniteIntegrationType|)))
```

———-

1.92.11 RationalFunctionFactor

— defclass RationalFunctionFactorType —

```
(defclass |RationalFunctionFactorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionFactor")
   (marker :initform 'package)
   (abbreviation :initform 'RFFACT)
   (comment :initform (list
     "Factorization of univariate polynomials with coefficients which"
     "are rational functions with integer coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionFactor|
  (progn
    (push '|RationalFunctionFactor| *Packages*)
    (make-instance '|RationalFunctionFactorType|)))
```

———-

1.92.12 RationalFunctionFactorizer

— defclass RationalFunctionFactorizerType —

```
(defclass |RationalFunctionFactorizerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionFactorizer")
   (marker :initform 'package)
   (abbreviation :initform 'RFFACTOR)
   (comment :initform (list
     "\spadtype{RationalFunctionFactorizer} contains the factor function"
     "(called factorFraction) which factors fractions of polynomials by factoring"
     "the numerator and denominator. Since any non zero fraction is a unit"
     "the usual factor operation will just return the original fraction."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionFactorizer|
  (progn
    (push '|RationalFunctionFactorizer| *Packages*)
    (make-instance '|RationalFunctionFactorizerType|)))
```

———-

1.92.13 RationalFunctionIntegration

— defclass RationalFunctionIntegrationType —

```
(defclass |RationalFunctionIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTRF)
   (comment :initform (list
     "This package provides functions for the integration of rational functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionIntegration|
  (progn
    (push '|RationalFunctionIntegration| *Packages*)
    (make-instance '|RationalFunctionIntegrationType|)))
```

———-

1.92.14 RationalFunctionLimitPackage

— defclass RationalFunctionLimitPackageType —

```
(defclass |RationalFunctionLimitPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionLimitPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LIMITRF)
   (comment :initform (list
     "Computation of limits for rational functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionLimitPackage|
  (progn
    (push '|RationalFunctionLimitPackage| *Packages*)
    (make-instance '|RationalFunctionLimitPackageType|)))
```

———-

1.92.15 RationalFunctionSign

— defclass RationalFunctionSignType —

```
(defclass |RationalFunctionSignType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionSign")
   (marker :initform 'package)
   (abbreviation :initform 'SIGNRF)
   (comment :initform (list
     "Find the sign of a rational function around a point or infinity."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionSign|
  (progn
    (push '|RationalFunctionSign| *Packages*)
    (make-instance '|RationalFunctionSignType|)))
```

———-

1.92.16 RationalFunctionSum

— defclass RationalFunctionSumType —

```
(defclass |RationalFunctionSumType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionSum")
   (marker :initform 'package)
   (abbreviation :initform 'SUMRF)
   (comment :initform (list
     "Computes sums of rational functions"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionSum|
  (progn
    (push '|RationalFunctionSum| *Packages*)
    (make-instance '|RationalFunctionSumType|)))
```

———-

1.92.17 RationalIntegration

— defclass RationalIntegrationType —

```
(defclass |RationalIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTRAT)
   (comment :initform (list
     "This package provides functions for the base case of the Risch algorithm."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalIntegration|
  (progn
    (push '|RationalIntegration| *Packages*)
    (make-instance '|RationalIntegrationType|)))
```

———-

1.92.18 RationalInterpolation

— defclass RationalInterpolationType —

```
(defclass |RationalInterpolationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalInterpolation")
   (marker :initform 'package)
   (abbreviation :initform 'RINTERP)
   (comment :initform (list
     "This package exports rational interpolation algorithms"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalInterpolation|
  (progn
    (push '|RationalInterpolation| *Packages*)
    (make-instance '|RationalInterpolationType|)))
```

———-

1.92.19 RationalLODE

— defclass RationalLODEType —

```
(defclass |RationalLODEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalLODE")
   (marker :initform 'package)
   (abbreviation :initform 'ODERAT)
   (comment :initform (list
     "RationalLODE provides functions for in-field solutions of linear"
     "ordinary differential equations, in the rational case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalLODE|
  (progn
    (push '|RationalLODE| *Packages*)
    (make-instance '|RationalLODEType|)))
```

———-

1.92.20 RationalRetractions

— defclass RationalRetractionsType —

```
(defclass |RationalRetractionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalRetractions")
   (marker :initform 'package)
   (abbreviation :initform 'RATRET)
   (comment :initform (list
     "Rational number testing and retraction functions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalRetractions|
  (progn
    (push '|RationalRetractions| *Packages*)
    (make-instance '|RationalRetractionsType|)))
```

———-

1.92.21 RationalRicDE

— defclass RationalRicDEType —

```
(defclass |RationalRicDEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalRicDE")
   (marker :initform 'package)
   (abbreviation :initform 'ODERTRIC)
   (comment :initform (list
     "In-field solution of Riccati equations, rational case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalRicDE|
  (progn
    (push '|RationalRicDE| *Packages*)
    (make-instance '|RationalRicDEType|)))
```

———-

1.92.22 RationalUnivariateRepresentationPackage

— defclass RationalUnivariateRepresentationPackageType —

```
(defclass |RationalUnivariateRepresentationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalUnivariateRepresentationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'RURPK)
   (comment :initform (list
     "A package for computing the rational univariate representation"
     "of a zero-dimensional algebraic variety given by a regular"
     "triangular set. This package is essentially an interface for the"
     "InternalRationalUnivariateRepresentationPackage constructor."
     "It is used in the ZeroDimensionalSolvePackage"
     "for solving polynomial systems with finitely many solutions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalUnivariateRepresentationPackage|
  (progn
    (push '|RationalUnivariateRepresentationPackage| *Packages*)
    (make-instance '|RationalUnivariateRepresentationPackageType|)))
```

———-

1.92.23 RealPolynomialUtilitiesPackage

— defclass RealPolynomialUtilitiesPackageType —

```
(defclass |RealPolynomialUtilitiesPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RealPolynomialUtilitiesPackage")
   (marker :initform 'package)
   (abbreviation :initform 'POLUTIL)
   (comment :initform (list
     "RealPolynomialUtilitiesPackage provides common functions used"
     "by interval coding."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealPolynomialUtilitiesPackage|
  (progn
    (push '|RealPolynomialUtilitiesPackage| *Packages*)
    (make-instance '|RealPolynomialUtilitiesPackageType|)))
```

———-

1.92.24 RealSolvePackage

— defclass RealSolvePackageType —

```
(defclass |RealSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RealSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'REALSOLV)
   (comment :initform (list
     "This package provides numerical solutions of systems of"
     "polynomial equations for use in ACPLOT"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealSolvePackage|
  (progn
    (push '|RealSolvePackage| *Packages*)
    (make-instance '|RealSolvePackageType|)))
```

———-

1.92.25 RealZeroPackage

— defclass RealZeroPackageType —

```
(defclass |RealZeroPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RealZeroPackage")
   (marker :initform 'package)
   (abbreviation :initform 'REAL0)
   (comment :initform (list
     "This package provides functions for finding the real zeros"
     "of univariate polynomials over the integers to arbitrary user-specified"
     "precision. The results are returned as a list of"
     "isolating intervals which are expressed as records with"
     "'left' and 'right' rational number components."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealZeroPackage|
  (progn
    (push '|RealZeroPackage| *Packages*)
    (make-instance '|RealZeroPackageType|)))
```

———-

1.92.26 RealZeroPackageQ

— defclass RealZeroPackageQType —

```
(defclass |RealZeroPackageQType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RealZeroPackageQ")
   (marker :initform 'package)
   (abbreviation :initform 'REAL0Q)
   (comment :initform (list
     "This package provides functions for finding the real zeros of univariate"
     "polynomials over the rational numbers to arbitrary user-specified"
     "precision. The results are returned as a list of isolating intervals,"
     "expressed as records with 'left' and 'right' rational number components."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealZeroPackageQ|
  (progn
    (push '|RealZeroPackageQ| *Packages*)
    (make-instance '|RealZeroPackageQType|)))
```

———-

1.92.27 RectangularMatrixCategoryFunctions2

— defclass RectangularMatrixCategoryFunctions2Type —

```
(defclass |RectangularMatrixCategoryFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RectangularMatrixCategoryFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'RMCAT2)
   (comment :initform (list
     "RectangularMatrixCategoryFunctions2 provides functions between"
     "two matrix domains.  The functions provided are map and"
     "reduce."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RectangularMatrixCategoryFunctions2|
  (progn
    (push '|RectangularMatrixCategoryFunctions2| *Packages*)
    (make-instance '|RectangularMatrixCategoryFunctions2Type|)))
```

————-

1.92.28 RecurrenceOperator

— defclass RecurrenceOperatorType —

```
(defclass |RecurrenceOperatorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RecurrenceOperator")
   (marker :initform 'package)
   (abbreviation :initform 'RECOP)
   (comment :initform (list
     "This package provides an operator for the n-th term of a recurrence and an"
     "operator for the coefficient of x^n in a function specified by a functional"
     "equation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RecurrenceOperator|
  (progn
    (push '|RecurrenceOperator| *Packages*)
    (make-instance '|RecurrenceOperatorType|)))
```

————-

1.92.29 ReducedDivisor

— defclass ReducedDivisorType —

```
(defclass |ReducedDivisorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ReducedDivisor")
   (marker :initform 'package)
   (abbreviation :initform 'RDIV)
   (comment :initform (list
     "Finds the order of a divisor over a finite field"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ReducedDivisor|
  (progn
    (push '|ReducedDivisor| *Packages*)
    (make-instance '|ReducedDivisorType|)))
```

———-

1.92.30 ReduceLODE

— defclass ReduceLODEType —

```
(defclass |ReduceLODEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ReduceLODE")
   (marker :initform 'package)
   (abbreviation :initform 'ODERED)
   (comment :initform (list
     "Elimination of an algebraic from the coefficentss"
     "of a linear ordinary differential equation."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ReduceLODE|
  (progn
    (push '|ReduceLODE| *Packages*)
    (make-instance '|ReduceLODEType|)))
```

———-

1.92.31 ReductionOfOrder

— defclass ReductionOfOrderType —

```
(defclass |ReductionOfOrderType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ReductionOfOrder")
   (marker :initform 'package)
   (abbreviation :initform 'REDORDER)
   (comment :initform (list
     "ReductionOfOrder provides"
     "functions for reducing the order of linear ordinary differential equations"
     "once some solutions are known."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ReductionOfOrder|
  (progn
    (push '|ReductionOfOrder| *Packages*)
    (make-instance '|ReductionOfOrderType|)))
```

———-

1.92.32 RegularSetDecompositionPackage

— defclass RegularSetDecompositionPackageType —

```
(defclass |RegularSetDecompositionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RegularSetDecompositionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'RSDCMPK)
   (comment :initform (list
     "A package providing a new algorithm for solving polynomial systems"
     "by means of regular chains. Two ways of solving are proposed:"
     "in the sense of Zariski closure (like in Kalkbrener's algorithm)"
     "or in the sense of the regular zeros (like in Wu, Wang or Lazard"
     "methods). This algorithm is valid for any type"
     "of regular set. It does not care about the way a polynomial is"
     "added in an regular set, or how two quasi-components are compared"
     "(by an inclusion-test), or how the invertibility test is made in"
     "the tower of simple extensions associated with a regular set."
     "These operations are realized respectively by the domain TS"
     "and the packages"
     "QCMPACK(R,E,V,P,TS) and RSETGCD(R,E,V,P,TS)."
     "The same way it does not care about the way univariate polynomial"
     "gcd (with coefficients in the tower of simple extensions associated"
     "with a regular set) are computed. The only requirement is that these"
     "gcd need to have invertible initials (normalized or not)."
     "WARNING. There is no need for a user to call directly any operation"
     "of this package since they can be accessed by the domain TS."
     "Thus, the operations of this package are not documented."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularSetDecompositionPackage|
  (progn
    (push '|RegularSetDecompositionPackage| *Packages*)
    (make-instance '|RegularSetDecompositionPackageType|)))
```

———-

1.92.33 RegularTriangularSetGcdPackage

— defclass RegularTriangularSetGcdPackageType —

```
(defclass |RegularTriangularSetGcdPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RegularTriangularSetGcdPackage")
   (marker :initform 'package)
   (abbreviation :initform 'RSETGCD)
   (comment :initform (list
     "An internal package for computing gcds and resultants of univariate"
     "polynomials with coefficients in a tower of simple extensions of a field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularTriangularSetGcdPackage|
  (progn
    (push '|RegularTriangularSetGcdPackage| *Packages*)
    (make-instance '|RegularTriangularSetGcdPackageType|)))
```

———-

1.92.34 RepeatedDoubling

— defclass RepeatedDoublingType —

```
(defclass |RepeatedDoublingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RepeatedDoubling")
   (marker :initform 'package)
   (abbreviation :initform 'REPDB)
   (comment :initform (list
     "Implements multiplication by repeated addition"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepeatedDoubling|
  (progn
    (push '|RepeatedDoubling| *Packages*)
    (make-instance '|RepeatedDoublingType|)))
```

———-

1.92.35 RepeatedSquaring

— defclass RepeatedSquaringType —

```
(defclass |RepeatedSquaringType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RepeatedSquaring")
   (marker :initform 'package)
   (abbreviation :initform 'REPSQ)
   (comment :initform (list
     "Implements exponentiation by repeated squaring"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepeatedSquaring|
  (progn
    (push '|RepeatedSquaring| *Packages*)
    (make-instance '|RepeatedSquaringType|)))
```

———-

1.92.36 RepresentationPackage1

— defclass RepresentationPackage1Type —

```
(defclass |RepresentationPackage1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RepresentationPackage1")
   (marker :initform 'package)
   (abbreviation :initform 'REP1)
   (comment :initform (list
     "RepresentationPackage1 provides functions for representation theory"
     "for finite groups and algebras."
     "The package creates permutation representations and uses tensor products"
     "and its symmetric and antisymmetric components to create new"
     "representations of larger degree from given ones."
     "Note that instead of having parameters from Permutation"
     "this package allows list notation of permutations as well:"
     "for example [1,4,3,2] denotes permutes 2 and 4 and fixes 1 and 3."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepresentationPackage1|
  (progn
    (push '|RepresentationPackage1| *Packages*)
    (make-instance '|RepresentationPackage1Type|)))
```

———-

1.92.37 RepresentationPackage2

— defclass RepresentationPackage2Type —

```
(defclass |RepresentationPackage2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RepresentationPackage2")
   (marker :initform 'package)
   (abbreviation :initform 'REP2)
   (comment :initform (list
     "RepresentationPackage2 provides functions for working with"
     "modular representations of finite groups and algebra."
     "The routines in this package are created, using ideas of R. Parker,"
     "(the meat-Axe) to get smaller representations from bigger ones,"
     "finding sub- and factormodules, or to show, that such the"
     "representations are irreducible."
     "Note that most functions are randomized functions of Las Vegas type"
     "every answer is correct, but with small probability"
     "the algorithm fails to get an answer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepresentationPackage2|
  (progn
    (push '|RepresentationPackage2| *Packages*)
    (make-instance '|RepresentationPackage2Type|)))
```

———-

1.92.38 ResolveLatticeCompletion

— defclass ResolveLatticeCompletionType —

```
(defclass |ResolveLatticeCompletionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ResolveLatticeCompletion")
   (marker :initform 'package)
   (abbreviation :initform 'RESLATC)
   (comment :initform (list
     "This package provides coercions for the special types Exit"
     "and Void."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ResolveLatticeCompletion|
  (progn
    (push '|ResolveLatticeCompletion| *Packages*)
    (make-instance '|ResolveLatticeCompletionType|)))
```

———-

1.92.39 RetractSolvePackage

— defclass RetractSolvePackageType —

```
(defclass |RetractSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RetractSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'RETSOL)
   (comment :initform (list
     "RetractSolvePackage is an interface to SystemSolvePackage"
     "that attempts to retract the coefficients of the equations before"
     "solving."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RetractSolvePackage|
  (progn
    (push '|RetractSolvePackage| *Packages*)
    (make-instance '|RetractSolvePackageType|)))
```

———-

1.92.40 RootsFindingPackage

— defclass RootsFindingPackageType —

```
(defclass |RootsFindingPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RootsFindingPackage")
   (marker :initform 'package)
   (abbreviation :initform 'RFP)
   (comment :initform (list
     "This pacackage finds all the roots of a polynomial. If the constant field is"
     "not large enough then it returns the list of found zeros and the degree"
     "of the extension need to find the other roots missing. If the return"
     "degree is 1 then all the roots have been found. If 0 is return"
     "for the extension degree then there are an infinite number of zeros,"
     "that is you ask for the zeroes of 0. In the case of infinite field"
     "a list of all found zeros is kept and for each other call of a function"
     "that finds zeroes, a check is made on that list; this is to keep"
     "a kind of 'canonical' representation of the elements."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RootsFindingPackage|
  (progn
    (push '|RootsFindingPackage| *Packages*)
    (make-instance '|RootsFindingPackageType|)))
```

———-

1.93 S

1.93.1 SAERationalFunctionAlgFactor

— defclass SAERationalFunctionAlgFactorType —

```
(defclass |SAERationalFunctionAlgFactorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SAERationalFunctionAlgFactor")
   (marker :initform 'package)
   (abbreviation :initform 'SAERFFC)
   (comment :initform (list
     "Factorization of univariate polynomials with coefficients in an"
     "algebraic extension of \spadtype{Fraction Polynomial Integer}."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SAERationalFunctionAlgFactor|
  (progn
    (push '|SAERationalFunctionAlgFactor| *Packages*)
    (make-instance '|SAERationalFunctionAlgFactorType|)))
```

———-

1.93.2 ScriptFormulaFormat1

— defclass ScriptFormulaFormat1Type —

```
(defclass |ScriptFormulaFormat1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ScriptFormulaFormat1")
   (marker :initform 'package)
   (abbreviation :initform 'FORMULA1)
   (comment :initform (list
     "ScriptFormulaFormat1 provides a utility coercion for"
     "changing to SCRIPT formula format anything that has a coercion to"
     "the standard output format."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ScriptFormulaFormat1|
  (progn
    (push '|ScriptFormulaFormat1| *Packages*)
    (make-instance '|ScriptFormulaFormat1Type|)))
```

———-

1.93.3 SegmentBindingFunctions2

— defclass SegmentBindingFunctions2Type —

```
(defclass |SegmentBindingFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SegmentBindingFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'SEGBIND2)
   (comment :initform (list
     "This package provides operations for mapping functions onto"
     "SegmentBindings."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SegmentBindingFunctions2|
  (progn
    (push '|SegmentBindingFunctions2| *Packages*)
    (make-instance '|SegmentBindingFunctions2Type|)))
```

———-

1.93.4 SegmentFunctions2

— defclass SegmentFunctions2Type —

```
(defclass |SegmentFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SegmentFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'SEG2)
   (comment :initform (list
     "This package provides operations for mapping functions onto segments."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SegmentFunctions2|
  (progn
    (push '|SegmentFunctions2| *Packages*)
    (make-instance '|SegmentFunctions2Type|)))
```

———-

1.93.5 SimpleAlgebraicExtensionAlgFactor

— defclass SimpleAlgebraicExtensionAlgFactorType —

```
(defclass |SimpleAlgebraicExtensionAlgFactorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SimpleAlgebraicExtensionAlgFactor")
   (marker :initform 'package)
   (abbreviation :initform 'SAEFACT)
   (comment :initform (list
     "Factorization of univariate polynomials with coefficients in an"
     "algebraic extension of the rational numbers (Fraction Integer)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleAlgebraicExtensionAlgFactor|
  (progn
    (push '|SimpleAlgebraicExtensionAlgFactor| *Packages*)
    (make-instance '|SimpleAlgebraicExtensionAlgFactorType|)))
```

———-

1.93.6 SimplifyAlgebraicNumberConvertPackage

— defclass SimplifyAlgebraicNumberConvertPackageType —

```
(defclass |SimplifyAlgebraicNumberConvertPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SimplifyAlgebraicNumberConvertPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SIMPAN)
   (comment :initform (list
     "Package to allow simplify to be called on AlgebraicNumbers"
     "by converting to EXPR(INT)"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimplifyAlgebraicNumberConvertPackage|
  (progn
    (push '|SimplifyAlgebraicNumberConvertPackage| *Packages*)
    (make-instance '|SimplifyAlgebraicNumberConvertPackageType|)))
```

———-

1.93.7 SmithNormalForm

— defclass SmithNormalFormType —

```
(defclass |SmithNormalFormType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SmithNormalForm")
   (marker :initform 'package)
   (abbreviation :initform 'SMITH)
   (comment :initform (list
     "SmithNormalForm is a package"
     "which provides some standard canonical forms for matrices."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SmithNormalForm|
  (progn
    (push '|SmithNormalForm| *Packages*)
    (make-instance '|SmithNormalFormType|)))
```

———-

1.93.8 SortedCache

— defclass SortedCacheType —

```
(defclass |SortedCacheType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SortedCache")
   (marker :initform 'package)
   (abbreviation :initform 'SCACHE)
   (comment :initform (list
     "A sorted cache of a cachable set S is a dynamic structure that"
     "keeps the elements of S sorted and assigns an integer to each"
     "element of S once it is in the cache. This way, equality and ordering"
     "on S are tested directly on the integers associated with the elements"
     "of S, once they have been entered in the cache."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SortedCache|
  (progn
    (push '|SortedCache| *Packages*)
    (make-instance '|SortedCacheType|)))
```

———-

1.93.9 SortPackage

— defclass SortPackageType —

```
(defclass |SortPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SortPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SORTPAK)
   (comment :initform (list
     "This package exports sorting algorithnms"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SortPackage|
  (progn
    (push '|SortPackage| *Packages*)
    (make-instance '|SortPackageType|)))
```

———-

1.93.10 SparseUnivariatePolynomialFunctions2

— defclass SparseUnivariatePolynomialFunctions2Type —

```
(defclass |SparseUnivariatePolynomialFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SparseUnivariatePolynomialFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'SUP2)
   (comment :initform (list
     "This package lifts a mapping from coefficient rings R to S to"
     "a mapping from sparse univariate polynomial over R to"
     "a sparse univariate polynomial over S."
     "Note that the mapping is assumed"
     "to send zero to zero, since it will only be applied to the non-zero"
     "coefficients of the polynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariatePolynomialFunctions2|
  (progn
    (push '|SparseUnivariatePolynomialFunctions2| *Packages*)
    (make-instance '|SparseUnivariatePolynomialFunctions2Type|)))
```

———-

1.93.11 SpecialOutputPackage

— defclass SpecialOutputPackageType —

```
(defclass |SpecialOutputPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SpecialOutputPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SPECOUT)
   (comment :initform (list
     "SpecialOutputPackage allows FORTRAN, Tex and"
     "Script Formula Formatter output from programs."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SpecialOutputPackage|
  (progn
    (push '|SpecialOutputPackage| *Packages*)
    (make-instance '|SpecialOutputPackageType|)))
```

———-

1.93.12 SquareFreeQuasiComponentPackage

— defclass SquareFreeQuasiComponentPackageType —

```
(defclass |SquareFreeQuasiComponentPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SquareFreeQuasiComponentPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SFQCMPK)
   (comment :initform (list
     "A internal package for removing redundant quasi-components and redundant"
     "branches when decomposing a variety by means of quasi-components"
     "of regular triangular sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeQuasiComponentPackage|
  (progn
    (push '|SquareFreeQuasiComponentPackage| *Packages*)
    (make-instance '|SquareFreeQuasiComponentPackageType|)))
```

———-

1.93.13 SquareFreeRegularSetDecompositionPackage

— defclass SquareFreeRegularSetDecompositionPackageType —

```
(defclass |SquareFreeRegularSetDecompositionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SquareFreeRegularSetDecompositionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SRDCMPK)
   (comment :initform (list
     "A package providing a new algorithm for solving polynomial systems"
     "by means of regular chains. Two ways of solving are provided:"
     "in the sense of Zariski closure (like in Kalkbrener's algorithm)"
     "or in the sense of the regular zeros (like in Wu, Wang or Lazard-"
     "Moreno methods). This algorithm is valid for any type"
     "of regular set. It does not care about the way a polynomial is"
     "added in an regular set, or how two quasi-components are compared"
     "(by an inclusion-test), or how the invertibility test is made in"
     "the tower of simple extensions associated with a regular set."
     "These operations are realized respectively by the domain TS"
     "and the packages QCMPPK(R,E,V,P,TS) and RSETGCD(R,E,V,P,TS)."
     "The same way it does not care about the way univariate polynomial"
     "gcds (with coefficients in the tower of simple extensions associated"
     "with a regular set) are computed. The only requirement is that these"
     "gcds need to have invertible initials (normalized or not)."
     "WARNING. There is no need for a user to call directly any operation"
     "of this package since they can be accessed by the domain TS."
     "Thus, the operations of this package are not documented."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularSetDecompositionPackage|
  (progn
    (push '|SquareFreeRegularSetDecompositionPackage| *Packages*)
    (make-instance '|SquareFreeRegularSetDecompositionPackageType|)))
```

———-

1.93.14 SquareFreeRegularTriangularSetGcdPackage

— defclass SquareFreeRegularTriangularSetGcdPackageType —

```
(defclass |SquareFreeRegularTriangularSetGcdPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SquareFreeRegularTriangularSetGcdPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SFRGCD)
   (comment :initform (list
     "A internal package for computing gcds and resultants of univariate"
     "polynomials with coefficients in a tower of simple extensions of a field."
     "There is no need to use directly this package since its main operations are"
     "available from TS."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularTriangularSetGcdPackage|
  (progn
    (push '|SquareFreeRegularTriangularSetGcdPackage| *Packages*)
    (make-instance '|SquareFreeRegularTriangularSetGcdPackageType|)))
```

———-

1.93.15 StorageEfficientMatrixOperations

— defclass StorageEfficientMatrixOperationsType —

```
(defclass |StorageEfficientMatrixOperationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StorageEfficientMatrixOperations")
   (marker :initform 'package)
   (abbreviation :initform 'MATSTOR)
   (comment :initform (list
     "This package provides standard arithmetic operations on matrices."
     "The functions in this package store the results of computations"
     "in existing matrices, rather than creating new matrices.  This"
     "package works only for matrices of type Matrix and uses the"
     "internal representation of this type."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StorageEfficientMatrixOperations|
  (progn
    (push '|StorageEfficientMatrixOperations| *Packages*)
    (make-instance '|StorageEfficientMatrixOperationsType|)))
```

———-

1.93.16 StreamFunctions1

— defclass StreamFunctions1Type —

```
(defclass |StreamFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'STREAM1)
   (comment :initform (list
    "Functions defined on streams with entries in one set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamFunctions1|
  (progn
    (push '|StreamFunctions1| *Packages*)
    (make-instance '|StreamFunctions1Type|)))
```

————-

1.93.17 StreamFunctions2

— defclass StreamFunctions2Type —

```
(defclass |StreamFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'STREAM2)
   (comment :initform (list
     "Functions defined on streams with entries in two sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamFunctions2|
  (progn
    (push '|StreamFunctions2| *Packages*)
    (make-instance '|StreamFunctions2Type|)))
```

————-

1.93.18 StreamFunctions3

— defclass StreamFunctions3Type —

```
(defclass |StreamFunctions3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamFunctions3")
   (marker :initform 'package)
   (abbreviation :initform 'STREAM3)
   (comment :initform (list
     "Functions defined on streams with entries in three sets."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamFunctions3|
  (progn
    (push '|StreamFunctions3| *Packages*)
    (make-instance '|StreamFunctions3Type|)))
```

———-

1.93.19 StreamInfiniteProduct

— defclass StreamInfiniteProductType —

```
(defclass |StreamInfiniteProductType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamInfiniteProduct")
   (marker :initform 'package)
   (abbreviation :initform 'STINPROD)
   (comment :initform (list
     "This package computes infinite products of Taylor series over an"
     "integral domain of characteristic 0.  Here Taylor series are"
     "represented by streams of Taylor coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamInfiniteProduct|
  (progn
    (push '|StreamInfiniteProduct| *Packages*)
    (make-instance '|StreamInfiniteProductType|)))
```

———-

1.93.20 StreamTaylorSeriesOperations

— defclass StreamTaylorSeriesOperationsType —

```
(defclass |StreamTaylorSeriesOperationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamTaylorSeriesOperations")
   (marker :initform 'package)
   (abbreviation :initform 'STTAYLOR)
   (comment :initform (list
     "StreamTaylorSeriesOperations implements Taylor series arithmetic,"
     "where a Taylor series is represented by a stream of its coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
  (addlist :initform nil)))

(defvar |StreamTaylorSeriesOperations|
  (progn
    (push '|StreamTaylorSeriesOperations| *Packages*)
    (make-instance '|StreamTaylorSeriesOperationsType|)))
```

———-

1.93.21 StreamTensor

— defclass StreamTensorType —

```
(defclass |StreamTensorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamTensor")
   (marker :initform 'package)
   (abbreviation :initform 'STNSR)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamTensor|
  (progn
    (push '|StreamTensor| *Packages*)
    (make-instance '|StreamTensorType|)))
```

———-

1.93.22 StreamTranscendentalFunctions

— defclass StreamTranscendentalFunctionsType —

```
(defclass |StreamTranscendentalFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamTranscendentalFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'STTF)
   (comment :initform (list
     "StreamTranscendentalFunctions implements transcendental functions on"
     "Taylor series, where a Taylor series is represented by a stream of"
     "its coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |StreamTranscendentalFunctions|
  (progn
    (push '|StreamTranscendentalFunctions| *Packages*)
    (make-instance '|StreamTranscendentalFunctionsType|)))
```

———-

1.93.23 StreamTranscendentalFunctionsNonCommutative

— defclass StreamTranscendentalFunctionsNonCommutativeType —

```
(defclass |StreamTranscendentalFunctionsNonCommutativeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamTranscendentalFunctionsNonCommutative")
   (marker :initform 'package)
   (abbreviation :initform 'STTFNC)
   (comment :initform (list
     "StreamTranscendentalFunctionsNonCommutative implements transcendental"
     "functions on Taylor series over a non-commutative ring, where a Taylor"
     "series is represented by a stream of its coefficients."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamTranscendentalFunctionsNonCommutative|
  (progn
    (push '|StreamTranscendentalFunctionsNonCommutative| *Packages*)
    (make-instance '|StreamTranscendentalFunctionsNonCommutativeType|)))
```

———-

1.93.24 StructuralConstantsPackage

— defclass StructuralConstantsPackageType —

```
(defclass |StructuralConstantsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StructuralConstantsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SCPKG)
   (comment :initform (list
     "StructuralConstantsPackage provides functions creating"
     "structural constants from a multiplication tables or a basis"
     "of a matrix algebra and other useful functions in this context."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |StructuralConstantsPackage|
  (progn
    (push '|StructuralConstantsPackage| *Packages*)
    (make-instance '|StructuralConstantsPackageType|)))
```

———-

1.93.25 SturmHabichtPackage

— defclass SturmHabichtPackageType —

```
(defclass |SturmHabichtPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SturmHabichtPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SHP)
   (comment :initform (list
     "This package produces functions for counting etc. real roots of univariate"
     "polynomials in x over R, which must be an OrderedIntegralDomain"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SturmHabichtPackage|
  (progn
    (push '|SturmHabichtPackage| *Packages*)
    (make-instance '|SturmHabichtPackageType|)))
```

———-

1.93.26 SubResultantPackage

— defclass SubResultantPackageType —

```
(defclass |SubResultantPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SubResultantPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SUBRESP)
   (comment :initform (list
     "This package computes the subresultants of two polynomials which is needed"
     "for the `Lazard Rioboo' enhancement to Tragers integrations formula"
     "For efficiency reasons this has been rewritten to call Lionel Ducos"
     "package which is currently the best one."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |SubResultantPackage|
  (progn
    (push '|SubResultantPackage| *Packages*)
    (make-instance '|SubResultantPackageType|)))
```

———-

1.93.27 SupFractionFactorizer

— defclass SupFractionFactorizerType —

```
(defclass |SupFractionFactorizerType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SupFractionFactorizer")
   (marker :initform 'package)
   (abbreviation :initform 'SUPFRACF)
   (comment :initform (list
     "SupFractionFactorize  contains the factor function for univariate"
     "polynomials over the quotient field of a ring S such that the package"
     "MultivariateFactorize works for S"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SupFractionFactorizer|
  (progn
    (push '|SupFractionFactorizer| *Packages*)
    (make-instance '|SupFractionFactorizerType|)))
```

———-

1.93.28 SystemODESolver

— defclass SystemODESolverType —

```
(defclass |SystemODESolverType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SystemODESolver")
   (marker :initform 'package)
   (abbreviation :initform 'ODESYS)
   (comment :initform (list
     "SystemODESolver provides tools for triangulating"
     "and solving some systems of linear ordinary differential equations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SystemODESolver|
  (progn
    (push '|SystemODESolver| *Packages*)
    (make-instance '|SystemODESolverType|)))
```

———-

1.93.29 SystemSolvePackage

— defclass SystemSolvePackageType —

```
(defclass |SystemSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SystemSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'SYSSOLP)
   (comment :initform (list
     "Symbolic solver for systems of rational functions with coefficients"
     "in an integral domain R."
     "The systems are solved in the field of rational functions over R."
     "Solutions are exact of the form variable = value when the value is"
     "a member of the coefficient domain R. Otherwise the solutions"
     "are implicitly expressed as roots of univariate polynomial equations over R."
     "Care is taken to guarantee that the denominators of the input"
     "equations do not vanish on the solution sets."
     "The arguments to solve can either be given as equations or"
     "as rational functions interpreted as equal"
     "to zero. The user can specify an explicit list of symbols to"
     "be solved for, treating all other symbols appearing as parameters"
     "or omit the list of symbols in which case the system tries to"
     "solve with respect to all symbols appearing in the input."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SystemSolvePackage|
  (progn
    (push '|SystemSolvePackage| *Packages*)
    (make-instance '|SystemSolvePackageType|)))
```

———-

1.93.30 SymmetricGroupCombinatoricFunctions

— defclass SymmetricGroupCombinatoricFunctionsType —

```
(defclass |SymmetricGroupCombinatoricFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SymmetricGroupCombinatoricFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'SGCF)
   (comment :initform (list
     "SymmetricGroupCombinatoricFunctions contains combinatoric"
     "functions concerning symmetric groups and representation"
     "theory: list young tableaus, improper partitions, subsets"
     "bijection of Coleman."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SymmetricGroupCombinatoricFunctions|
  (progn
    (push '|SymmetricGroupCombinatoricFunctions| *Packages*)
    (make-instance '|SymmetricGroupCombinatoricFunctionsType|)))
```

——–-

1.93.31 SymmetricFunctions

— defclass SymmetricFunctionsType —

```
(defclass |SymmetricFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SymmetricFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'SYMFUNC)
   (comment :initform (list
     "Computes all the symmetric functions in n variables."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SymmetricFunctions|
  (progn
    (push '|SymmetricFunctions| *Packages*)
    (make-instance '|SymmetricFunctionsType|)))
```

——–-

1.94 T

1.94.1 TableauxBumpers

— defclass TableauxBumpersType —

```
(defclass |TableauxBumpersType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TableauxBumpers")
   (marker :initform 'package)
   (abbreviation :initform 'TABLBUMP)
   (comment :initform (list
     "TableauBumpers implements the Schenstead-Knuth"
     "correspondence between sequences and pairs of Young tableaux."
     "The 2 Young tableaux are represented as a single tableau with"
     "pairs as components."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TableauxBumpers|
  (progn
    (push '|TableauxBumpers| *Packages*)
    (make-instance '|TableauxBumpersType|)))
```

————-

1.94.2 TabulatedComputationPackage

— defclass TabulatedComputationPackageType —

```
(defclass |TabulatedComputationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TabulatedComputationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'TBCMPPK)
   (comment :initform (list
     "TabulatedComputationPackage(Key ,Entry) provides some modest support"
     "for dealing with operations with type Key -> Entry. The result of"
     "such operations can be stored and retrieved with this package by using"
     "a hash-table. The user does not need to worry about the management of"
     "this hash-table. However, onnly one hash-table is built by calling"
     "TabulatedComputationPackage(Key ,Entry)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TabulatedComputationPackage|
  (progn
    (push '|TabulatedComputationPackage| *Packages*)
    (make-instance '|TabulatedComputationPackageType|)))
```

————-

1.94.3 TangentExpansions

— defclass TangentExpansionsType —

```
(defclass |TangentExpansionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TangentExpansions")
   (marker :initform 'package)
   (abbreviation :initform 'TANEXP)
   (comment :initform (list
     "Expands tangents of sums and scalar products."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TangentExpansions|
  (progn
    (push '|TangentExpansions| *Packages*)
    (make-instance '|TangentExpansionsType|)))
```

———-

1.94.4 TaylorSolve

— defclass TaylorSolveType —

```
(defclass |TaylorSolveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TaylorSolve")
   (marker :initform 'package)
   (abbreviation :initform 'UTSSOL)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TaylorSolve|
  (progn
    (push '|TaylorSolve| *Packages*)
    (make-instance '|TaylorSolveType|)))
```

———-

1.94.5 TemplateUtilities

— defclass TemplateUtilitiesType —

```
(defclass |TemplateUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TemplateUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'TEMUTL)
   (comment :initform (list
     "This package provides functions for template manipulation"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TemplateUtilities|
  (progn
    (push '|TemplateUtilities| *Packages*)
    (make-instance '|TemplateUtilitiesType|)))
```

———-

1.94.6 TexFormat1

— defclass TexFormat1Type —

```
(defclass |TexFormat1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TexFormat1")
   (marker :initform 'package)
   (abbreviation :initform 'TEX1)
   (comment :initform (list
     "TexFormat1 provides a utility coercion for changing"
     "to TeX format anything that has a coercion to the standard output format."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TexFormat1|
  (progn
    (push '|TexFormat1| *Packages*)
    (make-instance '|TexFormat1Type|)))
```

———-

1.94.7 ToolsForSign

— defclass ToolsForSignType —

```
(defclass |ToolsForSignType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ToolsForSign")
   (marker :initform 'package)
   (abbreviation :initform 'TOOLSIGN)
   (comment :initform (list
     "Tools for the sign finding utilities."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ToolsForSign|
  (progn
    (push '|ToolsForSign| *Packages*)
    (make-instance '|ToolsForSignType|)))
```

———-

1.94.8 TopLevelDrawFunctions

— defclass TopLevelDrawFunctionsType —

```
(defclass |TopLevelDrawFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TopLevelDrawFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'DRAW)
   (comment :initform (list
     "TopLevelDrawFunctions provides top level functions for"
     "drawing graphics of expressions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctions|
  (progn
    (push '|TopLevelDrawFunctions| *Packages*)
    (make-instance '|TopLevelDrawFunctionsType|)))
```

———-

1.94.9 TopLevelDrawFunctionsForAlgebraicCurves

— defclass TopLevelDrawFunctionsForAlgebraicCurvesType —

```
(defclass |TopLevelDrawFunctionsForAlgebraicCurvesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TopLevelDrawFunctionsForAlgebraicCurves")
   (marker :initform 'package)
   (abbreviation :initform 'DRAWCURV)
   (comment :initform (list
     "TopLevelDrawFunctionsForAlgebraicCurves provides top level"
     "functions for drawing non-singular algebraic curves."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctionsForAlgebraicCurves|
  (progn
    (push '|TopLevelDrawFunctionsForAlgebraicCurves| *Packages*)
    (make-instance '|TopLevelDrawFunctionsForAlgebraicCurvesType|)))
```

———-

1.94.10 TopLevelDrawFunctionsForCompiledFunctions

— defclass TopLevelDrawFunctionsForCompiledFunctionsType —

```
(defclass |TopLevelDrawFunctionsForCompiledFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TopLevelDrawFunctionsForCompiledFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'DRAWCFUN)
   (comment :initform (list
     "TopLevelDrawFunctionsForCompiledFunctions provides top level"
     "functions for drawing graphics of expressions."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctionsForCompiledFunctions|
  (progn
    (push '|TopLevelDrawFunctionsForCompiledFunctions| *Packages*)
    (make-instance '|TopLevelDrawFunctionsForCompiledFunctionsType|)))
```

———-

1.94.11 TopLevelDrawFunctionsForPoints

— defclass TopLevelDrawFunctionsForPointsType —

```
(defclass |TopLevelDrawFunctionsForPointsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TopLevelDrawFunctionsForPoints")
   (marker :initform 'package)
   (abbreviation :initform 'DRAWPT)
   (comment :initform (list
     "TopLevelDrawFunctionsForPoints provides top level functions"
     "for drawing curves and surfaces described by sets of points."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctionsForPoints|
  (progn
    (push '|TopLevelDrawFunctionsForPoints| *Packages*)
    (make-instance '|TopLevelDrawFunctionsForPointsType|)))
```

———-

1.94.12 TopLevelThreeSpace

— defclass TopLevelThreeSpaceType —

```
(defclass |TopLevelThreeSpaceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TopLevelThreeSpace")
   (marker :initform 'package)
   (abbreviation :initform 'TOPSP)
   (comment :initform (list
     "This package exports a function for making a ThreeSpace"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelThreeSpace|
  (progn
    (push '|TopLevelThreeSpace| *Packages*)
    (make-instance '|TopLevelThreeSpaceType|)))
```

———-

1.94.13 TranscendentalHermiteIntegration

— defclass TranscendentalHermiteIntegrationType —

```
(defclass |TranscendentalHermiteIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalHermiteIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTHERTR)
   (comment :initform (list
     "Hermite integration, transcendental case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalHermiteIntegration|
  (progn
    (push '|TranscendentalHermiteIntegration| *Packages*)
    (make-instance '|TranscendentalHermiteIntegrationType|)))
```

———-

1.94.14 TranscendentalIntegration

— defclass TranscendentalIntegrationType —

```
(defclass |TranscendentalIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTTR)
   (comment :initform (list
     "This package provides functions for the transcendental"
     "case of the Risch algorithm."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalIntegration|
  (progn
    (push '|TranscendentalIntegration| *Packages*)
    (make-instance '|TranscendentalIntegrationType|)))
```

———-

1.94.15 TranscendentalManipulations

— defclass TranscendentalManipulationsType —

```
(defclass |TranscendentalManipulationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalManipulations")
   (marker :initform 'package)
   (abbreviation :initform 'TRMANIP)
   (comment :initform (list
     "TranscendentalManipulations provides functions to simplify and"
     "expand expressions involving transcendental operators."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalManipulations|
  (progn
    (push '|TranscendentalManipulations| *Packages*)
    (make-instance '|TranscendentalManipulationsType|)))
```

———-

1.94.16 TranscendentalRischDE

— defclass TranscendentalRischDEType —

```
(defclass |TranscendentalRischDEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalRischDE")
   (marker :initform 'package)
   (abbreviation :initform 'RDETR)
   (comment :initform (list
     "Risch differential equation, transcendental case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalRischDE|
  (progn
    (push '|TranscendentalRischDE| *Packages*)
    (make-instance '|TranscendentalRischDEType|)))
```

———-

1.94.17 TranscendentalRischDESystem

— defclass TranscendentalRischDESystemType —

```
(defclass |TranscendentalRischDESystemType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalRischDESystem")
   (marker :initform 'package)
   (abbreviation :initform 'RDETRS)
   (comment :initform (list
     "Risch differential equation system, transcendental case."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalRischDESystem|
  (progn
    (push '|TranscendentalRischDESystem| *Packages*)
    (make-instance '|TranscendentalRischDESystemType|)))
```

———-

1.94.18 TransSolvePackage

— defclass TransSolvePackageType —

```
(defclass |TransSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TransSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'SOLVETRA)
   (comment :initform (list
     "This package tries to find solutions of equations of type Expression(R)."
     "This means expressions involving transcendental, exponential, logarithmic"
     "and nthRoot functions."
     "After trying to transform different kernels to one kernel by applying"
     "several rules, it calls zerosOf for the SparseUnivariatePolynomial in"
     "the remaining kernel."
     "For example the expression sin(x)*cos(x)-2 will be transformed to"
     "-2 tan(x/2)**4 -2 tan(x/2)**3 -4 tan(x/2)**2 +2 tan(x/2) -2"
     "by using the function normalize and then to"
     "-2 tan(x)**2 + tan(x) -2"
     "with help of subsTan. This function tries to express the given function"
     "in terms of tan(x/2) to express in terms of tan(x)."
     "Other examples are the expressions sqrt(x+1)+sqrt(x+7)+1 or"
     "sqrt(sin(x))+1 ."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TransSolvePackage|
  (progn
    (push '|TransSolvePackage| *Packages*)
    (make-instance '|TransSolvePackageType|)))
```

———-

1.94.19 TransSolvePackageService

— defclass TransSolvePackageServiceType —

```
(defclass |TransSolvePackageServiceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TransSolvePackageService")
   (marker :initform 'package)
   (abbreviation :initform 'SOLVESER)
   (comment :initform (list
     "This package finds the function func3 where  func1 and func2"
     "are given and  func1 = func3(func2) .  If there is no solution then"
     "function func1 will be returned."
     "An example would be func1:= 8*X**3+32*X**2-14*X ::EXPR INT and"
     "func2:=2*X ::EXPR INT convert them via univariate"
     "to FRAC SUP EXPR INT and then the solution is func3:=X**3+X**2-X"
     "of type FRAC SUP EXPR INT"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TransSolvePackageService|
  (progn
    (push '|TransSolvePackageService| *Packages*)
    (make-instance '|TransSolvePackageServiceType|)))
```

———-

1.94.20 TriangularMatrixOperations

— defclass TriangularMatrixOperationsType —

```
(defclass |TriangularMatrixOperationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TriangularMatrixOperations")
   (marker :initform 'package)
   (abbreviation :initform 'TRIMAT)
   (comment :initform (list
     "This package provides functions that compute 'fraction-free'"
     "inverses of upper and lower triangular matrices over a integral"
     "domain.  By 'fraction-free inverses' we mean the following:"
     "given a matrix B with entries in R and an element d of R such that"
     "d * inv(B) also has entries in R, we return d * inv(B).  Thus,"
     "it is not necessary to pass to the quotient field in any of our"
     "computations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TriangularMatrixOperations|
  (progn
    (push '|TriangularMatrixOperations| *Packages*)
    (make-instance '|TriangularMatrixOperationsType|)))
```

———-

1.94.21 TrigonometricManipulations

— defclass TrigonometricManipulationsType —

```
(defclass |TrigonometricManipulationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TrigonometricManipulations")
   (marker :initform 'package)
   (abbreviation :initform 'TRIGMNIP)
   (comment :initform (list
     "TrigonometricManipulations provides transformations from"
     "trigonometric functions to complex exponentials and logarithms, and back."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TrigonometricManipulations|
  (progn
    (push '|TrigonometricManipulations| *Packages*)
    (make-instance '|TrigonometricManipulationsType|)))
```

———-

1.94.22 TubePlotTools

— defclass TubePlotToolsType —

```
(defclass |TubePlotToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TubePlotTools")
   (marker :initform 'package)
   (abbreviation :initform 'TUBETOOL)
   (comment :initform (list
     "Tools for constructing tubes around 3-dimensional parametric curves."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TubePlotTools|
  (progn
    (push '|TubePlotTools| *Packages*)
    (make-instance '|TubePlotToolsType|)))
```

———-

1.94.23 TwoDimensionalPlotClipping

— defclass TwoDimensionalPlotClippingType —

```
(defclass |TwoDimensionalPlotClippingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TwoDimensionalPlotClipping")
   (marker :initform 'package)
   (abbreviation :initform 'CLIP)
   (comment :initform (list
     "Automatic clipping for 2-dimensional plots"
     "The purpose of this package is to provide reasonable plots of"
```

```
    "functions with singularities."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoDimensionalPlotClipping|
  (progn
    (push '|TwoDimensionalPlotClipping| *Packages*)
    (make-instance '|TwoDimensionalPlotClippingType|)))
```

———-

1.94.24 TwoFactorize

— defclass TwoFactorizeType —

```
(defclass |TwoFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TwoFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'TWOFACT)
   (comment :initform (list
     "A basic package for the factorization of bivariate polynomials"
     "over a finite field."
     "The functions here represent the base step for the multivariate factorizer."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoFactorize|
  (progn
    (push '|TwoFactorize| *Packages*)
    (make-instance '|TwoFactorizeType|)))
```

———-

1.95 U

1.95.1 UnivariateFactorize

— defclass UnivariateFactorizeType —

```
(defclass |UnivariateFactorizeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariateFactorize")
   (marker :initform 'package)
   (abbreviation :initform 'UNIFACT)
   (comment :initform (list
     "Package for the factorization of univariate polynomials with integer"
     "coefficients. The factorization is done by 'lifting' (HENSEL) the"
     "factorization over a finite field."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateFactorize|
  (progn
    (push '|UnivariateFactorize| *Packages*)
    (make-instance '|UnivariateFactorizeType|)))
```

———-

1.95.2 UnivariateFormalPowerSeriesFunctions

— defclass UnivariateFormalPowerSeriesFunctionsType —

```
(defclass |UnivariateFormalPowerSeriesFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariateFormalPowerSeriesFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'UFPS1)
   (comment :initform (list
     "This package has no description"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateFormalPowerSeriesFunctions|
  (progn
    (push '|UnivariateFormalPowerSeriesFunctions| *Packages*)
    (make-instance '|UnivariateFormalPowerSeriesFunctionsType|)))
```

———-

1.95.3 UnivariateLaurentSeriesFunctions2

— defclass UnivariateLaurentSeriesFunctions2Type —

```
(defclass |UnivariateLaurentSeriesFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariateLaurentSeriesFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ULS2)
   (comment :initform (list
     "Mapping package for univariate Laurent series"
     "This package allows one to apply a function to the coefficients of"
     "a univariate Laurent series."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeriesFunctions2|
  (progn
    (push '|UnivariateLaurentSeriesFunctions2| *Packages*)
    (make-instance '|UnivariateLaurentSeriesFunctions2Type|)))
```

————-

1.95.4 UnivariatePolynomialCategoryFunctions2

— defclass UnivariatePolynomialCategoryFunctions2Type —

```
(defclass |UnivariatePolynomialCategoryFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePolynomialCategoryFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'UPOLYC2)
   (comment :initform (list
     "Mapping from polynomials over R to polynomials over S"
     "given a map from R to S assumed to send zero to zero."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialCategoryFunctions2|
  (progn
    (push '|UnivariatePolynomialCategoryFunctions2| *Packages*)
    (make-instance '|UnivariatePolynomialCategoryFunctions2Type|)))
```

————-

1.95.5 UnivariatePolynomialCommonDenominator

— defclass UnivariatePolynomialCommonDenominatorType —

```
(defclass |UnivariatePolynomialCommonDenominatorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePolynomialCommonDenominator")
   (marker :initform 'package)
   (abbreviation :initform 'UPCDEN)
   (comment :initform (list
     "UnivariatePolynomialCommonDenominator provides"
     "functions to compute the common denominator of the coefficients of"
```

```
    "univariate polynomials over the quotient field of a gcd domain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialCommonDenominator|
  (progn
    (push '|UnivariatePolynomialCommonDenominator| *Packages*)
    (make-instance '|UnivariatePolynomialCommonDenominatorType|)))
```

———-

1.95.6 UnivariatePolynomialDecompositionPackage

— defclass UnivariatePolynomialDecompositionPackageType —

```
(defclass |UnivariatePolynomialDecompositionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePolynomialDecompositionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'UPDECOMP)
   (comment :initform (list
     "UnivariatePolynomialDecompositionPackage implements"
     "functional decomposition of univariate polynomial with coefficients"
     "in an IntegralDomain of CharacteristicZero."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialDecompositionPackage|
  (progn
    (push '|UnivariatePolynomialDecompositionPackage| *Packages*)
    (make-instance '|UnivariatePolynomialDecompositionPackageType|)))
```

———-

1.95.7 UnivariatePolynomialDivisionPackage

— defclass UnivariatePolynomialDivisionPackageType —

```
(defclass |UnivariatePolynomialDivisionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePolynomialDivisionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'UPDIVP)
   (comment :initform (list
     "UnivariatePolynomialDivisionPackage provides a"
     "division for non monic univarite polynomials with coefficients in"
     "an IntegralDomain."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialDivisionPackage|
  (progn
    (push '|UnivariatePolynomialDivisionPackage| *Packages*)
    (make-instance '|UnivariatePolynomialDivisionPackageType|)))
```

———-

1.95.8 UnivariatePolynomialFunctions2

— defclass UnivariatePolynomialFunctions2Type —

```
(defclass |UnivariatePolynomialFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePolynomialFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'UP2)
   (comment :initform (list
     "This package lifts a mapping from coefficient rings R to S to"
     "a mapping from UnivariatePolynomial(x,R) to"
     "UnivariatePolynomial(y,S). Note that the mapping is assumed"
     "to send zero to zero, since it will only be applied to the non-zero"
     "coefficients of the polynomial."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialFunctions2|
  (progn
    (push '|UnivariatePolynomialFunctions2| *Packages*)
    (make-instance '|UnivariatePolynomialFunctions2Type|)))
```

———-

1.95.9 UnivariatePolynomialMultiplicationPackage

— defclass UnivariatePolynomialMultiplicationPackageType —

```
(defclass |UnivariatePolynomialMultiplicationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePolynomialMultiplicationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'UPMP)
   (comment :initform (list
     "This package implements Karatsuba's trick for multiplying"
     "(large) univariate polynomials. It could be improved with"
     "a version doing the work on place and also with a special"
     "case for squares. We've done this in Basicmath, but we"
     "believe that this out of the scope of AXIOM."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialMultiplicationPackage|
  (progn
    (push '|UnivariatePolynomialMultiplicationPackage| *Packages*)
    (make-instance '|UnivariatePolynomialMultiplicationPackageType|)))
```

———-

1.95.10 UnivariatePolynomialSquareFree

— defclass UnivariatePolynomialSquareFreeType —

```
(defclass |UnivariatePolynomialSquareFreeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePolynomialSquareFree")
   (marker :initform 'package)
   (abbreviation :initform 'UPSQFREE)
   (comment :initform (list
     "This package provides for square-free decomposition of"
     "univariate polynomials over arbitrary rings,"
     "a partial factorization such that each factor is a product"
     "of irreducibles with multiplicity one and the factors are"
     "pairwise relatively prime. If the ring"
     "has characteristic zero, the result is guaranteed to satisfy"
     "this condition. If the ring is an infinite ring of"
     "finite characteristic, then it may not be possible to decide when"
     "polynomials contain factors which are pth powers. In this"
     "case, the flag associated with that polynomial is set to 'nil'"
     "(meaning that that polynomials are not guaranteed to be square-free)."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialSquareFree|
  (progn
    (push '|UnivariatePolynomialSquareFree| *Packages*)
    (make-instance '|UnivariatePolynomialSquareFreeType|)))
```

———-

1.95.11 UnivariatePuiseuxSeriesFunctions2

— defclass UnivariatePuiseuxSeriesFunctions2Type —

```
(defclass |UnivariatePuiseuxSeriesFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariatePuiseuxSeriesFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'UPXS2)
   (comment :initform (list
     "Mapping package for univariate Puiseux series."
     "This package allows one to apply a function to the coefficients of"
     "a univariate Puiseux series."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesFunctions2|
  (progn
    (push '|UnivariatePuiseuxSeriesFunctions2| *Packages*)
    (make-instance '|UnivariatePuiseuxSeriesFunctions2Type|)))
```

———-

1.95.12 UnivariateSkewPolynomialCategoryOps

— defclass UnivariateSkewPolynomialCategoryOpsType —

```
(defclass |UnivariateSkewPolynomialCategoryOpsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariateSkewPolynomialCategoryOps")
   (marker :initform 'package)
   (abbreviation :initform 'OREPCTO)
   (comment :initform (list
     "UnivariateSkewPolynomialCategoryOps provides products and"
     "divisions of univariate skew polynomials."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateSkewPolynomialCategoryOps|
  (progn
    (push '|UnivariateSkewPolynomialCategoryOps| *Packages*)
    (make-instance '|UnivariateSkewPolynomialCategoryOpsType|)))
```

———-

1.95.13 UnivariateTaylorSeriesFunctions2

— defclass UnivariateTaylorSeriesFunctions2Type —

```
(defclass |UnivariateTaylorSeriesFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariateTaylorSeriesFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'UTS2)
   (comment :initform (list
     "Mapping package for univariate Taylor series."
     "This package allows one to apply a function to the coefficients of"
     "a univariate Taylor series."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesFunctions2|
  (progn
    (push '|UnivariateTaylorSeriesFunctions2| *Packages*)
    (make-instance '|UnivariateTaylorSeriesFunctions2Type|)))
```

———-

1.95.14 UnivariateTaylorSeriesODESolver

— defclass UnivariateTaylorSeriesODESolverType —

```
(defclass |UnivariateTaylorSeriesODESolverType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariateTaylorSeriesODESolver")
   (marker :initform 'package)
   (abbreviation :initform 'UTSODE)
   (comment :initform (list
     "Taylor series solutions of explicit ODE's."
     "This package provides Taylor series solutions to regular"
     "linear or non-linear ordinary differential equations of"
     "arbitrary order."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesODESolver|
  (progn
    (push '|UnivariateTaylorSeriesODESolver| *Packages*)
    (make-instance '|UnivariateTaylorSeriesODESolverType|)))
```

———-

1.95.15 UniversalSegmentFunctions2

— defclass UniversalSegmentFunctions2Type —

```
(defclass |UniversalSegmentFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UniversalSegmentFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'UNISEG2)
   (comment :initform (list
     "This package provides operations for mapping functions onto segments."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UniversalSegmentFunctions2|
  (progn
    (push '|UniversalSegmentFunctions2| *Packages*)
    (make-instance '|UniversalSegmentFunctions2Type|)))
```

———-

1.95.16 UserDefinedPartialOrdering

— defclass UserDefinedPartialOrderingType —

```
(defclass |UserDefinedPartialOrderingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UserDefinedPartialOrdering")
   (marker :initform 'package)
   (abbreviation :initform 'UDPO)
   (comment :initform (list
     "Provides functions to force a partial ordering on any set."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UserDefinedPartialOrdering|
  (progn
    (push '|UserDefinedPartialOrdering| *Packages*)
    (make-instance '|UserDefinedPartialOrderingType|)))
```

———-

1.95.17 UserDefinedVariableOrdering

— defclass UserDefinedVariableOrderingType —

```
(defclass |UserDefinedVariableOrderingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UserDefinedVariableOrdering")
   (marker :initform 'package)
   (abbreviation :initform 'UDVO)
   (comment :initform (list
     "This packages provides functions to allow the user to select the ordering"
     "on the variables and operators for displaying polynomials,"
     "fractions and expressions. The ordering affects the display"
     "only and not the computations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UserDefinedVariableOrdering|
  (progn
    (push '|UserDefinedVariableOrdering| *Packages*)
    (make-instance '|UserDefinedVariableOrderingType|)))
```

———-

1.95.18 UTSodetools

— defclass UTSodetoolsType —

```
(defclass |UTSodetoolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UTSodetools")
   (marker :initform 'package)
   (abbreviation :initform 'UTSODETL)
   (comment :initform (list
     "RUTSodetools provides tools to interface with the series"
     "ODE solver when presented with linear ODEs."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UTSodetools|
  (progn
    (push '|UTSodetools| *Packages*)
    (make-instance '|UTSodetoolsType|)))
```

———-

1.95.19 U32VectorPolynomialOperations

— defclass U32VectorPolynomialOperationsType —

```
(defclass |U32VectorPolynomialOperationsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "U32VectorPolynomialOperations")
   (marker :initform 'package)
   (abbreviation :initform 'POLYVEC)
   (comment :initform (list
     "This is a low-level package which implements operations"
     "on vectors treated as univariate modular polynomials.  Most"
     "operations takes modulus as parameter.  Modulus is machine"
     "sized prime which should be small enough to avoid overflow"
     "in intermediate calculations."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U32VectorPolynomialOperations|
  (progn
    (push '|U32VectorPolynomialOperations| *Packages*)
    (make-instance '|U32VectorPolynomialOperationsType|)))
```

———-

1.96 V

1.96.1 VectorFunctions2

— defclass VectorFunctions2Type —

```
(defclass |VectorFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "VectorFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'VECTOR2)
   (comment :initform (list
     "This package provides operations which all take as arguments"
     "vectors of elements of some type A and functions from A to"
     "another of type B. The operations all iterate over their vector argument"
     "and either return a value of type B or a vector over B."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |VectorFunctions2|
  (progn
    (push '|VectorFunctions2| *Packages*)
    (make-instance '|VectorFunctions2Type|)))
```

———-

1.96.2 ViewDefaultsPackage

— defclass ViewDefaultsPackageType —

```
(defclass |ViewDefaultsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ViewDefaultsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'VIEWDEF)
   (comment :initform (list
     "ViewportDefaultsPackage describes default and user definable"
     "values for graphics"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ViewDefaultsPackage|
  (progn
    (push '|ViewDefaultsPackage| *Packages*)
    (make-instance '|ViewDefaultsPackageType|)))
```

————-

1.96.3 ViewportPackage

— defclass ViewportPackageType —

```
(defclass |ViewportPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ViewportPackage")
   (marker :initform 'package)
   (abbreviation :initform 'VIEW)
   (comment :initform (list
     "ViewportPackage provides functions for creating GraphImages"
     "and TwoDimensionalViewports from lists of lists of points."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ViewportPackage|
  (progn
    (push '|ViewportPackage| *Packages*)
    (make-instance '|ViewportPackageType|)))
```

————-

1.97 W

1.97.1 WeierstrassPreparation

— defclass WeierstrassPreparationType —

```
(defclass |WeierstrassPreparationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "WeierstrassPreparation")
   (marker :initform 'package)
   (abbreviation :initform 'WEIER)
   (comment :initform (list
     "This package implements the Weierstrass preparation"
     "theorem f or multivariate power series."
     "weierstrass(v,p) where v is a variable, and p is a"
     "TaylorSeries(R) in which the terms"
     "of lowest degree s must include c*v**s where c is a constant,s>0,"
     "is a list of TaylorSeries coefficients A[i] of the equivalent polynomial"
     "A = A[0] + A[1]*v + A[2]*v**2 + ... + A[s-1]*v**(s-1) + v**s"
     "such that p=A*B , B being a TaylorSeries of minimum degree 0"))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |WeierstrassPreparation|
  (progn
    (push '|WeierstrassPreparation| *Packages*)
    (make-instance '|WeierstrassPreparationType|)))
```

———-

1.97.2 WildFunctionFieldIntegralBasis

— defclass WildFunctionFieldIntegralBasisType —

```
(defclass |WildFunctionFieldIntegralBasisType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "WildFunctionFieldIntegralBasis")
   (marker :initform 'package)
   (abbreviation :initform 'WFFINTBS)
   (comment :initform (list
     "In this package K is a finite field, R is a ring of univariate"
     "polynomials over K, and F is a framed algebra over R.  The package"
     "provides a function to compute the integral closure of R in the quotient"
     "field of F as well as a function to compute a 'local integral basis'"
     "at a specific prime."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |WildFunctionFieldIntegralBasis|
  (progn
    (push '|WildFunctionFieldIntegralBasis| *Packages*)
    (make-instance '|WildFunctionFieldIntegralBasisType|)))
```

———-

1.98 X

1.98.1 XExponentialPackage

— defclass XExponentialPackageType —

```
(defclass |XExponentialPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "XExponentialPackage")
   (marker :initform 'package)
   (abbreviation :initform 'XEXPPKG)
   (comment :initform (list
     "This package provides computations of logarithms and exponentials"
     "for polynomials in non-commutative variables."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XExponentialPackage|
  (progn
    (push '|XExponentialPackage| *Packages*)
    (make-instance '|XExponentialPackageType|)))
```

———-

1.99 Z

1.99.1 ZeroDimensionalSolvePackage

— defclass ZeroDimensionalSolvePackageType —

```
(defclass |ZeroDimensionalSolvePackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ZeroDimensionalSolvePackage")
   (marker :initform 'package)
   (abbreviation :initform 'ZDSOLVE)
   (comment :initform (list
     "A package for computing symbolically the complex and real roots of"
     "zero-dimensional algebraic systems over the integer or rational"
     "numbers. Complex roots are given by means of univariate representations"
     "of irreducible regular chains. Real roots are given by means of tuples"
     "of coordinates lying in the RealClosure of the coefficient ring."
     "This constructor takes three arguments. The first one R is the"
     "coefficient ring. The second one ls is the list of variables"
     "involved in the systems to solve. The third one must be concat(ls,s)"
     "where s is an additional symbol used for the univariate"
     "representations."
     "WARNING. The third argument is not checked."
     "All operations are based on triangular decompositions."
     "The default is to compute these decompositions directly from the input"
     "system by using the RegularChain domain constructor."
     "The lexTriangular algorithm can also be used for computing these"
     "decompositions (see LexTriangularPackage package constructor)."
     "For that purpose, the operations univariateSolve, realSolve and"
     "positiveSolve admit an optional argument."))
   (argslist :initform nil)
   (macros :initform nil)
   (withlist :initform (list
     (make-signature '|triangSolve| '(List |RegularChain| R ls) '(LP B B)
       (list
        "binomial(n,r) returns the \spad{(n,r)} binomial coefficient"
        "(often denoted in the literature by \spad{C(n,r)})."
        "Note that \spad{C(n,r) = n!/(r!(n-r)!)} where \spad{n >= r >= 0}.")
       (list
        "[binomial(5,i) for i in 0..5]"))
     (make-signature '|triangSolve| '(List |RegularChain| R ls) '(LP B)
       (list
         "factorial(n) computes the factorial of n"
         "(denoted in the literature by \spad{n!})"
         "Note that \spad{n! = n (n-1)! when n > 0}; also, \spad{0! = 1}."))
     (make-signature '|triangSolve| '(List |RegularChain| R ls) '(LP)
       (list
         "factorial(n) computes the factorial of n"
         "(denoted in the literature by \spad{n!})"
         "Note that \spad{n! = n (n-1)! when n > 0}; also, \spad{0! = 1}."))
     (make-signature '|permutation| '% '(% %)
       (list
         "permutation(n, m) returns the number of"
         "permutations of n objects taken m at a time."
         "Note that \spad{permutation(n,m) = n!/(n-m)!}."))))
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ZeroDimensionalSolvePackage|
  (progn
    (push '|ZeroDimensionalSolvePackage| *Packages*)
    (make-instance '|ZeroDimensionalSolvePackageType|)))
```

——————-

Support code

The **SigSort** function is called **during build** from the Makefile to sort the **signature.tex** file. The **signatures.tex** file contains lines like:

```
\hyperpage {46}\hskip 1em\relax pretty : treeform $\rightarrow $ values
```

which is the tex code to generate the signature for the function **pretty**

$$pretty : treeform \rightarrow values$$

The **SigSort** function is called to sort these lines. The sort is by name so we select out the name, using it as a hash key to remember the whole line. Finally we sort the hash keys and then walk the sorted keys to output the sorted lines.

— defun.SigSort —

```
(defun SigSort ()
  (let ((point 0) (mark 0) name (table (make-hash-table)) keys)
    (with-open-file (sigOut "signatures.sort" :direction :output
                                              :if-exists :supersede)
      (with-open-file (sigIn "signatures.tex" :direction :input)
        (loop for line = (read-line sigIn nil nil)
          while line do
             (when (setq point (search "relax" line))
               (setq point (+ 6 point))
               (setq mark (- (1- (search ":" line)) point))
               (setq name (subseq line point (+ point mark)))
               (setf (gethash (string-downcase name) table) line))))
     (maphash #'(lambda (k v)
                  (declare (ignore v))
                  (push k keys)) table)
     (setq keys (sort keys #'string<))
     (loop for key in keys do
       (format sigOut "~a~%~%" (gethash key table))))))

(progn (SigSort) (quit))
```

——————-

— all —

```
\getchunk{defvar categories}
\getchunk{defvar domains}
\getchunk{defvar forcelongs}
```

```
\getchunk{defvar indent}
\getchunk{defvar infixOperations}
\getchunk{defvar packages}
\getchunk{defvar place}
\getchunk{defvar theparse}

\getchunk{defclass hasclause}
\getchunk{defclass haslist}
\getchunk{defclass AxiomClass}
\getchunk{defclass FSM-VAR}
\getchunk{defclass WithClass}
\getchunk{defclass AddClass}
\getchunk{defclass signature}
\getchunk{defclass amacro}
\getchunk{defclass operation}
\getchunk{defclass CDPSigClass}
\getchunk{defclass ParserClass}
\getchunk{defclass gather}
\getchunk{defclass sourcecode}
\getchunk{defclass abbreviation}
\getchunk{defclass specification}
\getchunk{defclass operation}

\getchunk{defgeneric showSig}
\getchunk{defgeneric operation-add}

\getchunk{defmethod print-object}
\getchunk{defmethod operation-add}
\getchunk{defmethod showSig}

\getchunk{defmacro birth}
\getchunk{defmacro defunt}
\getchunk{defmacro depthof}
\getchunk{defmacro expect}
\getchunk{defmacro FSM-abbname}
\getchunk{defmacro FSM-AND}
\getchunk{defmacro FSM-cdpname}
\getchunk{defmacro FSM-dword}
\getchunk{defmacro FSM-gather}
\getchunk{defmacro FSM-match}
\getchunk{defmacro FSM-OR}
\getchunk{defmacro FSM-startsWith}
\getchunk{defmacro FSM-uword}
\getchunk{defmacro FSM-word}
\getchunk{defmacro ll}
\getchunk{defmacro must}
\getchunk{defmacro nameof}
\getchunk{defmacro trim}

\getchunk{defun indent}
\getchunk{defun pretty}
\getchunk{defun spawn}
\getchunk{defun splitchar}
\getchunk{defun pile2tree}
\getchunk{defun make-abbreviation}
\getchunk{defun joinsplit}

\getchunk{defun compileSpad}
\getchunk{defun explode}
```

```
\getchunk{defun FSM-abbrev}
\getchunk{defun FSM-gatherList}
\getchunk{defun FSM-gatherVars}
\getchunk{defun FSM-catBody}
\getchunk{defun FSM-cdpsig}
\getchunk{defun FSM-comment}
\getchunk{defun FSM-dpBody}
\getchunk{defun FSM-extract1Macro}
\getchunk{defun FSM-isCategory?}
\getchunk{defun FSM-macroNames}
\getchunk{defun FSM-macros}
\getchunk{defun FSM-variables}
\getchunk{defun FSMSIG}
\getchunk{defun getAncestors}
\getchunk{defun getDomains}
\getchunk{defun joinsplit}
\getchunk{defun leaves}
\getchunk{defun make-hasclause}
\getchunk{defun make-haslist}
\getchunk{defun make-operation}
\getchunk{defun make-Sourcecode}
\getchunk{defun make-signature}
\getchunk{defun operationToSignature}
\getchunk{defun operations-show}
\getchunk{defun Parse}
\getchunk{defun parseSignature}
\getchunk{defun sane}
\getchunk{defun signatureToOperation}

\getchunk{defvar level1}
\getchunk{defclass AdditiveValuationAttributeType}
\getchunk{defclass ApproximateAttributeType}
\getchunk{defclass ArbitraryExponentAttributeType}
\getchunk{defclass ArbitraryPrecisionAttributeType}
\getchunk{defclass ArcHyperbolicFunctionCategoryType}
\getchunk{defclass ArcTrigonometricFunctionCategoryType}
\getchunk{defclass AttributeRegistryType}
\getchunk{defclass BasicTypeType}
\getchunk{defclass CanonicalAttributeType}
\getchunk{defclass CanonicalClosedAttributeType}
\getchunk{defclass CanonicalUnitNormalAttributeType}
\getchunk{defclass CentralAttributeType}
\getchunk{defclass CoercibleToType}
\getchunk{defclass CombinatorialFunctionCategoryType}
\getchunk{defclass CommutativeStarAttributeType}
\getchunk{defclass ConvertibleToType}
\getchunk{defclass ElementaryFunctionCategoryType}
\getchunk{defclass EltableType}
\getchunk{defclass FiniteAggregateAttributeType}
\getchunk{defclass HyperbolicFunctionCategoryType}
\getchunk{defclass InnerEvalableType}
\getchunk{defclass JacobiIdentityAttributeType}
\getchunk{defclass LazyRepresentationAttributeType}
\getchunk{defclass LeftUnitaryAttributeType}
\getchunk{defclass ModularAlgebraicGcdOperationsType}
\getchunk{defclass MultiplicativeValuationAttributeType}
\getchunk{defclass NoZeroDivisorsAttributeType}
\getchunk{defclass NotherianAttributeType}
\getchunk{defclass NullSquareAttributeType}
```

```
\getchunk{defclass OpenMathType}
\getchunk{defclass PartialTranscendentalFunctionsType}
\getchunk{defclass PartiallyOrderedSetAttributeType}
\getchunk{defclass PrimitiveFunctionCategoryType}
\getchunk{defclass RadicalCategoryType}
\getchunk{defclass RetractableToType}
\getchunk{defclass RightUnitaryAttributeType}
\getchunk{defclass ShallowlyMutableAttributeType}
\getchunk{defclass SpecialFunctionCategoryType}
\getchunk{defclass TrigonometricFunctionCategoryType}
\getchunk{defclass TypeType}
\getchunk{defclass UnitsKnownAttributeType}

\getchunk{defvar level2}
\getchunk{defclass AggregateType}
\getchunk{defclass CombinatorialOpsCategoryType}
\getchunk{defclass EltableAggregateType}
\getchunk{defclass EvalableType}
\getchunk{defclass FortranProgramCategoryType}
\getchunk{defclass FullyRetractableToType}
\getchunk{defclass LogicType}
\getchunk{defclass PatternableType}
\getchunk{defclass PlottablePlaneCurveCategoryType}
\getchunk{defclass PlottableSpaceCurveCategoryType}
\getchunk{defclass RealConstantType}
\getchunk{defclass SegmentCategoryType}
\getchunk{defclass SetCategoryType}
\getchunk{defclass TranscendentalFunctionCategoryType}

\getchunk{defvar level3}
\getchunk{defclass AbelianSemiGroupType}
\getchunk{defclass BlowUpMethodCategoryType}
\getchunk{defclass ComparableType}
\getchunk{defclass FileCategoryType}
\getchunk{defclass FileNameCategoryType}
\getchunk{defclass FiniteType}
\getchunk{defclass FortranFunctionCategoryType}
\getchunk{defclass FortranMatrixCategoryType}
\getchunk{defclass FortranMatrixFunctionCategoryType}
\getchunk{defclass FortranVectorCategoryType}
\getchunk{defclass FortranVectorFunctionCategoryType}
\getchunk{defclass FullyEvalableOverType}
\getchunk{defclass GradedModuleType}
\getchunk{defclass HomogeneousAggregateType}
\getchunk{defclass IndexedDirectProductCategoryType}
\getchunk{defclass LiouvillianFunctionCategoryType}
\getchunk{defclass MonadType}
\getchunk{defclass NumericalIntegrationCategoryType}
\getchunk{defclass NumericalOptimizationCategoryType}
\getchunk{defclass OrderedSetType}
\getchunk{defclass OrdinaryDifferentialEquationsSolverCategoryType}
\getchunk{defclass PartialDifferentialEquationsSolverCategoryType}
\getchunk{defclass PatternMatchableType}
\getchunk{defclass RealRootCharacterizationCategoryType}
\getchunk{defclass SExpressionCategoryType}
\getchunk{defclass SegmentExpansionCategoryType}
\getchunk{defclass SemiGroupType}
\getchunk{defclass SetCategoryWithDegreeType}
\getchunk{defclass StepThroughType}
```

```
\getchunk{defclass ThreeSpaceCategoryType}

\getchunk{defvar level4}
\getchunk{defclass AbelianMonoidType}
\getchunk{defclass AffineSpaceCategoryType}
\getchunk{defclass BagAggregateType}
\getchunk{defclass CachableSetType}
\getchunk{defclass CollectionType}
\getchunk{defclass DifferentialVariableCategoryType}
\getchunk{defclass ExpressionSpaceType}
\getchunk{defclass FullyPatternMatchableType}
\getchunk{defclass GradedAlgebraType}
\getchunk{defclass IndexedAggregateType}
\getchunk{defclass InfinitlyClosePointCategoryType}
\getchunk{defclass MonadWithUnitType}
\getchunk{defclass MonoidType}
\getchunk{defclass |OrderedAbelianSemiGroupType}
\getchunk{defclass OrderedFiniteType}
\getchunk{defclass PlacesCategoryType}
\getchunk{defclass ProjectiveSpaceCategoryType}
\getchunk{defclass RecursiveAggregateType}
\getchunk{defclass TwoDimensionalArrayCategoryType}

\getchunk{defvar level5}
\getchunk{defclass BinaryRecursiveAggregateType}
\getchunk{defclass CancellationAbelianMonoidType}
\getchunk{defclass DesingTreeCategoryType}
\getchunk{defclass DoublyLinkedAggregateType}
\getchunk{defclass GroupType}
\getchunk{defclass LinearAggregateType}
\getchunk{defclass MatrixCategoryType}
\getchunk{defclass OrderedAbelianMonoidType}
\getchunk{defclass OrderedMonoidType}
\getchunk{defclass PolynomialSetCategoryType}
\getchunk{defclass PriorityQueueAggregateType}
\getchunk{defclass QueueAggregateType}
\getchunk{defclass SetAggregateType}
\getchunk{defclass StackAggregateType}
\getchunk{defclass UnaryRecursiveAggregateType}

\getchunk{defvar level6}
\getchunk{defclass AbelianGroupType}
\getchunk{defclass BinaryTreeCategoryType}
\getchunk{defclass DequeueAggregateType}
\getchunk{defclass DictionaryOperationsType}
\getchunk{defclass ExtensibleLinearAggregateType}
\getchunk{defclass FiniteLinearAggregateType}
\getchunk{defclass FreeAbelianMonoidCategoryType}
\getchunk{defclass OrderedCancellationAbelianMonoidType}
\getchunk{defclass PermutationCategoryType}
\getchunk{defclass StreamAggregateType}
\getchunk{defclass TriangularSetCategoryType}

\getchunk{defvar level7}
\getchunk{defclass DictionaryType}
\getchunk{defclass FiniteDivisorCategoryType}
\getchunk{defclass LazyStreamAggregateType}
\getchunk{defclass LeftModuleType}
\getchunk{defclass ListAggregateType}
```

```
\getchunk{defclass MultiDictionaryType}
\getchunk{defclass MultisetAggregateType}
\getchunk{defclass NonAssociativeRngType}
\getchunk{defclass OneDimensionalArrayAggregateType}
\getchunk{defclass OrderedAbelianGroupType}
\getchunk{defclass OrderedAbelianMonoidSupType}
\getchunk{defclass RegularTriangularSetCategoryType}
\getchunk{defclass RightModuleType}
\getchunk{defclass RngType}

\getchunk{defvar level8}
\getchunk{defclass BiModuleType}
\getchunk{defclass |BitAggregateType}
\getchunk{defclass FiniteSetAggregateType}
\getchunk{defclass KeyedDictionaryType}
\getchunk{defclass NonAssociativeRingType}
\getchunk{defclass NormalizedTriangularSetCategoryType}
\getchunk{defclass OrderedMultisetAggregateType}
\getchunk{defclass RingType}
\getchunk{defclass SquareFreeRegularTriangularSetCategoryType}
\getchunk{defclass StringAggregateType}
\getchunk{defclass |VectorCategoryType}

\getchunk{defvar level9}
\getchunk{defclass CharacteristicNonZeroType}
\getchunk{defclass CharacteristicZeroType}
\getchunk{defclass CommutativeRingType}
\getchunk{defclass DifferentialRingType}
\getchunk{defclass EntireRingType}
\getchunk{defclass LeftAlgebraType}
\getchunk{defclass LinearlyExplicitRingOverType}
\getchunk{defclass ModuleType}
\getchunk{defclass OrderedRingType}
\getchunk{defclass PartialDifferentialRingType}
\getchunk{defclass PointCategoryType}
\getchunk{defclass SquareFreeNormalizedTriangularSetCategoryType}
\getchunk{defclass StringCategoryType}
\getchunk{defclass TableAggregateType}

\getchunk{defvar level10}
\getchunk{defclass AlgebraType}
\getchunk{defclass AssociationListAggregateType}
\getchunk{defclass DifferentialExtensionType}
\getchunk{defclass DivisorCategoryType}
\getchunk{defclass FreeModuleCatType}
\getchunk{defclass FullyLinearlyExplicitRingOverType}
\getchunk{defclass LeftOreRingType}
\getchunk{defclass LieAlgebraType}
\getchunk{defclass NonAssociativeAlgebraType}
\getchunk{defclass RectangularMatrixCategoryType}
\getchunk{defclass VectorSpaceType}

\getchunk{defvar level11}
\getchunk{defclass DirectProductCategoryType}
\getchunk{defclass DivisionRingType}
\getchunk{defclass FiniteRankAlgebraType}
\getchunk{defclass FiniteRankNonAssociativeAlgebraType}
\getchunk{defclass FreeLieAlgebraType}
\getchunk{defclass IntegralDomainType}
```

```
\getchunk{defclass MonogenicLinearOperatorType}
\getchunk{defclass OctonionCategoryType}
\getchunk{defclass SquareMatrixCategoryType}
\getchunk{defclass UnivariateSkewPolynomialCategoryType}
\getchunk{defclass XAlgebraType}

\getchunk{defvar level12}
\getchunk{defclass AbelianMonoidRingType}
\getchunk{defclass FortranMachineTypeCategoryType}
\getchunk{defclass FramedAlgebraType}
\getchunk{defclass FramedNonAssociativeAlgebraType}
\getchunk{defclass GcdDomainType}
\getchunk{defclass LinearOrdinaryDifferentialOperatorCategoryType}
\getchunk{defclass OrderedIntegralDomainType}
\getchunk{defclass QuaternionCategoryType}
\getchunk{defclass XFreeAlgebraType}

\getchunk{defvar level13}
\getchunk{defclass FiniteAbelianMonoidRingType}
\getchunk{defclass IntervalCategoryType}
\getchunk{defclass PowerSeriesCategoryType}
\getchunk{defclass PrincipalIdealDomainType}
\getchunk{defclass UniqueFactorizationDomainType}
\getchunk{defclass XPolynomialsCatType}

\getchunk{defvar level14}
\getchunk{defclass EuclideanDomainType}
\getchunk{defclass MultivariateTaylorSeriesCategoryType}
\getchunk{defclass PolynomialFactorizationExplicitType}
\getchunk{defclass UnivariatePowerSeriesCategoryType}

\getchunk{defvar level15}
\getchunk{defclass FieldType}
\getchunk{defclass IntegerNumberSystemType}
\getchunk{defclass PAdicIntegerCategoryType}
\getchunk{defclass PolynomialCategoryType}
\getchunk{defclass UnivariateTaylorSeriesCategoryType}

\getchunk{defvar level16}
\getchunk{defclass AlgebraicallyClosedFieldType}
\getchunk{defclass DifferentialPolynomialCategoryType}
\getchunk{defclass FieldOfPrimeCharacteristicType}
\getchunk{defclass FunctionSpaceType}
\getchunk{defclass LocalPowerSeriesCategoryType}
\getchunk{defclass PseudoAlgebraicClosureOfPerfectFieldCategoryType}
\getchunk{defclass QuotientFieldCategoryType}
\getchunk{defclass RealClosedFieldType}
\getchunk{defclass RealNumberSystemType}
\getchunk{defclass RecursivePolynomialCategoryType}
\getchunk{defclass UnivariateLaurentSeriesCategoryType}
\getchunk{defclass UnivariatePolynomialCategoryType}
\getchunk{defclass UnivariatePuiseuxSeriesCategoryType}

\getchunk{defvar level17}
\getchunk{defclass AlgebraicallyClosedFunctionSpaceType}
\getchunk{defclass ExtensionFieldType}
\getchunk{defclass FiniteFieldCategoryType}
\getchunk{defclass FloatingPointSystemType}
\getchunk{defclass UnivariateLaurentSeriesConstructorCategoryType}
```

```
\getchunk{defclass UnivariatePuiseuxSeriesConstructorCategoryType}

\getchunk{defvar level18}
\getchunk{defclass FiniteAlgebraicExtensionFieldType}
\getchunk{defclass MonogenicAlgebraType}
\getchunk{defclass PseudoAlgebraicClosureOfFiniteFieldCategoryType}
\getchunk{defclass PseudoAlgebraicClosureOfRationalNumberCategoryType}

\getchunk{defvar level19}
\getchunk{defclass ComplexCategoryType}
\getchunk{defclass FunctionFieldCategoryType}
\getchunk{defclass PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategoryType}
\getchunk{defclass AffinePlaneType}
\getchunk{defclass AffinePlaneOverPseudoAlgebraicClosureOfFiniteFieldType}
\getchunk{defclass AffineSpaceType}
\getchunk{defclass AlgebraGivenByStructuralConstantsType}
\getchunk{defclass AlgebraicFunctionFieldType}
\getchunk{defclass AlgebraicNumberType}
\getchunk{defclass AnonymousFunctionType}
\getchunk{defclass AntiSymmType}
\getchunk{defclass AnyType}
\getchunk{defclass ArrayStackType}
\getchunk{defclass Asp1Type}
\getchunk{defclass Asp10Type}
\getchunk{defclass |Asp12Type}
\getchunk{defclass Asp19Type}
\getchunk{defclass Asp20Type}
\getchunk{defclass Asp24Type}
\getchunk{defclass Asp27Type}
\getchunk{defclass Asp28Type}
\getchunk{defclass Asp29Type}
\getchunk{defclass Asp30Type}
\getchunk{defclass Asp31Type}
\getchunk{defclass Asp33Type}
\getchunk{defclass Asp34Type}
\getchunk{defclass Asp35Type}
\getchunk{defclass Asp4Type}
\getchunk{defclass Asp41Type}
\getchunk{defclass Asp42Type}
\getchunk{defclass Asp49Type}
\getchunk{defclass Asp50Type}
\getchunk{defclass Asp55Type}
\getchunk{defclass Asp6Type}
\getchunk{defclass Asp7Type}
\getchunk{defclass Asp73Type}
\getchunk{defclass Asp74Type}
\getchunk{defclass Asp77Type}
\getchunk{defclass Asp78Type}
\getchunk{defclass Asp8Type}
\getchunk{defclass Asp80Type}
\getchunk{defclass Asp9Type}
\getchunk{defclass AssociatedJordanAlgebraType}
\getchunk{defclass AssociatedLieAlgebraType}
\getchunk{defclass AssociationListType}
\getchunk{defclass AttributeButtonsType}
\getchunk{defclass AutomorphismType}
\getchunk{defclass BalancedBinaryTreeType}
\getchunk{defclass BalancedPAdicIntegerType}
\getchunk{defclass BalancedPAdicRationalType}
```

```
\getchunk{defclass BasicFunctionsType}
\getchunk{defclass BasicOperatorType}
\getchunk{defclass BasicStochasticDifferentialType}
\getchunk{defclass BinaryExpansionType}
\getchunk{defclass BinaryFileType}
\getchunk{defclass BinarySearchTreeType}
\getchunk{defclass BinaryTournamentType}
\getchunk{defclass BinaryTreeType}
\getchunk{defclass BitsType}
\getchunk{defclass BlowUpWithHamburgerNoetherType}
\getchunk{defclass BlowUpWithQuadTransType}
\getchunk{defclass BooleanType}
\getchunk{defclass CardinalNumberType}
\getchunk{defclass CartesianTensorType}
\getchunk{defclass CellType}
\getchunk{defclass CharacterType}
\getchunk{defclass CharacterClassType}
\getchunk{defclass CliffordAlgebraType}
\getchunk{defclass ColorType}
\getchunk{defclass CommutatorType}
\getchunk{defclass ComplexType}
\getchunk{defclass ComplexDoubleFloatMatrixType}
\getchunk{defclass ComplexDoubleFloatVectorType}
\getchunk{defclass ContinuedFractionType}
\getchunk{defclass DatabaseType}
\getchunk{defclass DataListType}
\getchunk{defclass DecimalExpansionType}
\getchunk{defclass DenavitHartenbergMatrixType}
\getchunk{defclass DequeueType}
\getchunk{defclass DeRhamComplexType}
\getchunk{defclass DesingTreeType}
\getchunk{defclass DifferentialSparseMultivariatePolynomialType}
\getchunk{defclass DirectProductType}
\getchunk{defclass DirectProductMatrixModuleType}
\getchunk{defclass DirectProductModuleType}
\getchunk{defclass DirichletRingType}
\getchunk{defclass DistributedMultivariatePolynomialType}
\getchunk{defclass DivisorType}
\getchunk{defclass DoubleFloatType}
\getchunk{defclass DoubleFloatMatrixType}
\getchunk{defclass DoubleFloatVectorType}
\getchunk{defclass DrawOptionType}
\getchunk{defclass d01ajfAnnaTypeType}
\getchunk{defclass d01akfAnnaTypeType}
\getchunk{defclass d01alfAnnaTypeType}
\getchunk{defclass d01amfAnnaTypeType}
\getchunk{defclass d01anfAnnaTypeType}
\getchunk{defclass d01apfAnnaTypeType}
\getchunk{defclass d01aqfAnnaTypeType}
\getchunk{defclass d01asfAnnaTypeType}
\getchunk{defclass d01fcfAnnaTypeType}
\getchunk{defclass d01gbfAnnaTypeType}
\getchunk{defclass d01TransformFunctionTypeType}
\getchunk{defclass d02bbfAnnaTypeType}
\getchunk{defclass d02bhfAnnaTypeType}
\getchunk{defclass d02cjfAnnaTypeType}
\getchunk{defclass d02ejfAnnaTypeType}
\getchunk{defclass d03eefAnnaTypeType}
\getchunk{defclass d03fafAnnaTypeType}
```

```
\getchunk{defclass ElementaryFunctionsUnivariateLaurentSeriesType}
\getchunk{defclass ElementaryFunctionsUnivariatePuiseuxSeriesType}
\getchunk{defclass EquationType}
\getchunk{defclass EqTableType}
\getchunk{defclass EuclideanModularRingType}
\getchunk{defclass ExitType}
\getchunk{defclass ExponentialExpansionType}
\getchunk{defclass ExpressionType}
\getchunk{defclass ExponentialOfUnivariatePuiseuxSeriesType}
\getchunk{defclass ExtAlgBasisType}
\getchunk{defclass e04dgfAnnaTypeType}
\getchunk{defclass e04fdfAnnaTypeType}
\getchunk{defclass e04gcfAnnaTypeType}
\getchunk{defclass e04jafAnnaTypeType}
\getchunk{defclass e04mbfAnnaTypeType}
\getchunk{defclass e04nafAnnaTypeType}
\getchunk{defclass e04ucfAnnaTypeType}
\getchunk{defclass FactoredType}
\getchunk{defclass FileType}
\getchunk{defclass FileNameType}
\getchunk{defclass FiniteDivisorType}
\getchunk{defclass FiniteFieldType}
\getchunk{defclass FiniteFieldCyclicGroupType}
\getchunk{defclass FiniteFieldCyclicGroupExtensionType}
\getchunk{defclass FiniteFieldCyclicGroupExtensionByPolynomialType}
\getchunk{defclass FiniteFieldExtensionType}
\getchunk{defclass FiniteFieldExtensionByPolynomialType}
\getchunk{defclass FiniteFieldNormalBasisType}
\getchunk{defclass FiniteFieldNormalBasisExtensionType}
\getchunk{defclass FiniteFieldNormalBasisExtensionByPolynomialType}
\getchunk{defclass FlexibleArrayType}
\getchunk{defclass FloatType}
\getchunk{defclass FortranCodeType}
\getchunk{defclass FortranExpressionType}
\getchunk{defclass FortranProgramType}
\getchunk{defclass FortranScalarTypeType}
\getchunk{defclass FortranTemplateType}
\getchunk{defclass FortranTypeType}
\getchunk{defclass FourierComponentType}
\getchunk{defclass FourierSeriesType}
\getchunk{defclass FractionType}
\getchunk{defclass FractionalIdealType}
\getchunk{defclass FramedModuleType}
\getchunk{defclass FreeAbelianGroupType}
\getchunk{defclass FreeAbelianMonoidType}
\getchunk{defclass FreeGroupType}
\getchunk{defclass FreeModuleType}
\getchunk{defclass FreeModule1Type}
\getchunk{defclass FreeMonoidType}
\getchunk{defclass FreeNilpotentLieType}
\getchunk{defclass FullPartialFractionExpansionType}
\getchunk{defclass FunctionCalledType}
\getchunk{defclass GeneralDistributedMultivariatePolynomialType}
\getchunk{defclass GeneralModulePolynomialType}
\getchunk{defclass GenericNonAssociativeAlgebraType}
\getchunk{defclass GeneralPolynomialSetType}
\getchunk{defclass GeneralSparseTableType}
\getchunk{defclass GeneralTriangularSetType}
\getchunk{defclass GeneralUnivariatePowerSeriesType}
```

```
\getchunk{defclass GraphImageType}
\getchunk{defclass GuessOptionType}
\getchunk{defclass GuessOptionFunctions0Type}
\getchunk{defclass HashTableType}
\getchunk{defclass HeapType}
\getchunk{defclass HexadecimalExpansionType}
\getchunk{defclass HTMLFormatType}
\getchunk{defclass HomogeneousDirectProductType}
\getchunk{defclass HomogeneousDistributedMultivariatePolynomialType}
\getchunk{defclass HyperellipticFiniteDivisorType}
\getchunk{defclass InfClsPtType}
\getchunk{defclass IndexCardType}
\getchunk{defclass IndexedBitsType}
\getchunk{defclass IndexedDirectProductAbelianGroupType}
\getchunk{defclass IndexedDirectProductAbelianMonoidType}
\getchunk{defclass IndexedDirectProductObjectType}
\getchunk{defclass IndexedDirectProductOrderedAbelianMonoidType}
\getchunk{defclass IndexedDirectProductOrderedAbelianMonoidSupType}
\getchunk{defclass IndexedExponentsType}
\getchunk{defclass IndexedFlexibleArrayType}
\getchunk{defclass IndexedListType}
\getchunk{defclass IndexedMatrixType}
\getchunk{defclass IndexedOneDimensionalArrayType}
\getchunk{defclass IndexedStringType}
\getchunk{defclass IndexedTwoDimensionalArrayType}
\getchunk{defclass IndexedVectorType}
\getchunk{defclass InfiniteTupleType}
\getchunk{defclass InfinitlyClosePointType}
\getchunk{defclass InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteFieldType}
\getchunk{defclass InnerAlgebraicNumberType}
\getchunk{defclass InnerFiniteFieldType}
\getchunk{defclass InnerFreeAbelianMonoidType}
\getchunk{defclass InnerIndexedTwoDimensionalArrayType}
\getchunk{defclass InnerPAdicIntegerType}
\getchunk{defclass InnerPrimeFieldType}
\getchunk{defclass InnerSparseUnivariatePowerSeriesType}
\getchunk{defclass InnerTableType}
\getchunk{defclass InnerTaylorSeriesType}
\getchunk{defclass InputFormType}
\getchunk{defclass IntegerType}
\getchunk{defclass IntegerModType}
\getchunk{defclass IntegrationFunctionsTableType}
\getchunk{defclass IntegrationResultType}
\getchunk{defclass IntervalType}
\getchunk{defclass KernelType}
\getchunk{defclass KeyedAccessFileType}
\getchunk{defclass LaurentPolynomialType}
\getchunk{defclass LibraryType}
\getchunk{defclass LieExponentialsType}
\getchunk{defclass LiePolynomialType}
\getchunk{defclass LieSquareMatrixType}
\getchunk{defclass LinearOrdinaryDifferentialOperatorType}
\getchunk{defclass LinearOrdinaryDifferentialOperator1Type}
\getchunk{defclass LinearOrdinaryDifferentialOperator2Type}
\getchunk{defclass ListType}
\getchunk{defclass ListMonoidOpsType}
\getchunk{defclass ListMultiDictionaryType}
\getchunk{defclass LocalAlgebraType}
\getchunk{defclass LocalizeType}
```

```
\getchunk{defclass LyndonWordType}
\getchunk{defclass MachineIntegerType}
\getchunk{defclass MagmaType}
\getchunk{defclass MakeCachableSetType}
\getchunk{defclass MathMLFormatType}
\getchunk{defclass MatrixType}
\getchunk{defclass ModMonicType}
\getchunk{defclass ModularFieldType}
\getchunk{defclass ModularRingType}
\getchunk{defclass ModuleMonomialType}
\getchunk{defclass ModuleOperatorType}
\getchunk{defclass MoebiusTransformType}
\getchunk{defclass MonoidRingType}
\getchunk{defclass MultisetType}
\getchunk{defclass MultivariatePolynomialType}
\getchunk{defclass MyExpressionType}
\getchunk{defclass MyUnivariatePolynomialType}
\getchunk{defclass NeitherSparseOrDensePowerSeriesType}
\getchunk{defclass NewSparseMultivariatePolynomialType}
\getchunk{defclass NewSparseUnivariatePolynomialType}
\getchunk{defclass NoneType}
\getchunk{defclass NonNegativeIntegerType}
\getchunk{defclass NottinghamGroupType}
\getchunk{defclass NumericalIntegrationProblemType}
\getchunk{defclass NumericalODEProblemType}
\getchunk{defclass NumericalOptimizationProblemType}
\getchunk{defclass NumericalPDEProblemType}
\getchunk{defclass OctonionType}
\getchunk{defclass ODEIntensityFunctionsTableType}
\getchunk{defclass OneDimensionalArrayType}
\getchunk{defclass OnePointCompletionType}
\getchunk{defclass OpenMathConnectionType}
\getchunk{defclass OpenMathDeviceType}
\getchunk{defclass OpenMathEncodingType}
\getchunk{defclass OpenMathErrorType}
\getchunk{defclass OpenMathErrorKindType}
\getchunk{defclass OperatorType}
\getchunk{defclass OppositeMonogenicLinearOperatorType}
\getchunk{defclass OrderedCompletionType}
\getchunk{defclass OrderedDirectProductType}
\getchunk{defclass OrderedFreeMonoidType}
\getchunk{defclass OrderedVariableListType}
\getchunk{defclass OrderlyDifferentialPolynomialType}
\getchunk{defclass OrderlyDifferentialVariableType}
\getchunk{defclass OrdinaryDifferentialRingType}
\getchunk{defclass OrdinaryWeightedPolynomialsType}
\getchunk{defclass OrdSetIntsType}
\getchunk{defclass OutputFormType}
\getchunk{defclass PAdicIntegerType}
\getchunk{defclass PAdicRationalType}
\getchunk{defclass PAdicRationalConstructorType}
\getchunk{defclass PaletteType}
\getchunk{defclass ParametricPlaneCurveType}
\getchunk{defclass ParametricSpaceCurveType}
\getchunk{defclass ParametricSurfaceType}
\getchunk{defclass PartialFractionType}
\getchunk{defclass PartitionType}
\getchunk{defclass PatternType}
\getchunk{defclass PatternMatchListResultType}
```

```
\getchunk{defclass PatternMatchResultType}
\getchunk{defclass PendantTreeType}
\getchunk{defclass PermutationType}
\getchunk{defclass PermutationGroupType}
\getchunk{defclass PiType}
\getchunk{defclass PlaneAlgebraicCurvePlotType}
\getchunk{defclass PlacesType}
\getchunk{defclass PlacesOverPseudoAlgebraicClosureOfFiniteFieldType}
\getchunk{defclass PlcsType}
\getchunk{defclass PlotType}
\getchunk{defclass Plot3DType}
\getchunk{defclass PoincareBirkhoffWittLyndonBasisType}
\getchunk{defclass PointType}
\getchunk{defclass PolynomialType}
\getchunk{defclass PolynomialIdealsType}
\getchunk{defclass PolynomialRingType}
\getchunk{defclass PositiveIntegerType}
\getchunk{defclass PrimeFieldType}
\getchunk{defclass PrimitiveArrayType}
\getchunk{defclass ProductType}
\getchunk{defclass ProjectivePlaneType}
\getchunk{defclass ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteFieldType}
\getchunk{defclass ProjectiveSpaceType}
\getchunk{defclass PseudoAlgebraicClosureOfAlgExtOfRationalNumberType}
\getchunk{defclass PseudoAlgebraicClosureOfFiniteFieldType}
\getchunk{defclass PseudoAlgebraicClosureOfRationalNumberType}
\getchunk{defclass QuadraticFormType}
\getchunk{defclass QuasiAlgebraicSetType}
\getchunk{defclass QuaternionType}
\getchunk{defclass QueryEquationType}
\getchunk{defclass QueueType}
\getchunk{defclass RadicalFunctionFieldType}
\getchunk{defclass RadixExpansionType}
\getchunk{defclass RealClosureType}
\getchunk{defclass RectangularMatrixType}
\getchunk{defclass ReferenceType}
\getchunk{defclass RegularChainType}
\getchunk{defclass RegularTriangularSetType}
\getchunk{defclass ResidueRingType}
\getchunk{defclass ResultType}
\getchunk{defclass RewriteRuleType}
\getchunk{defclass RightOpenIntervalRootCharacterizationType}
\getchunk{defclass RomanNumeralType}
\getchunk{defclass RoutinesTableType}
\getchunk{defclass RuleCalledType}
\getchunk{defclass RulesetType}
\getchunk{defclass ScriptFormulaFormatType}
\getchunk{defclass SegmentType}
\getchunk{defclass SegmentBindingType}
\getchunk{defclass SetType}
\getchunk{defclass SetOfMIntegersInOneToNType}
\getchunk{defclass SequentialDifferentialPolynomialType}
\getchunk{defclass SequentialDifferentialVariableType}
\getchunk{defclass SExpressionType}
\getchunk{defclass SExpressionOfType}
\getchunk{defclass SimpleAlgebraicExtensionType}
\getchunk{defclass SimpleCellType}
\getchunk{defclass SimpleFortranProgramType}
\getchunk{defclass SingleIntegerType}
```

```
\getchunk{defclass SingletonAsOrderedSetType}
\getchunk{defclass SparseEchelonMatrixType}
\getchunk{defclass SparseMultivariatePolynomialType}
\getchunk{defclass SparseMultivariateTaylorSeriesType}
\getchunk{defclass SparseTableType}
\getchunk{defclass SparseUnivariateLaurentSeriesType}
\getchunk{defclass SparseUnivariatePolynomialType}
\getchunk{defclass SparseUnivariatePolynomialExpressionsType}
\getchunk{defclass SparseUnivariatePuiseuxSeriesType}
\getchunk{defclass SparseUnivariateSkewPolynomialType}
\getchunk{defclass SparseUnivariateTaylorSeriesType}
\getchunk{defclass SplitHomogeneousDirectProductType}
\getchunk{defclass SplittingNodeType}
\getchunk{defclass SplittingTreeType}
\getchunk{defclass SquareFreeRegularTriangularSetType}
\getchunk{defclass SquareMatrixType}
\getchunk{defclass StackType}
\getchunk{defclass StochasticDifferentialType}
\getchunk{defclass StreamType}
\getchunk{defclass StringType}
\getchunk{defclass StringTableType}
\getchunk{defclass SubSpaceType}
\getchunk{defclass SubSpaceComponentPropertyType}
\getchunk{defclass SuchThatType}
\getchunk{defclass SwitchType}
\getchunk{defclass SymbolType}
\getchunk{defclass SymbolTableType}
\getchunk{defclass SymmetricPolynomialType}
\getchunk{defclass TableType}
\getchunk{defclass TableauType}
\getchunk{defclass TaylorSeriesoType}
\getchunk{defclass TexFormatType}
\getchunk{defclass TextFileType}
\getchunk{defclass TheSymbolTableType}
\getchunk{defclass ThreeDimensionalMatrixType}
\getchunk{defclass ThreeDimensionalViewportType}
\getchunk{defclass ThreeSpaceType}
\getchunk{defclass TreeType}
\getchunk{defclass TubePlotType}
\getchunk{defclass TupleType}
\getchunk{defclass TwoDimensionalArrayType}
\getchunk{defclass TwoDimensionalViewportType}
\getchunk{defclass UnivariateFormalPowerSeriesType}
\getchunk{defclass UnivariateLaurentSeriesType}
\getchunk{defclass UnivariateLaurentSeriesConstructorType}
\getchunk{defclass UnivariatePolynomialType}
\getchunk{defclass UnivariatePuiseuxSeriesType}
\getchunk{defclass UnivariatePuiseuxSeriesConstructorType}
\getchunk{defclass UnivariatePuiseuxSeriesWithExponentialSingularityType}
\getchunk{defclass UnivariateSkewPolynomialType}
\getchunk{defclass UnivariateTaylorSeriesType}
\getchunk{defclass UnivariateTaylorSeriesCZeroType}
\getchunk{defclass UniversalSegmentType}
\getchunk{defclass U8MatrixType}
\getchunk{defclass U16MatrixType}
\getchunk{defclass U32MatrixType}
\getchunk{defclass U8VectorType}
\getchunk{defclass U16VectorType}
\getchunk{defclass U32VectorType}
```

```
\getchunk{defclass VariableType}
\getchunk{defclass VectorType}
\getchunk{defclass VoidType}
\getchunk{defclass WeightedPolynomialsType}
\getchunk{defclass WuWenTsunTriangularSetType}
\getchunk{defclass XDistributedPolynomialType}
\getchunk{defclass XPBWPolynomialType}
\getchunk{defclass XPolynomialType}
\getchunk{defclass XPolynomialRingType}
\getchunk{defclass XRecursivePolynomialType}
\getchunk{defclass AffineAlgebraicSetComputeWithGroebnerBasisType}
\getchunk{defclass AffineAlgebraicSetComputeWithResultantType}
\getchunk{defclass AlgebraicFunctionType}
\getchunk{defclass AlgebraicHermiteIntegrationType}
\getchunk{defclass AlgebraicIntegrateType}
\getchunk{defclass AlgebraicIntegrationType}
\getchunk{defclass AlgebraicManipulationsType}
\getchunk{defclass AlgebraicMultFactType}
\getchunk{defclass AlgebraPackageType}
\getchunk{defclass AlgFactorType}
\getchunk{defclass AnnaNumericalIntegrationPackageType}
\getchunk{defclass AnnaNumericalOptimizationPackageType}
\getchunk{defclass AnnaOrdinaryDifferentialEquationPackageType}
\getchunk{defclass AnnaPartialDifferentialEquationPackageType}
\getchunk{defclass AnyFunctions1Type}
\getchunk{defclass ApplicationProgramInterfaceType}
\getchunk{defclass ApplyRulesType}
\getchunk{defclass ApplyUnivariateSkewPolynomialType}
\getchunk{defclass AssociatedEquationsType}
\getchunk{defclass AttachPredicatesType}
\getchunk{defclass AxiomServerType}
\getchunk{defclass BalancedFactorisationType}
\getchunk{defclass BasicOperatorFunctions1Type}
\getchunk{defclass BezierType}
\getchunk{defclass BezoutMatrixType}
\getchunk{defclass BlowUpPackageType}
\getchunk{defclass BoundIntegerRootsType}
\getchunk{defclass BrillhartTestsType}
\getchunk{defclass CartesianTensorFunctions2Type}
\getchunk{defclass ChangeOfVariableType}
\getchunk{defclass CharacteristicPolynomialInMonogenicalAlgebraType}
\getchunk{defclass CharacteristicPolynomialPackageType}
\getchunk{defclass ChineseRemainderToolsForIntegralBasesType}
\getchunk{defclass CoerceVectorMatrixPackageType}
\getchunk{defclass CombinatorialFunctionType}
\getchunk{defclass CommonDenominatorType}
\getchunk{defclass CommonOperatorsType}
\getchunk{defclass CommuteUnivariatePolynomialCategoryType}
\getchunk{defclass ComplexFactorizationType}
\getchunk{defclass ComplexFunctions2Type}
\getchunk{defclass ComplexIntegerSolveLinearPolynomialEquationType}
\getchunk{defclass ComplexPatternType}
\getchunk{defclass ComplexPatternMatchType}
\getchunk{defclass ComplexRootFindingPackageType}
\getchunk{defclass ComplexRootPackageType}
\getchunk{defclass ComplexTrigonometricManipulationsType}
\getchunk{defclass ConstantLODEType}
\getchunk{defclass CoordinateSystemsType}
\getchunk{defclass CRApackageType}
```

```
\getchunk{defclass CycleIndicatorsType}
\getchunk{defclass CyclicStreamToolsType}
\getchunk{defclass CyclotomicPolynomialPackageType}
\getchunk{defclass CylindricalAlgebraicDecompositionPackageType}
\getchunk{defclass CylindricalAlgebraicDecompositionUtilitiesType}
\getchunk{defclass DefiniteIntegrationToolsType}
\getchunk{defclass DegreeReductionPackageType}
\getchunk{defclass DesingTreePackageType}
\getchunk{defclass DiophantineSolutionPackageType}
\getchunk{defclass DirectProductFunctions2Type}
\getchunk{defclass DiscreteLogarithmPackageType}
\getchunk{defclass DisplayPackageType}
\getchunk{defclass DistinctDegreeFactorizeType}
\getchunk{defclass DoubleFloatSpecialFunctionsType}
\getchunk{defclass DoubleResultantPackageType}
\getchunk{defclass DrawComplexType}
\getchunk{defclass DrawNumericHackType}
\getchunk{defclass DrawOptionFunctions0Type}
\getchunk{defclass DrawOptionFunctions1Type}
\getchunk{defclass d01AgentsPackageType}
\getchunk{defclass d01WeightsPackageType}
\getchunk{defclass d02AgentsPackageType}
\getchunk{defclass d03AgentsPackageType}
\getchunk{defclass EigenPackageType}
\getchunk{defclass ElementaryFunctionType}
\getchunk{defclass ElementaryFunctionDefiniteIntegrationType}
\getchunk{defclass ElementaryFunctionLODESolverType}
\getchunk{defclass ElementaryFunctionODESolverType}
\getchunk{defclass ElementaryFunctionSignType}
\getchunk{defclass ElementaryFunctionStructurePackageType}
\getchunk{defclass ElementaryIntegrationType}
\getchunk{defclass ElementaryRischDEType}
\getchunk{defclass ElementaryRischDESystemType}
\getchunk{defclass EllipticFunctionsUnivariateTaylorSeriesType}
\getchunk{defclass EquationFunctions2Type}
\getchunk{defclass ErrorFunctionsType}
\getchunk{defclass EuclideanGroebnerBasisPackageType}
\getchunk{defclass EvaluateCycleIndicatorsType}
\getchunk{defclass ExpertSystemContinuityPackageType}
\getchunk{defclass ExpertSystemContinuityPackage1Type}
\getchunk{defclass ExpertSystemToolsPackageType}
\getchunk{defclass ExpertSystemToolsPackage1Type}
\getchunk{defclass ExpertSystemToolsPackage2Type}
\getchunk{defclass ExpressionFunctions2Type}
\getchunk{defclass ExpressionSolveType}
\getchunk{defclass ExpressionSpaceFunctions1Type}
\getchunk{defclass ExpressionSpaceFunctions2Type}
\getchunk{defclass ExpressionSpaceODESolverType}
\getchunk{defclass ExpressionToOpenMathType}
\getchunk{defclass ExpressionToUnivariatePowerSeriesType}
\getchunk{defclass ExpressionTubePlotType}
\getchunk{defclass Export3DType}
\getchunk{defclass e04AgentsPackageType}
\getchunk{defclass FactoredFunctionsType}
\getchunk{defclass FactoredFunctions2Type}
\getchunk{defclass FactoredFunctionUtilitiesType}
\getchunk{defclass FactoringUtilitiesType}
\getchunk{defclass FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumberType}
\getchunk{defclass FactorisationOverPseudoAlgebraicClosureOfRationalNumberType}
```

```
\getchunk{defclass FGLMIfCanPackageType}
\getchunk{defclass FindOrderFiniteType}
\getchunk{defclass FiniteAbelianMonoidRingFunctions2Type}
\getchunk{defclass FiniteDivisorFunctions2Type}
\getchunk{defclass FiniteFieldFactorizationType}
\getchunk{defclass FiniteFieldFactorizationWithSizeParseBySideEffectType}
\getchunk{defclass FiniteFieldFunctionsType}
\getchunk{defclass FiniteFieldHomomorphismsType}
\getchunk{defclass FiniteFieldPolynomialPackageType}
\getchunk{defclass FiniteFieldPolynomialPackage2Type}
\getchunk{defclass FiniteFieldSolveLinearPolynomialEquationType}
\getchunk{defclass FiniteFieldSquareFreeDecompositionType}
\getchunk{defclass FiniteLinearAggregateFunctions2Type}
\getchunk{defclass FiniteLinearAggregateSortType}
\getchunk{defclass FiniteSetAggregateFunctions2Type}
\getchunk{defclass FloatingComplexPackageType}
\getchunk{defclass FloatingRealPackageType}
\getchunk{defclass FloatSpecialFunctionsType}
\getchunk{defclass FortranCodePackage1Type}
\getchunk{defclass FortranOutputStackPackageType}
\getchunk{defclass FortranPackageType}
\getchunk{defclass FractionalIdealFunctions2Type}
\getchunk{defclass FractionFreeFastGaussianType}
\getchunk{defclass FractionFreeFastGaussianFractionsType}
\getchunk{defclass FractionFunctions2Type}
\getchunk{defclass FramedNonAssociativeAlgebraFunctions2Type}
\getchunk{defclass FunctionalSpecialFunctionType}
\getchunk{defclass FunctionFieldCategoryFunctions2Type}
\getchunk{defclass FunctionFieldIntegralBasisType}
\getchunk{defclass FunctionSpaceAssertionsType}
\getchunk{defclass FunctionSpaceAttachPredicatesType}
\getchunk{defclass FunctionSpaceComplexIntegrationType}
\getchunk{defclass FunctionSpaceFunctions2Type}
\getchunk{defclass FunctionSpaceIntegrationType}
\getchunk{defclass FunctionSpacePrimitiveElementType}
\getchunk{defclass FunctionSpaceReduceType}
\getchunk{defclass FunctionSpaceSumType}
\getchunk{defclass FunctionSpaceToExponentialExpansionType}
\getchunk{defclass FunctionSpaceToUnivariatePowerSeriesType}
\getchunk{defclass FunctionSpaceUnivariatePolynomialFactorType}
\getchunk{defclass GaloisGroupFactorizationUtilitiesType}
\getchunk{defclass GaloisGroupFactorizerType}
\getchunk{defclass GaloisGroupPolynomialUtilitiesType}
\getchunk{defclass GaloisGroupUtilitiesType}
\getchunk{defclass GaussianFactorizationPackageType}
\getchunk{defclass GeneralHenselPackageType}
\getchunk{defclass GeneralizedMultivariateFactorizeType}
\getchunk{defclass GeneralPackageForAlgebraicFunctionFieldType}
\getchunk{defclass GeneralPolynomialGcdPackageType}
\getchunk{defclass GenerateUnivariatePowerSeriesType}
\getchunk{defclass GenExEuclidType}
\getchunk{defclass GenUFactorizeType}
\getchunk{defclass GenusZeroIntegrationType}
\getchunk{defclass GnuDrawType}
\getchunk{defclass GosperSummationMethodType}
\getchunk{defclass GraphicsDefaultsType}
\getchunk{defclass GraphvizType}
\getchunk{defclass GrayCodeType}
\getchunk{defclass GroebnerFactorizationPackageType}
```

```
\getchunk{defclass GroebnerInternalPackageType}
\getchunk{defclass GroebnerPackageType}
\getchunk{defclass GroebnerSolveType}
\getchunk{defclass GuessType}
\getchunk{defclass GuessAlgebraicNumberType}
\getchunk{defclass GuessFiniteType}
\getchunk{defclass GuessFiniteFunctionsType}
\getchunk{defclass GuessIntegerType}
\getchunk{defclass GuessPolynomialType}
\getchunk{defclass GuessUnivariatePolynomialType}
\getchunk{defclass HallBasisType}
\getchunk{defclass HeuGcdType}
\getchunk{defclass IdealDecompositionPackageType}
\getchunk{defclass IncrementingMapsType}
\getchunk{defclass InfiniteProductCharacteristicZeroType}
\getchunk{defclass InfiniteProductFiniteFieldType}
\getchunk{defclass InfiniteProductPrimeFieldType}
\getchunk{defclass InfiniteTupleFunctions2Type}
\getchunk{defclass InfiniteTupleFunctions3Type}
\getchunk{defclass InfinityType}
\getchunk{defclass InnerAlgFactorType}
\getchunk{defclass InnerCommonDenominatorType}
\getchunk{defclass InnerMatrixLinearAlgebraFunctionsType}
\getchunk{defclass InnerMatrixQuotientFieldFunctionsType}
\getchunk{defclass InnerModularGcdType}
\getchunk{defclass InnerMultFactType}
\getchunk{defclass InnerNormalBasisFieldFunctionsType}
\getchunk{defclass InnerNumericEigenPackageType}
\getchunk{defclass InnerNumericFloatSolvePackageType}
\getchunk{defclass InnerPolySignType}
\getchunk{defclass InnerPolySumType}
\getchunk{defclass InnerTrigonometricManipulationsType}
\getchunk{defclass InputFormFunctions1Type}
\getchunk{defclass InterfaceGroebnerPackageType}
\getchunk{defclass IntegerBitsType}
\getchunk{defclass IntegerCombinatoricFunctionsType}
\getchunk{defclass IntegerFactorizationPackageType}
\getchunk{defclass IntegerLinearDependenceType}
\getchunk{defclass IntegerNumberTheoryFunctionsType}
\getchunk{defclass IntegerPrimesPackageType}
\getchunk{defclass IntegerRetractionsType}
\getchunk{defclass IntegerRootsType}
\getchunk{defclass IntegerSolveLinearPolynomialEquationType}
\getchunk{defclass IntegralBasisToolsType}
\getchunk{defclass IntegralBasisPolynomialToolsType}
\getchunk{defclass IntegrationResultFunctions2Type}
\getchunk{defclass IntegrationResultRFToFunctionType}
\getchunk{defclass IntegrationResultToFunctionType}
\getchunk{defclass IntegrationToolsType}
\getchunk{defclass InternalPrintPackageType}
\getchunk{defclass InternalRationalUnivariateRepresentationPackageType}
\getchunk{defclass InterpolateFormsPackageType}
\getchunk{defclass IntersectionDivisorPackageType}
\getchunk{defclass IrredPolyOverFiniteFieldType}
\getchunk{defclass IrrRepSymNatPackageType}
\getchunk{defclass InverseLaplaceTransformType}
\getchunk{defclass KernelFunctions2Type}
\getchunk{defclass KovacicType}
\getchunk{defclass LaplaceTransformType}
```

```
\getchunk{defclass LazardSetSolvingPackageType}
\getchunk{defclass LeadingCoefDeterminationType}
\getchunk{defclass LexTriangularPackageType}
\getchunk{defclass LinearDependenceType}
\getchunk{defclass LinearOrdinaryDifferentialOperatorFactorizerType}
\getchunk{defclass LinearOrdinaryDifferentialOperatorsOpsType}
\getchunk{defclass LinearPolynomialEquationByFractionsType}
\getchunk{defclass LinearSystemFromPowerSeriesPackageType}
\getchunk{defclass LinearSystemMatrixPackageType}
\getchunk{defclass LinearSystemMatrixPackage1Type}
\getchunk{defclass LinearSystemPolynomialPackageType}
\getchunk{defclass LinGroebnerPackageType}
\getchunk{defclass LinesOpPackType}
\getchunk{defclass LiouvillianFunctionType}
\getchunk{defclass ListFunctions2Type}
\getchunk{defclass ListFunctions3Type}
\getchunk{defclass ListToMapType}
\getchunk{defclass LocalParametrizationOfSimplePointPackageType}
\getchunk{defclass MakeBinaryCompiledFunctionType}
\getchunk{defclass MakeFloatCompiledFunctionType}
\getchunk{defclass MakeFunctionType}
\getchunk{defclass MakeRecordType}
\getchunk{defclass MakeUnaryCompiledFunctionType}
\getchunk{defclass MappingPackageInternalHacks1Type}
\getchunk{defclass MappingPackageInternalHacks2Type}
\getchunk{defclass MappingPackageInternalHacks3Type}
\getchunk{defclass MappingPackage1Type}
\getchunk{defclass MappingPackage2Type}
\getchunk{defclass MappingPackage3Type}
\getchunk{defclass MappingPackage4Type}
\getchunk{defclass MatrixCategoryFunctions2Type}
\getchunk{defclass MatrixCommonDenominatorType}
\getchunk{defclass MatrixLinearAlgebraFunctionsType}
\getchunk{defclass MatrixManipulationType}
\getchunk{defclass MergeThingType}
\getchunk{defclass MeshCreationRoutinesForThreeDimensionsType}
\getchunk{defclass ModularDistinctDegreeFactorizerType}
\getchunk{defclass ModularHermitianRowReductionType}
\getchunk{defclass MonoidRingFunctions2Type}
\getchunk{defclass MonomialExtensionToolsType}
\getchunk{defclass MoreSystemCommandsType}
\getchunk{defclass MPolyCatPolyFactorizerType}
\getchunk{defclass MPolyCatRationalFunctionFactorizerType}
\getchunk{defclass MPolyCatFunctions2Type}
\getchunk{defclass MPolyCatFunctions3Type}
\getchunk{defclass MRationalFactorizeType}
\getchunk{defclass MultFiniteFactorizeType}
\getchunk{defclass MultipleMapType}
\getchunk{defclass MultiVariableCalculusFunctionsType}
\getchunk{defclass MultivariateFactorizeType}
\getchunk{defclass MultivariateLiftingType}
\getchunk{defclass MultivariateSquareFreeType}
\getchunk{defclass NagEigenPackageType}
\getchunk{defclass NagFittingPackageType}
\getchunk{defclass NagLinearEquationSolvingPackageType}
\getchunk{defclass NAGLinkSupportPackageType}
\getchunk{defclass NagIntegrationPackageType}
\getchunk{defclass NagInterpolationPackageType}
\getchunk{defclass NagLapackType}
```

```
\getchunk{defclass NagMatrixOperationsPackageType}
\getchunk{defclass NagOptimisationPackageType}
\getchunk{defclass NagOrdinaryDifferentialEquationsPackageType}
\getchunk{defclass NagPartialDifferentialEquationsPackageType}
\getchunk{defclass NagPolynomialRootsPackageType}
\getchunk{defclass NagRootFindingPackageType}
\getchunk{defclass NagSeriesSummationPackageType}
\getchunk{defclass NagSpecialFunctionsPackageType}
\getchunk{defclass NewSparseUnivariatePolynomialFunctions2Type}
\getchunk{defclass NewtonInterpolationType}
\getchunk{defclass NewtonPolygonType}
\getchunk{defclass NonCommutativeOperatorDivisionType}
\getchunk{defclass NoneFunctions1Type}
\getchunk{defclass NonLinearFirstOrderODESolverType}
\getchunk{defclass NonLinearSolvePackageType}
\getchunk{defclass NormalizationPackageType}
\getchunk{defclass NormInMonogenicAlgebraType}
\getchunk{defclass NormRetractPackageType}
\getchunk{defclass NPCoefType}
\getchunk{defclass NumberFieldIntegralBasisType}
\getchunk{defclass NumberFormatsType}
\getchunk{defclass NumberTheoreticPolynomialFunctionsType}
\getchunk{defclass NumericType}
\getchunk{defclass NumericalOrdinaryDifferentialEquationsType}
\getchunk{defclass NumericalQuadratureType}
\getchunk{defclass NumericComplexEigenPackageType}
\getchunk{defclass NumericContinuedFractionType}
\getchunk{defclass NumericRealEigenPackageType}
\getchunk{defclass NumericTubePlotType}
\getchunk{defclass OctonionCategoryFunctions2Type}
\getchunk{defclass ODEIntegrationType}
\getchunk{defclass ODEToolsType}
\getchunk{defclass OneDimensionalArrayFunctions2Type}
\getchunk{defclass OnePointCompletionFunctions2Type}
\getchunk{defclass OpenMathPackageType}
\getchunk{defclass OpenMathServerPackageType}
\getchunk{defclass OperationsQueryType}
\getchunk{defclass OrderedCompletionFunctions2Type}
\getchunk{defclass OrderingFunctionsType}
\getchunk{defclass OrthogonalPolynomialFunctionsType}
\getchunk{defclass OutputPackageType}
\getchunk{defclass PackageForAlgebraicFunctionFieldType}
\getchunk{defclass PackageForAlgebraicFunctionFieldOverFiniteFieldType}
\getchunk{defclass PackageForPolyType}
\getchunk{defclass PadeApproximantPackageType}
\getchunk{defclass PadeApproximantsType}
\getchunk{defclass PAdicWildFunctionFieldIntegralBasisType}
\getchunk{defclass ParadoxicalCombinatorsForStreamsType}
\getchunk{defclass ParametricLinearEquationsType}
\getchunk{defclass ParametricPlaneCurveFunctions2Type}
\getchunk{defclass ParametricSpaceCurveFunctions2Type}
\getchunk{defclass ParametricSurfaceFunctions2Type}
\getchunk{defclass ParametrizationPackageType}
\getchunk{defclass PartialFractionPackageType}
\getchunk{defclass PartitionsAndPermutationsType}
\getchunk{defclass PatternFunctions1Type}
\getchunk{defclass PatternFunctions2Type}
\getchunk{defclass PatternMatchType}
\getchunk{defclass PatternMatchAssertionsType}
```

```
\getchunk{defclass PatternMatchFunctionSpaceType}
\getchunk{defclass PatternMatchIntegerNumberSystemType}
\getchunk{defclass PatternMatchIntegrationType}
\getchunk{defclass PatternMatchKernelType}
\getchunk{defclass PatternMatchListAggregateType}
\getchunk{defclass PatternMatchPolynomialCategoryType}
\getchunk{defclass PatternMatchPushDownType}
\getchunk{defclass PatternMatchQuotientFieldCategoryType}
\getchunk{defclass PatternMatchResultFunctions2Type}
\getchunk{defclass PatternMatchSymbolType}
\getchunk{defclass PatternMatchToolsType}
\getchunk{defclass PermanentType}
\getchunk{defclass PermutationGroupExamplesType}
\getchunk{defclass PiCoercionsType}
\getchunk{defclass PlotFunctions1Type}
\getchunk{defclass PlotToolsType}
\getchunk{defclass ProjectiveAlgebraicSetPackageType}
\getchunk{defclass PointFunctions2Type}
\getchunk{defclass PointPackageType}
\getchunk{defclass PointsOfFiniteOrderType}
\getchunk{defclass PointsOfFiniteOrderRationalType}
\getchunk{defclass PointsOfFiniteOrderToolsType}
\getchunk{defclass PolynomialPackageForCurveType}
\getchunk{defclass PolToPolType}
\getchunk{defclass PolyGroebnerType}
\getchunk{defclass PolynomialAN2ExpressionType}
\getchunk{defclass PolynomialCategoryLiftingType}
\getchunk{defclass PolynomialCategoryQuotientFunctionsType}
\getchunk{defclass PolynomialCompositionType}
\getchunk{defclass PolynomialDecompositionType}
\getchunk{defclass PolynomialFactorizationByRecursionType}
\getchunk{defclass PolynomialFactorizationByRecursionUnivariateType}
\getchunk{defclass PolynomialFunctions2Type}
\getchunk{defclass PolynomialGcdPackageType}
\getchunk{defclass PolynomialInterpolationType}
\getchunk{defclass PolynomialInterpolationAlgorithmsType}
\getchunk{defclass PolynomialNumberTheoryFunctionsType}
\getchunk{defclass PolynomialRootsType}
\getchunk{defclass PolynomialSetUtilitiesPackageType}
\getchunk{defclass PolynomialSolveByFormulasType}
\getchunk{defclass PolynomialSquareFreeType}
\getchunk{defclass PolynomialToUnivariatePolynomialType}
\getchunk{defclass PowerSeriesLimitPackageType}
\getchunk{defclass PrecomputedAssociatedEquationsType}
\getchunk{defclass PrimitiveArrayFunctions2Type}
\getchunk{defclass PrimitiveElementType}
\getchunk{defclass PrimitiveRatDEType}
\getchunk{defclass PrimitiveRatRicDEType}
\getchunk{defclass PrintPackageType}
\getchunk{defclass PseudoLinearNormalFormType}
\getchunk{defclass PseudoRemainderSequenceType}
\getchunk{defclass PureAlgebraicIntegrationType}
\getchunk{defclass PureAlgebraicLODEType}
\getchunk{defclass PushVariablesType}
\getchunk{defclass QuasiAlgebraicSet2Type}
\getchunk{defclass QuasiComponentPackageType}
\getchunk{defclass QuotientFieldCategoryFunctions2Type}
\getchunk{defclass QuaternionCategoryFunctions2Type}
\getchunk{defclass RadicalEigenPackageType}
```

```
\getchunk{defclass RadicalSolvePackageType}
\getchunk{defclass RadixUtilitiesType}
\getchunk{defclass RandomDistributionsType}
\getchunk{defclass RandomFloatDistributionsType}
\getchunk{defclass RandomIntegerDistributionsType}
\getchunk{defclass RandomNumberSourceType}
\getchunk{defclass RationalFactorizeType}
\getchunk{defclass RationalFunctionType}
\getchunk{defclass RationalFunctionDefiniteIntegrationType}
\getchunk{defclass RationalFunctionFactorType}
\getchunk{defclass RationalFunctionFactorizerType}
\getchunk{defclass RationalFunctionIntegrationType}
\getchunk{defclass RationalFunctionLimitPackageType}
\getchunk{defclass RationalFunctionSignType}
\getchunk{defclass RationalFunctionSumType}
\getchunk{defclass RationalIntegrationType}
\getchunk{defclass RationalInterpolationType}
\getchunk{defclass RationalLODEType}
\getchunk{defclass RationalRetractionsType}
\getchunk{defclass RationalRicDEType}
\getchunk{defclass RationalUnivariateRepresentationPackageType}
\getchunk{defclass RealPolynomialUtilitiesPackageType}
\getchunk{defclass RealSolvePackageType}
\getchunk{defclass RealZeroPackageType}
\getchunk{defclass RealZeroPackageQType}
\getchunk{defclass RectangularMatrixCategoryFunctions2Type}
\getchunk{defclass RecurrenceOperatorType}
\getchunk{defclass ReducedDivisorType}
\getchunk{defclass ReduceLODEType}
\getchunk{defclass ReductionOfOrderType}
\getchunk{defclass RegularSetDecompositionPackageType}
\getchunk{defclass RegularTriangularSetGcdPackageType}
\getchunk{defclass RepeatedDoublingType}
\getchunk{defclass RepeatedSquaringType}
\getchunk{defclass RepresentationPackage1Type}
\getchunk{defclass RepresentationPackage2Type}
\getchunk{defclass ResolveLatticeCompletionType}
\getchunk{defclass RetractSolvePackageType}
\getchunk{defclass RootsFindingPackageType}
\getchunk{defclass SAERationalFunctionAlgFactorType}
\getchunk{defclass ScriptFormulaFormat1Type}
\getchunk{defclass SegmentBindingFunctions2Type}
\getchunk{defclass SegmentFunctions2Type}
\getchunk{defclass SimpleAlgebraicExtensionAlgFactorType}
\getchunk{defclass SimplifyAlgebraicNumberConvertPackageType}
\getchunk{defclass SmithNormalFormType}
\getchunk{defclass SortedCacheType}
\getchunk{defclass SortPackageType}
\getchunk{defclass SparseUnivariatePolynomialFunctions2Type}
\getchunk{defclass SpecialOutputPackageType}
\getchunk{defclass SquareFreeQuasiComponentPackageType}
\getchunk{defclass SquareFreeRegularSetDecompositionPackageType}
\getchunk{defclass SquareFreeRegularTriangularSetGcdPackageType}
\getchunk{defclass StorageEfficientMatrixOperationsType}
\getchunk{defclass StreamFunctions1Type}
\getchunk{defclass StreamFunctions2Type}
\getchunk{defclass StreamFunctions3Type}
\getchunk{defclass StreamInfiniteProductType}
\getchunk{defclass StreamTaylorSeriesOperationsType}
```

```
\getchunk{defclass StreamTensorType}
\getchunk{defclass StreamTranscendentalFunctionsType}
\getchunk{defclass StreamTranscendentalFunctionsNonCommutativeType}
\getchunk{defclass StructuralConstantsPackageType}
\getchunk{defclass SturmHabichtPackageType}
\getchunk{defclass SubResultantPackageType}
\getchunk{defclass SupFractionFactorizerType}
\getchunk{defclass SystemODESolverType}
\getchunk{defclass SystemSolvePackageType}
\getchunk{defclass SymmetricGroupCombinatoricFunctionsType}
\getchunk{defclass SymmetricFunctionsType}
\getchunk{defclass TableauxBumpersType}
\getchunk{defclass TabulatedComputationPackageType}
\getchunk{defclass TangentExpansionsType}
\getchunk{defclass TaylorSolveType}
\getchunk{defclass TemplateUtilitiesType}
\getchunk{defclass TexFormat1Type}
\getchunk{defclass ToolsForSignType}
\getchunk{defclass TopLevelDrawFunctionsType}
\getchunk{defclass TopLevelDrawFunctionsForAlgebraicCurvesType}
\getchunk{defclass TopLevelDrawFunctionsForCompiledFunctionsType}
\getchunk{defclass TopLevelDrawFunctionsForPointsType}
\getchunk{defclass TopLevelThreeSpaceType}
\getchunk{defclass TranscendentalHermiteIntegrationType}
\getchunk{defclass TranscendentalIntegrationType}
\getchunk{defclass TranscendentalManipulationsType}
\getchunk{defclass TranscendentalRischDEType}
\getchunk{defclass TranscendentalRischDESystemType}
\getchunk{defclass TransSolvePackageType}
\getchunk{defclass TransSolvePackageServiceType}
\getchunk{defclass TriangularMatrixOperationsType}
\getchunk{defclass TrigonometricManipulationsType}
\getchunk{defclass TubePlotToolsType}
\getchunk{defclass TwoDimensionalPlotClippingType}
\getchunk{defclass TwoFactorizeType}
\getchunk{defclass UnivariateFactorizeType}
\getchunk{defclass UnivariateFormalPowerSeriesFunctionsType}
\getchunk{defclass UnivariateLaurentSeriesFunctions2Type}
\getchunk{defclass UnivariatePolynomialCategoryFunctions2Type}
\getchunk{defclass UnivariatePolynomialCommonDenominatorType}
\getchunk{defclass UnivariatePolynomialDecompositionPackageType}
\getchunk{defclass UnivariatePolynomialDivisionPackageType}
\getchunk{defclass UnivariatePolynomialFunctions2Type}
\getchunk{defclass UnivariatePolynomialMultiplicationPackageType}
\getchunk{defclass UnivariatePolynomialSquareFreeType}
\getchunk{defclass UnivariatePuiseuxSeriesFunctions2Type}
\getchunk{defclass UnivariateSkewPolynomialCategoryOpsType}
\getchunk{defclass UnivariateTaylorSeriesFunctions2Type}
\getchunk{defclass UnivariateTaylorSeriesODESolverType}
\getchunk{defclass UniversalSegmentFunctions2Type}
\getchunk{defclass UserDefinedPartialOrderingType}
\getchunk{defclass UserDefinedVariableOrderingType}
\getchunk{defclass UTSodetoolsType}
\getchunk{defclass U32VectorPolynomialOperationsType}
\getchunk{defclass VectorFunctions2Type}
\getchunk{defclass ViewDefaultsPackageType}
\getchunk{defclass ViewportPackageType}
\getchunk{defclass WeierstrassPreparationType}
\getchunk{defclass WildFunctionFieldIntegralBasisType}
```

```
\getchunk{defclass XExponentialPackageType}
\getchunk{defclass ZeroDimensionalSolvePackageType}
```

———–

Musings

This chapter contains snippets from the literature containing ideas that need to be "pondered". They may or may not survive and will almost certainly be modified beyond these notes. So this isn't part of the document, but rather a collection of notes for later.

1.100 Hardware Proof Checking

1.100.1 Mealy/Moore equivalence [Fade17]

Prove that if a Mealy machine is strongly connected and completely specified, the corresponding Moore machine will also be strongly connected and completely specified.

As the Mealy Machine is completely specified, we have 2 cases for each state:

1. All transitions coming into this state have the same output
2. At least 2 transitions coming into this state have different outputs

For case 1: The equivalent for this state would simply be changing the notation, i.e. mark all transitions into this state simply by the input symbols and just mention the output with the state as is.

Now, as the machine is completely specified, each such input transition and its corresponding output will be defined, and the states resulting due to such states will also have completely specified transitions.

For case 2: The equivalent for such a state requires the splitting/addition of new states such that each split state corresponds to the different outputs that can be generated. We split a state into 2 corresponding states.

Now, as the machine is completely specified, each such input transition and its corresponding output will be defined, and the states resulting due to such states will also have completely specified transitions.

Strongly Connected Machine: If for any two states S_i, S_j of the machine, there exists a finite sequence of input characters that can make the machine transition from S_i to S_j, the machine is strongly connected.

Completely Specified Machine: Consider $M = (I, O, S, F, Z)$

- **I** : Input Symbols
- **O** : Output Symbols
- **S** : Set of States
- **F** : Transition function $F : S \times I \rightarrow S$

- **Z** : Output function
- **Z** : $S \to O$ (Moore)
- **Z** : $S \times I \to O$ (Mealy)

Now, if for every (p, x) such that $(p \in S, x \in I)$ there exist some $q \in S$ and $y \in O$ such that $F(p, x) = q$ and $Z(p, x) = y$ (Mealy), $Z(p) = y$ (Moore).

Let $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be a Mealy machine where Δ is the output alphabet and λ is the output function.

Denote states of the corresponding Mealy machine by q_{ib} if there is input a and output b such that $\delta(q_j, a) = (q_i, b)$. This means if the Mealy machine enters a state q by emitting more than two different outputs then the state q_i splits. Also, denote corresponding Moore machine's transition function by δ'

Now consider two states q_{ib_1} and q_{jb_2} (of corresponding Moore machine). Existence of the state q_{jb_2} implies that there is a states q_k and input symbols a such that $\delta(q_k, a) = (q_j, b_2)$. Since we made the assumption that the Mealy machine is strongly connected, there is a string w such that $\delta(q_i, w) = (q_k, b_k)$ for some output symbol b_k. So, we have

$$\delta'(q_{ib_1}, wa) = \delta'(\delta'(q_{ib_1}, w), a) = \delta'(q_{kb_k}, a) = q_{jb_2}$$

This shows that the corresponding Moore machine is strongly connected.

As for the completely specification, let q_{ib_1} be a state of the Moore machine, and a is an input symbol. Then $\delta'(q_{ib_1}, a) = (q_{jb_2}, b_2)$ where $\delta(q_i, a) = (q_j, b_2)$, which shows it is completely specified.

1.101 Specification

1.101.1 Binary Search [Guen19]

A Binary Search in OCaml

— BS —

```
(* Requires a to be a sorted array of integers.
   Returns k such that i <= k < j and a.(k) = v
   or -1 if there is no such k *)

let rec bsearch a v i j =
  if j <= i then -1
  else
    let k = i + (j - i) / 2 in
    if v = a.(k) then k
    else if v < a.(k) then bsearch a v i k
    else bsearch a v (k+1) j
```

———-

There is an informal specification in the comments. The `bsearch` program takes as input an array of integers a (which is assumed to be sorted), a value v which is the target of the search, and lower and upper bounds i and j. If v can be found between indices i (included) and j (excluded), the `bsearch` returns its index in the array; otherwise it returns -1.

The Separation Logic specification for `bsearch` is

$\forall a\ xs\ v\ i\ j$
$0 \le i \le |xs| \land 0 \le j \le |xs| \land \texttt{sorted}\ xs \Rightarrow$
$\{a \rightsquigarrow \texttt{Array}\ xs\}\ \texttt{bsearch}\ a\ v\ i\ j\ \{\lambda k.\ a \rightsquigarrow \texttt{Array}\ xs * [\texttt{bsearch\_correct}\ xs\ v\ i\ j\ k]\}$

This statement consists of a Hoare triple ($\{H\}\ t\ \{Q\}$), after quantification (in the logic) over suitable arguments (the array location a and integers v, i, and j) as well as the logical model of the array xs (list of integers). Notice that we associate Coq integers (with v, i, j) with OCaml integers. This is not an abuse of notation but a feature of CFML, which reflects pure OCaml values directly as corresponding values in Coq. We require the indices i and j to be in bounds, and the contents of the array xs to be sorted. The Hoare triple which forms the conclusion of the statement relates an OCaml expression (here, a call to bsearch, "bsearch $a\ v\ i\ j$") with a pre-condition and post-condition describing the state of the heap before and after the execution, respectively. Both pre- and post-conditions are expressed using Separation Logic assertions; the post-condition is additionally parameterized by the value returned by the function (here, named k).

Generally speaking, Separation Logic assertions can be read in terms of ownership. For instance, the Separation Logic formula "$a \rightsquigarrow$ Array xs" asserts the unique ownership of an heap-allocated array, starting at local a, whose contents are described by the list xs. The operator $*$ should be read as a conjunction; and $[\cdot]$ lifts an ordinary Coq proposition as a Separation Logic assertion. To express the fact that the value returned by bsearch is indeed correct, we rely on an auxiliary definition bsearch_correct, which expresses the functional correctness property of bsearch. Thereafter, we write "$xs[i]$", where xs is a Coq list and i an integer, to denote the i-th element of the list xs (or a dummy value if i is not a valid index).

Functional Correctness of bsearch

$\texttt{bsearch\_correct}\ xs\ v\ i\ j\ k \triangleq$
$(k = -1 \land \forall\ p.\ i \le p < j \Rightarrow xs[p] \ne v)$
$\land\ (i \le k < j \land xs[k] = v)$

Assuming the preconditions, we have the following facts.

- bsearch will run and terminate without crashing (a consequence of the definition of Hoare triples)
- bsearch will not modify the contents of the input array ("$a \rightsquigarrow$ Array xs" appears in both the pre- and post-conditions, with the same xs)
- bsearch will return a correct result, according to the bsearch_correct predicate

A technical remark: one might wonder why the “pure” assumptions on i, j and xs are not part of the triple precondition, and instead are lifted as an implication at the meta-level (in the logic of Coq). Formally speaking, these two options are equivalent. That is, one could equivalently prove the following specification for bsearch

$\forall a\ xs\ v\ i\ j$
$\{a \rightsquigarrow \texttt{Array}\ xs * [0 \le i \le |xs| \land 0 \le j \le |xs| \land \texttt{sorted}\ xs]\}$
$\texttt{bsearch}\ a\ v\ i\ j$
$\{\lambda k.\ a \rightsquigarrow \texttt{Array}\ xs * [\texttt{bsearch\_correct}\ xs\ v\ i\ j\ k]\}$

1.102 Parsing

1.102.1 Top Down Operator Precedence [Prat73][?]

— TDOP —

```
def expression(rbp=0):
  global token
  t = token
  token = next()
  left = t.nud()
  while rbp < token.lbp:
    t = token
    token = next()
    left = t.led(left)
  return left

class literal_token(object):
  def __init__(self, value):
    self.value = int(value)
  def nud(self):
    return self.value

class operator_add_token(object):
  lbp = 10
   def led(self,left):
    right = expression(10)
    return left+right

class operator_mul_token(object):
  lbp = 20
  def led(self,left):
    return left * expression(20)

class end_token(object):
  lbp = 0
```

——————-

The tokenizer and parser

— TDOP —

```
import re
token_pat = re.compile("\s*(?:(\d+)|(.))")

def tokenize(program):
  for number, operator in token_pat.findall(program):
    if number:
      yield literal_token(number)
    elif
      yield operator_add_token()
    elif
      yield operator_mul_token()
    else:
      raise SyntaxError('unknown operator: %s', operator)
  yield end_token()

def parse(program):
  global token, next
  next = tokenize(program).next
  token = next()
  return expression()
```

———-

The **lbp** is the left binding power. For an infix operator, it tells us how strongly the operator binds to the argument on its left.

The **rbp** is the right binding power.

The **nud** (null denotation) is the prefix handler.

The **led** (left denotation) is the left handler, used to handle infix operators.

The **expression** handler function does the work. Operator handlers call it and pass in an **rbp**. It consumes tokens until it meets a token whose left binding power is equal or lower than the right binding power, that is until **lbp ¡= rbp**.

Handlers of operators call **expression** to process their arguments, providing it with their binding power to make user it gets just the right tokens from the input.

For example:

```
 3 + 1 * 2 * 4 + 5
```

has the call trace

```
expression with rbp 0
  literal nud = 3
  led of +
  expression with rbp 10
    literal nud = 1
    led of *
    expression with rbp 20
      literal nud = 2
    led of *
    expression with rbp 20
      literal nud = 4
  led of +
  expression with rbp 10
    literal nud = 5
```

3	+	1	*	2	*	4	+	5
–	–	–	–	0	–	–	–	–
		–	–	10	–	–		10
				20		20		

At each stage of the parser, there is an expression at each precedence level that is active at the moment. This instance awaits the results of the higher precedence instance and keeps going, until it has to stop itself and return to its caller.

Handling Unary Operators

TDOP makes an explicit distinction between unary and binary operators, encoding the difference between the **nud** and **let** methods. For example, the subtraction operator becomes

— TDOP —

```
class operator_sub_token(object):
  lbp = 10
  def nud(self):
    return -expression(100)
  def led(self, left):
    return left - expression(10)
```

———-

The **nud** handles the unary (prefix) form of minus. It has no left argument (since it is a prefix), and it negates its right argument, The binding power passed into **expression** is high since infix minus has a high precedence, higher than multiplication. The **led** handles the infix case similarly to the handlers of $+$ and $*$.

Right Associative Operators

1.103 Program proof

See Altenkirch et al. "Why Dependent Types Matter" [Alte05]

1.104 Notation

From Altenkirch [Alte18]

$$A \to B \equiv \Pi n : N.A^n \equiv \forall (x : A), B$$
$$A \to B \equiv \Pi \ - \ : \ A.B$$

$$ListA \equiv \Sigma_n : N.A^n$$
$$A \times B \equiv \Sigma \ - \ : \ A.B$$

1.105 Data Description Language

Jenkins [Jenk18] proposes combination of sum and product types, e.g.

```
data OrderResponse
  = PurchaseSuccessful
      { newOrder :: Order }
  | PaymentFailed
      { paymentProvider :: ProviderId,
        failureMessage  :: String }
  | NetworkError
      { statusCode :: Int,
        message    :: String
      }
```

1.106 Interactive Proof Semantics

1.106.1 Lean Semantics [Carn19b]

The Untyped Lambda Calculus

$$e := x \mid e \ e \mid \lambda \ x. \ e$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \ e_2 \rightsquigarrow e_1' \ e_2}$$

$$\frac{e_2 \rightsquigarrow e_2'}{e_1\ e_2 \rightsquigarrow e_1\ e_2'}$$

$$\frac{}{(\lambda x.\ e')\ e \rightsquigarrow e'[e/x]}$$

Simple Type Theory

$$\tau ::= \iota \mid \tau \to \tau$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\lambda x : \alpha . e) : \alpha \to \beta}$$

Dependent Type Theory

$$\tau ::= \iota \mid \tau \to \tau$$

$$e ::= x \mid e\ e \mid \lambda x : \tau . e$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\lambda x : \alpha . e) : \alpha \to \beta}$$

introducing $\forall x$:

$$\tau ::= \iota \mid \forall x : \tau .\ \tau$$

$$e ::= x \mid e\ e \mid \lambda x : \tau . e$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \forall x : \alpha . \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta[e_2/x]}$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\lambda x : \alpha . e) : \forall x : \alpha . \beta}$$

introducing universes:

$$\tau ::= \iota \mid \forall x : \tau.\ \tau \mid U$$

$$e ::= x \mid e\ e \mid \lambda x : \tau.e$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \forall x : \alpha.\beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta[e_2/x]}$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash (\lambda x : \alpha.e) : \forall x : \alpha.\beta}$$

$$\frac{}{\Gamma \vdash \iota : U}$$

$$\frac{\Gamma \vdash \alpha : U \qquad \Gamma, x : \alpha \vdash \beta : U}{\Gamma \vdash \forall x : \alpha.\beta : U}$$

$$\frac{}{\Gamma \vdash U : U}$$

1.107 Small Step Operational Semantics

1.107.1 Sylvan Clebsch [Cleb19]

$$\begin{array}{llll} x, y, z & \in & LocalID & \\ \rho & \in & Frame & = & LocalID \to Value \\ v & \in & Value & = & Integer \end{array}$$

$$\frac{}{\rho, e \rightsquigarrow \rho', e' | v}$$

To read a local variable:

$$\frac{x \in dom(\rho)}{\rho, x \rightsquigarrow \rho, \rho(x)}$$

To assign x a new value:

$$\frac{}{\rho, x = v \rightsquigarrow \rho[x \mapsto v], v}$$

To copy a value of y into x:

$$\frac{y \in dom(\rho) \qquad v = \rho(y)}{\rho, x = y \rightsquigarrow \rho[x \mapsto v], v}$$

Handling operators (e.g. $+$) (assuming integer $+$):

$$\frac{x \in dom(\rho) \qquad y \in dom(\rho) \qquad v = \rho(x) + \rho(y)}{\rho, z = x + y \rightsquigarrow \rho[z \mapsto v], v}$$

Small step semnatics allows us to take a single step and then decide what to do next. These two rules allow that.

Given a sequence of expressions e_1 and e_2, if e_1 were to evaluate to a new frame and an additional expression e_3 then the next step in the operation is to operate over that modified frame with e_3 followed by e_2:

$$\frac{\rho, e_1 \rightsquigarrow \rho', e_3}{\rho, e_1; e_2 \rightsquigarrow \rho', e_3, e_2}$$

If e_1 evaluates all the way to a value v then we are done and can evaluate e_2:

$$\frac{\rho, e_1 \rightsquigarrow \rho', v}{\rho, e_1; e_2 \rightsquigarrow \rho', e_2}$$

Assuming integer values only, if x is not zero then we evaluate to e_1 and drop e_2:

$$\frac{x \in dom(\rho) \qquad \rho(x) \neq 0}{\rho, \texttt{if } (x)\ \{e_1\} \texttt{ else } \{e_2\} \rightsquigarrow \rho, e_1}$$

The symmetric case, assuming integer values only, if x is zero then we evaluate to e_1 and drop e_2:

$$\frac{x \in dom(\rho) \qquad \rho(x) = 0}{\rho, \texttt{if } (x)\ \{e_1\} \texttt{ else } \{e_2\} \rightsquigarrow \rho, e_2}$$

These rules account for speculative execution.

$$\frac{}{\rho, \texttt{if } (*)\ \{e_1\} \texttt{ else } \{e_2\} \rightsquigarrow \rho, e_1}$$

$$\frac{}{\rho, \texttt{if } (*)\ \{e_1\} \texttt{ else } \{e_2\} \rightsquigarrow \rho, e_2}$$

Loops

$$\frac{x \in dom(\rho) \qquad \rho(x) \neq 0}{\rho, \texttt{while}(x)\{e\} \rightsquigarrow \rho, e;\ \texttt{while}(x)\{e\}}$$

Loop termination

$$\frac{x \in dom(\rho) \qquad \rho(x) = 0}{\rho, \texttt{while}(x)\{e\} \rightsquigarrow \rho, 0}$$

Heaps and Threads

$$\begin{array}{llll} v & \in & Value & = & Integer \mid Address \\ \chi & \in & Heap & = & \underline{Address} \mid Object \\ \sigma & \in & Stack & = & \overline{Frame} \\ \theta & \in & Thread & = & Stack \times Expression \end{array}$$

The expression gives us a new heap, stack, and frame.

$$\frac{}{\chi, \sigma \cdot \rho, e \rightsquigarrow \chi', \sigma' \cdot \rho', e | v}$$

where · is sequencing. The · is there so when we write preconditions we can talk about the Frame we are in.

Parallel Execution.

θ is a thread with a pair that is a stack and an expression. The overbar annotation means a sequence.

We non-deterministically select the thread to execute. The first rule is threads that are not done. The second rule is threads that are done. We have a (possible empty) sequence of threads $\overline{\theta}$, a particular thread, and another (possibly empty) sequence of threads. We select a thread to execute, which may terminate.

$$\frac{\chi, \sigma, e \rightsquigarrow \chi', \sigma', e'}{\chi, \overline{\theta}, (\sigma, e) \cdot \overline{\theta'} \rightsquigarrow \chi', \overline{\theta} \cdot (\sigma', e') \cdot \overline{\theta'}}$$

$$\frac{\chi, \sigma, e \rightsquigarrow \chi', \sigma', v}{\chi, \overline{\theta}, (\sigma, e) \cdot \overline{\theta'} \rightsquigarrow \chi', \overline{\theta} \cdot \overline{\theta'}}$$

1.108 Univalence and Homotopy Type Theory

1.108.1 Vladimir Voevodsky [Gray18]

The Univalence Axiom

$$(X = Y) \overset{\cong}{\equiv} (X \cong Y)$$

"Equality is equivalent to equivalence". In other words, if you have an equivalence between X and Y then you can promote that to an equality.

A function $f : X \to Y$ is an *equivalence* if, for each y in Y, there is just one x in X with $f(x) = y$.

The notation for a function being an equivalence is $f : X \xrightarrow{\cong} Y$

The notation for the type of all equivalences between X and Y is $X \overset{\cong}{\equiv} Y$

In Coq (by Voevodsky [Gray18a])

```
Definition eqweqmap {T1 T2 : UU} (e: paths T1 T2) : weq T1 T2.
Proof. intros, destruct e, apply idweq, Defined.

Axiom univalenceaxiom : forall T1 T2 : UU, isweq (@eqweqmap T1 T2).
```

where `isweq` is the $\overset{\cong}{\equiv}$ relation

Voevodsky's definition of a group:

Let U be a *universe*

A *group* in U is a sequence $(G, e, i, m, \lambda, \rho, \lambda', \rho', \alpha, \iota)$, where

- G is a type of U
- $e : G$
- $i : G \to G$

- $m : G \times G \to G$
- λ is a proof that for every $a : G, m(e, a) = a$
- ρ is a proof that for every $a : G, m(a, e) = a$
- λ' is a proof that for every $a : G, m(i(a), a) = e$
- ρ' is a proof that for every $a : G, m(a, i(a)) = e$
- α is a proof that for every $a, b, c : G : m(m(a, b), c) = m(a, m(b, c))$
- ι is a proof that G is a set

Why did he want Univalence? He knew from the beginning that it would be very important to correlate the results of different authors who were formalizing things in different ways. If you have one author giving one definition of triangle and another author giving another definition of triangle but they turn out to be equivalent then you would want the theorems of the second author to be useable by the first author and conversely. So you want some sort of mechanical way of transporting theorems between equivalent situations for maximum versatility. So what we have in mind here, from the start, is building a huge repository of mathematical theorems and their proofs and using that to build on forever. Using that as a daily aid to mathematicians in their research so that today they are not nervous about what they proved yesterday because they also formalized it in the computer.

What is needed is some sort of way to promote equivalences to something that can be used in the logic of the language to transport theorems between equivalent situations. – Dan Grayon [Gray18]

1.109 Various Logics

1.109.1 Barendregt's Lambda cube [Bare92, p78]

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

where

System	Logic
$\lambda\!\to$	(First Order) Propositional Calculus
$\lambda 2$	Second Order Propositional Calculus
$\lambda\underline{\omega}$	Weakly Higher Order Propositional Calculus
$\lambda\omega$	Higher Order Predicate Calculus
λP	(First Order) Predicate Logic
$\lambda P2$	Second Order Predicate Calculus
$\lambda P\underline{\omega}$	Weak Higher Order Predicate Calculus
$\lambda P\omega$	Calculus of Constructions

Moving along the X axis (left) $\rightarrow$ types depend on terms
Moving along the Y axis (up) $\rightarrow$ terms depend on types
Moving along the Z axis (back) $\rightarrow$ types depend on types

1.110 Lean and Theorem Proving

1.110.1 Jason Rute [Rute20]

A trivial program to add a goal to the list of goals from a command line input.

```
def experiment (s : string ) : lean.parser unit :=
do
  pe <- lean.parser.with_input interactive.types.texpr s, --parser monad
  e <- to_expr pe.1, -- tactic monad
  v <- mk_meta_var e,
  gs <- get_goals,
  set_goals $ v :: gs,
  t <- target
  trace t, -- should output the goal given by the string s
  return()

run_cmd experiment "true"


def experiment (s : string ) : lean.parser unit :=
do
  pe <- lean.parser.with_input interactive.types.texpr s, --parser monad
  e <-  lean.parser.of_tactic (tactic.to_expr pe.1),
  v <-  lean.parser.of_tactic (tactic.mk_meta_var e),
  gs <- lean.parser.of_tactic (tactic.get_goals),
  lean.parser.of_tactic (tactic.set_goals $ v :: gs),
  t <- lean.parser.of_tactic tactic.target,
  lean.parser.of_tactic (tactic.trace t),
  return ()

run_cmd experiment "true"


import system.io
open io

meta def serialize_expr (e : expr) : string := sorry
meta def deserialize_expr (s : string) : expr := sorry

meta def apply_tactic_to_goal (t : tactic unit) (goal : expr) : tactic (list expr) :=
do
```

```
  -- set expression as goal
  v <- tactic.mk_meta_var goal,
  tactic.set_goals $ [v],
  -- run tactic on goal
  t,
  -- return results
  gs <- tactic.get_goals,
  return gs

meta def read_eval_print (_ : unit) : io_core error (option unit) :=
do
  do
    --get serialized expression from stdin
    expr_str <- get_line,
    let e := deserialize_expr expr_str,
    -- run tactic (skep tactic in this case) on goal
    gs <- io.run_tactic (apply_tactic_to_goal tactic.skip e),
    -- print serialized goals to stdout
    let goal_strs := gs.map seriealize_expr,
    put_str (to_string goal_strs),
    return ()

meta def main : io unit :=
  -- loop forever
  do iterate () read_eval_print,
  return ()
```

1.110.2 Robert Y. Lewis [Lewi20]

This is an example from Lean 3. Note the use of the `carrier` which mirrors the Axiom Rep.

structure submodule (R : Type u) (M : Type v) [semiring R]
`[add_comm_monoid M] [module R M]` :=
(carrier : set M)
(zero : (0:M) ∈ carrier)
(add : ∀ {x,y}, x ∈ carrier → y ∈ carrier → x + y ∈ carrier)
(smul : ∀ (c:R) {x}, x ∈ carrier → c : x ∈ carrier)

Axiom NNI is

```
NonNegativeInteger() : SIG == CODE where

  SIG ==> Join(OrderedAbelianMonoidSup, Monoid) with

    _quo : (%, %) -> %
      ++ a quo b returns the quotient of \spad{a} and b, forgetting
      ++ the remainder.

    _rem : (%, %) -> %
      ++ a rem b returns the remainder of \spad{a} and b.

    gcd : (%, %) -> %
      ++ gcd(a, b) computes the greatest common divisor of two
      ++ non negative integers \spad{a} and b.
      ++
      ++X gcd(2415,945)
      ++X gcd(945,2415)
      ++X gcd(2415,0)
      ++X gcd(0,945)
      ++X gcd(15,15)
      ++X gcd(0,0)

    divide : (%, %) -> Record(quotient : %, remainder : %)
      ++ divide(a, b) returns a record containing both remainder and quotient.

    _exquo : (%,%) -> Union(%,"failed")
      ++ exquo(a,b) returns the quotient of \spad{a} and b, or "failed"
      ++ if b is zero or \spad{a} rem b is zero.

    shift : (%, Integer) -> %
      ++ shift(a, i) shift \spad{a} by i bits.

    random : % -> %
      ++ random(n) returns a random integer from 0 to \spad{n-1}.

    qcoerce : Integer -> %
      ++ qcoerce(n) coerces \spad{n} to \spad{%} trusting that
      ++ \spad{n} is nonnegative

  CODE ==> SubDomain(Integer, #1 >= 0) add

    x, y : %

    sup(x, y) == MAX(x, y)$Lisp

    shift(x : %, n : Integer) : % == ASH(x, n)$Lisp

    qcoerce(n) == n pretend %

    subtractIfCan(x, y) ==
      c : Integer := (x pretend Integer) - (y pretend Integer)
      c < 0 => "failed"
      c pretend %

    gcd(x,y) ==
      zero? x => y
      gcd(y rem x,x)
```

The Axiom NNI operations are:

```
 ?*? : (%,%) -> %
 ?*? : (NonNegativeInteger,%) -> %
 ?*? : (PositiveInteger,%) -> %
 ?**? : (%,PositiveInteger) -> %
 ?**? : (%,NonNegativeInteger) -> %
 ?+? : (%,%) -> %
 ?<? : (%,%) -> Boolean
 ?<=? : (%,%) -> Boolean
 ?=? : (%,%) -> Boolean
 ?>? : (%,%) -> Boolean
 ?>=? : (%,%) -> Boolean
 1 : () -> %
 0 : () -> %
 ?^? : (%,PositiveInteger) -> %
 ?^? : (%,NonNegativeInteger) -> %
 coerce : % -> OutputForm
```

```
gcd : (%,%) -> %
hash : % -> SingleInteger
latex : % -> String
max : (%,%) -> %
min : (%,%) -> %
one? : % -> Boolean
qcoerce : Integer -> %
?quo? : (%,%) -> %
random : % -> %
recip : % -> Union(%,"failed")
?rem? : (%,%) -> %
sample : () -> %
shift : (%,Integer) -> %
sup : (%,%) -> %
zero? : % -> Boolean
?~=? : (%,%) -> Boolean
divide : (%,%) -> Record(quotient: %,remainder: %)
exquo : (%,%) -> Union(%,"failed")
subtractIfCan : (%,%) -> Union(%,"failed")
```

So casting this in Lean might look like:
structure NonNegativeInteger (R : Type u) (M : Type v) [semiring R]
 [add_comm_monoid M] [module R M] :=
(carrier : set M)
(zero : (0:M) ∈ carrier)
(add : ∀ {x,y}, x ∈ carrier → y ∈ carrier → x + y ∈ carrier)
(smul : ∀ (c:R) {x}, x ∈ carrier → c : x ∈ carrier)

1.110.3 Andrej Bauer [Baue07]

Given General Indeuctive Types, the theory Branching describes that a branching type consists of a set s and a set t depending on s. The theory W is parameterized by a branching type B. It specifies a set w of well-founded trees and a tree-forming operation $tree$ with a dependent type $\Pi_{s \in B.s}(B.t(x) \rightarrow w) \rightarrow w$. Given a branching type x and a map $f : B.t(x) \rightarrow w$, $tree\ x\ f$ is the tree whose root has branching type x and whose successor labeled by $l \in B.t(x)$ is the tree $f(l)$. The inductive nature of w is expressed with the axiom induction, which states that for every property $M.p$, if $M.p$ is an inductive property then every tree satisfies it. A property is said to be *inductive* if a tree $tree\ x\ f$ satisfies it whenever all its successors satisfy it.

```
Parameter W : [B : Branching] →
thy
  Parameter w : Set.
  Parameter tree : [x : B.x] → (B.t x → w) → w.
  Axiom induction:
    ∀ M : thy Parameter p : w → Prop. end,
    (∀ x : B.x, ∀ f : B.t x → w,
      ((∀ y : B.t x, M.p (f y)) → M.p (tree x f))) →
    ∀ t : w, M.p t.
end.
```

A hand-written OCAML program is

```
module W (B : Branching) = struct
  type w = Tree of B.s * (B.t -> w)
  let tree x y = Tree (x, y)
  let rec induction f (Tree (x, g)) =
    f x g (fun y - induction f (g y))
```

```
end
```

The RZ program generates:

```
module type Branching =
  sig
    type s
    (** predicate (=s=) : s -> s -> bool *)
    (** assertion symmetric_s : forall x:s, y:s, x =s= y -> y =s= x
        assertion transitive_s :
          forall x:s, y:s, z:s, x =s= y /\ y =s= z -> x =s= z
    *)
    (** predicate ||s|| : s -> bool *)
    (** assertion total_def_s : forall x:s, x : ||s|| <-> x =s= x
    *)
    (** branching types *)
    type t
    (** predicate (=t=) : s -> t -> t -> bool *)
    (** assertion strict_t : forall x:s, y:t, z:t, y =(t x)= z -> x : ||s||
        assertion extensional_t :
          forall x:x, y:s, z:t, w:t, s =s= y -> z =(t x)= w -> z =(t y)= w
        assertion symetric_t :
          forall x:s, y:t z:t, y =(t x)= z -> z =(t x)= y
        assertion transitive_t :
          forall x:s, y:t, z:t, w:t, y =(t x)= z /\ z =(t x)= w ->
            y =(t x)= w
    *)
    (** predicate ||t|| : s -> t -> bool *)
    (** assertion total_def_t :
       forall x:s, y:t, y : ||t x|| <-> y =(t x)= y
    *)
    (** branch labels *)
  end

module W : functor (B : Branching) ->
sig
  type w
  (** prediate (=w=) : w -> w -> bool *)
  (** assertion symmetric_w :
       forall x:w, y:w, x =w= y -> y =w= x
      assertion transitive_w :
        forall x:w, y:w, z:w, x =w= y /\ y =w= z -> x =w= z
  *)
  (** predicate ||w|| : w -> bool *)
  (** assertion total_def_w : forall x:w, x : ||w|| <-> x =w= x
  *)
  val tree : B.s -> (B.t -> w) -> w
  (** assertion tree_support :
        forall x:B.s, y:B.s, x =B.s= y ->
          forall f:B.t -> w, g:B.t -> w,
            (forall z:B.t, t:B.t, z =(B.t x)= t -> f z =w= g t) ->
            tree x f =w= tree y g
  *)
  val induction : (B.s -> (B.t -> w) -> (B.t -> 'ty_p) -> 'ty_p) -> w -> 'ty_p
  (** assertion 'ty_p [p:w -> 'ty_p -> bool] induction :
        (forall x:w, a:'ty_p, p x a -> x : ||w||) ->
        (forall x:w, y:w, a:'ty_p, x =w= y -> p x a -> p y a) ->
        forall f:B.s -> (B.t -> w) -> (B.t -> 'ty_p) -> 'ty_p,
          (forall (x:||B.s||),
            forall f':B.t -> w,
              (forall y:B.t, z:B.t, y =(B.t x)= z ->
                f' y =w= f' z) ->
              forall g:B.t -> 'ty_p,
                (forall y:B.t, y : ||B.t x|| -> p (f' y) (g y)) ->
                p (tree x f') f x f' g)) ->
          forall (t:||w||), p t (induction f t)
  *)
end
```

From Bauer [Baue19]

- Type : Type ... a paradoxical universe of types
- $\Pi(x : T_1), T_2$ ... dependent products
- $\lambda(x : T) \to e$ ... functions
- $e_1 e_2$ ... application
- $e : T$ ... type ascriptions

- Definition x := e. ... define a value
- Axiom x : T. ... assume a constant
- Check e. ... check the type of an expression
- Eval e. ... evaluate an expression
- Load "`<file>`". ... load a file

From Bauer [Baue19a]

- $\Gamma \vdash A$type is a type
- $\Gamma \vdash t : A$ is a term of type A
- $\Gamma \vdash A \equiv B$ types A and B are equal
- $\Gamma \vdash t \equiv u : A$ terms t and u of type A are equal

From Bauer [Baue19a], Rules for dependent products

- rule Π (A type) ({x:A} B type) type
- rule λ (A type) ({x:A} B type) ({x:A} e : {B}) : Π A B
- app (A type) ({x:A} B type) (S : Π A B) (a : A) : B{a}
- rule Π_β (A type) ({x:A} B type) ({x:A} S : B{x}) (a : A)
  : app A B (λ A B S) a $\equiv$ S{a} : B{a}
- rule Id (A type (a : A) (b : A) type
- rule equality_reflection
  (A type (a:A0 (b:A) (p:Id A a b) : a $\equiv$ b : A

1.111 Verbx Regular Expressions

- `. .  A.C` match any character
- `[] [ABab] [A-Ca-c]` match any character in list
- `[^] [^Z] [^a]` match all except characters in list
- `^ ^C` next token must be the first part of string
- `$ [Ca]t$` prev token must be the last part of string
- `* [ab]*` match 0 or more copies
- `+ [ab]+` match 1 or more copies
- `\| c\|a` match either the 1st token or the second
- `\(\) \(CA\)+` combine multiple tokens into one

1.111.1 methods [verb20]

— verbx —

```
(defun sanitize (str)
  (replace str "([.$*+?^()\[\]{}\\|])" "\\\\$1"))
```

———–

add – Append a transformed value to internal expression that will compiled. This method should be private.

startOfLine – append ^ at start of expression

find – `"(?:" + value + ")"`

then – shorthand for **find**

maybe – 0 or 1 times `"(?:" + value + ")?"`

anything – matches everything `(?:.*)`

— verbx —

```
(defun anything (verbx)
  (add verbx "(?:.*)"))
```

———–

anythingBut – `"(?:[^" + value + "]*)"`

something – matches at least one char `(?:.+)`

— verbx —

```
(defun something (verbx)
  (add verbx "(?:.+)"))
```

———-

somethingBut – matches at least one char not in value `"(?:[^"+value+"]+)"`

replace – return replaced string

lineBreak – mathes linebreak `"(?:(?:\n)|(?:\r\n))"`

— **verbx** —

```
(defun lineBreak (verbx)
  (add verbx "(?:(?:\n)|(?:\r\n))"))
```

———-

br – shorthand for **linebreak**

— **axiom.bib** —

```
(defmacro br (verbx)
  `(lineBreak ,verbx))
```

———-

tab – match tab character `"\t"`

word – atches at least one word `"\w+"`

anyOf – matches any char in value `"(?:["+value+"])"`

any – shorthand for **anyOf**

— **axiom.bib** —

```
(defmacro any (verbx)
  `(anyOf ,verbx))
```

———-

range

withAnyCase – append modifier `"i"`

stopAtFirst – remove modifier `"g"` if `"true"`, else append modifier `"g"`

searchOneLine – if true, remove modifier `"m"`

multiple

or – split expression with `|`

beginCapture – initiate capture

endCapture – stop capturing elements

whitespace – `\\s`

oneOrMore – repeats the previous at least once.

Quote collections

The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct
– Donald Knuth, Selected Papers on Computer Science

Programmers are not mathematicians, no matter how much we wish and wish for it
– Richard P. Gabriel

Literate programs are a form of conversation, not docuentation
– Tim Daly

Say not 'This is the truth' but 'So it seems to me to be as I now see the things I think I see'
– David Love

Beware of bugs in the above code. I have only proved it correct, not tried it.
– Donald Knuth

No loop should be written down without providing proof of termination nor without stating the invariaant that will not be destroyed by the execution of the repeatable statement
– Edsger Dijkstra [Dijk72]

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.
– Edsger Dijkstra [Dijk72]

The competent programmer is fully aware of the strictly limited size of his own skull, therefore he approaches the programming task in full humility.
– Edsger Dijkstra [Dijk72]

For any perceived shortcoming of Common Lisp you can write a macro that fixes it for less effort than it takes to complain about it

A program without a specification cannot be incorrect, it can only be surprising
– J.J. Horning

A proof assistant is what happens if you spend all your time developing a type checker for your language and forget that programs also need to be run.
– Vladislav Zavialov [**?**]

Mathematicians are notoriously bad at given the complete context needed for definitions. One definition in the formal proof of the Kepler conjecture took 40 revisions to get right
– Thomas Hales

It would be a good thing to buy books if one could also buy the time to read them; but one usually confuses the purchase of books with the acquisition of their contents. To desire that a man should retain everything he has ever read, is the same as wishing him to retain in his stomach all that he has ever eaten.
– Arthur Schopenhauer [Scho16]

When one considers how hard it is to write a computer program even approaching the intellectual scope of a good mathematical paper, and how much greater time and effort have to be put into it to make it "almost" formally correct, it is preposterous to claim that mathematics as we practice it is anywhere near formally correct.
– William P. Thurston [Thur94]

Given the concepts 0, *number*, **and** *successor* **and the rules**

1. **0 is a** *number*
2. **the** *successor* **of any** *number* **is a** *number*
3. **no two** *numbers* **have the same** *successor*
4. **0 is not the** *successor* **of any** *number*
5. **any property which belongs to 0 and also to the** *succesor* **of every** *number* **which has the property, belongs to all the** *numbers*

– Giuseppe Peano [Russ1920]

I WANT a type theory with this strange property.
Well, then you deserve it.
– Andrej Bauer [Baue19a]

The Axiom Sane effort is, to quote Alan Kay, "the simplest thing that is qualitatively different from where we are".
– Tim Daly [Kayx19]

I am a biological copying machine.
– Tim Daly

Computers are intellectual mirrors. To program, you must be the machine.
– Tim Daly

From a certain point of view, having a convenient representation of one's behavior available for modification is what is meant by consciousness.
– John McCarthy [Mcca63]

Even when I write a one-shot program I write it in a literate way because I get it right faster that way.
– Donald Knuth [Knut19]

... the realization came over me with full force that a good part of theremainder of my life was going to be spent in finding errors in my own programs.
– Maurice Wilkes [Mack01]

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in a given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs.
– C. A. R. Hoare [Hoar69]

Let us avoid wasting those funds on pseudo mathematicians with computers and use them to support a few real live mathematicians.
–Frank Bonsall [Mack01]

Dependent types realize a continuum of precision up to a complete specification of the program's behaviour. Dependently typed programs are, by their nature, proof carrying code.
– Thorston AltenKirch, Conor McBride, James McKenna [Alte05]

Anything you can learn by a complete set of rules isn't worth knowing.
– Danial I.A. Cohen [Mack01]

Bought a mechanical keyboard with more resistance so my code will be strongly typed. – Jason Karns [Warn19]

DSL – A domain specific language, where code is written in one language and errors are given in another.
– Unknown [Unkn20]

Computers have enabled people to make more mistakes faster than almost any invention in history ... with the possible exception of tequila and hand guns.
– Ratcliffe [Brad20]

Me: There are a lot of 'Axiom' things around.
Simon: Of course. That's why we have the Axiom of Choice
– Simon Cruanes [Crua20]

Never trust a type theorist who has not changed their mind about equality at least once
– Conor McBride [Baze14]

A tecnnical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail.
– Vladimir Voevodsky [Gray18]

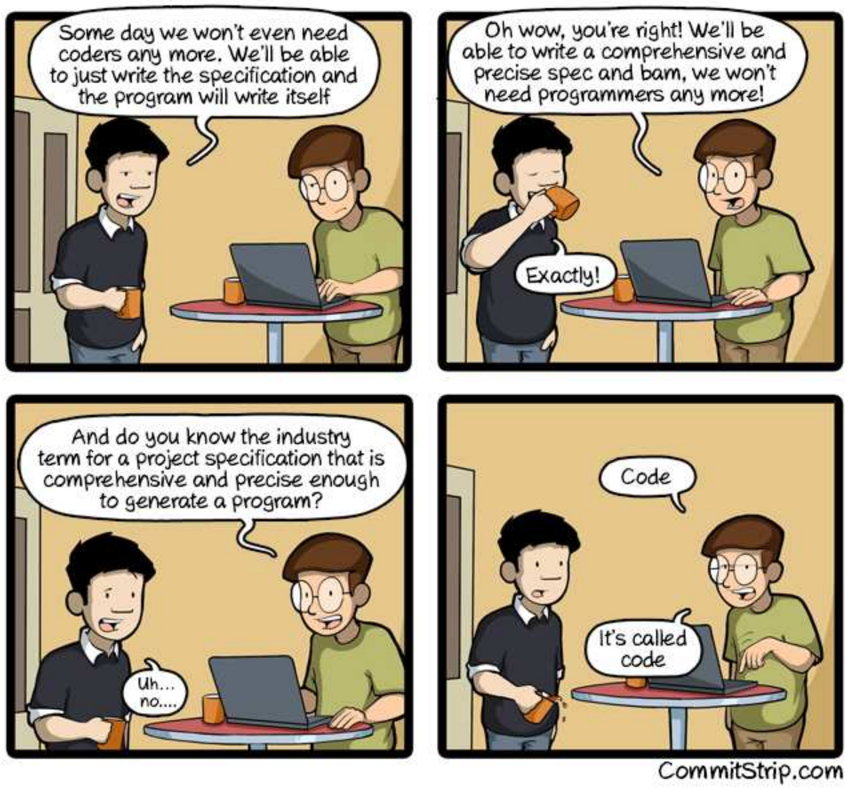
We're all on earth to help others – what I can't figure out is what the others are here for
– Marvin Minsky [Mcco79]

If I ask a student whether her design is as good as Chartres, she often smiles tolerantly at me as if to say "Of course not, that isn't what I am trying to do... I could never do that." Then, I express my disagreement, and tell her: "That standard *must* be ur standard."
– Christopher Alexander [Gabr96]

The distance is commonly very great between actual performances and speculative possibility. It is natural to suppose that as much as has been done today may be done tomorrow: but on the morrow some difficuty emerges, or some external impediment obstructs. Indolence, interruption, business, and pleasure, all take their turns of retardation; and every long work is lengthened by a thousand causes that can, and ten thousand that cannot, be recounted. Perhaps no extensive and multifarious performance was ever effected within the term originally fixed in the undertaker's mind. He that runs against Time has an antagonist not subject to casualties.
– Samuel Johnson [Gabr85]

I cannot claim to offer anything other than the notions of myself that I have formed over the space of roughly forty years, and their only singularity, it semms to me, is that they are not flattering.
– Peter Hogarth [Lemx68]

The shame of a genius may be his intellectual futility, the knowledge of how uncertain is all that he has accomplished.
– Peter Hogarth [Lemx68]


Some day we won't even need coders any more. We'll be able to just write the specification and the program will write itself
Oh wow, you're right! We'll be able to write a comprehensive and precise spec and bam, we won't need programmers any more!
Exactly!
And do you know the industry term for a project specification that is comprehensive and precise enough to generate a program?
Uh... no....
Code
It's called code
CommitStrip.com

[Comm16]

Coq Example Code

https://github.com/k-vs-coq-language-frameworks/coq/imp.v

```
(* Based on the imp example in coind,
   extracted to a single-file.
 *)
(* Developed with Coq 8.9.1 *)

Require Import String.
Require Import ZArith.
Require Import List.

Set Implicit Arguments.

Local Open Scope Z.
Import List.ListNotations.
Local Open Scope list.
Local Open Scope string.

(** * The syntax of IMP programs *)

Inductive AExp :=
  | var : string -> AExp
  | con : Z -> AExp
  | div : AExp -> AExp -> AExp
  | plus : AExp -> AExp -> AExp
  .

Inductive BExp :=
  | bcon : bool -> BExp
  | le : AExp -> AExp -> BExp
  | not : BExp -> BExp
  | and : BExp -> BExp -> BExp
  .

Inductive Stmt :=
  | assign : string -> AExp -> Stmt
  | cond : BExp -> Stmt -> Stmt -> Stmt
  | while : BExp -> Stmt -> Stmt
  | seq : Stmt -> Stmt -> Stmt
  | skip : Stmt
  .

Inductive Pgm :=
    pgm : list string -> Stmt -> Pgm.

(** Here is the sum program *)
Definition sum_pgm N : Pgm :=
 pgm ["n"; "sum"]
(seq (assign "n" (con N))
(seq (assign "sum" (con 0))
     (while (not (le (var "n") (con 0)))
     (seq (assign "sum" (plus (var "sum") (var "n")))
          (assign "n" (plus (var "n") (con (-1)))))))).

(** * The semantics of IMP programs *)

Definition Env := list (string * Z).
Definition empty_env : Env := [].
Fixpoint get x (env:Env) :=
  match env with
  | [] => None
  | (x',v)::env' =>
    if string_dec x x' then Some v else get x env'
  end.
Fixpoint set x v (env:Env) :=
  match env with
  | [] => []
  | (x',v')::env' =>
    if string_dec x x' then (x,v)::env' else (x',v')::set x v env'
  end.
  (* "simpl" should reduce set if concrete values are given for both variables *)

(* ** These "step" types together define single execution steps  *)
Inductive step_e : (AExp * Env) -> (AExp * Env) -> Prop :=
  | step_var: forall v x env, get v env = Some x ->
      step_e (var v, env) (con x, env)
  | step_plus: forall x y env,
      step_e (plus (con x) (con y), env) (con (Z.add x y), env)
  | step_div: forall x y env,
      y <> 0%Z ->
      step_e (div (con x) (con y), env) (con (Z.div x y), env)
  | cong_plus_r: forall e1 e2 e2' env env',
      step_e (e2, env) (e2', env') ->
      step_e (plus e1 e2, env) (plus e1 e2', env')
  | cong_plus_l: forall e2 e1 e1' env env',
      step_e (e1, env) (e1', env') ->
      step_e (plus e1 e2, env) (plus e1' e2, env')
  | cong_div_r: forall e1 e2 e2' env env',
      step_e (e2, env) (e2', env') ->
      step_e (div e1 e2, env) (div e1 e2', env')
  | cong_div_l: forall e2 e1 e1' env env',
      step_e (e1, env) (e1', env') ->
      step_e (div e1 e2, env) (div e1' e2, env')
  .

(* These abbreviations capture the pattern of the congruence rules *)
Notation cong_l op R1 R2 :=
  (forall a env a' env', R1 (a,env) (a',env') ->
   forall b, R2 (op a b, env) (op a' b, env')).
Notation cong_r op nf R1 R2 :=
  (forall b env b' env', R1 (b,env) (b',env') ->
   forall a, R2 (op (nf a) b, env) (op (nf a) b', env')).
Notation cong_1 op R1 R2 :=
  (forall a env a' env', R1 (a,env) (a',env') -> R2 (op a, env) (op a', env')).

Inductive step_b : (BExp * Env) -> (BExp * Env) -> Prop :=
  | eval_le : forall v1 v2 env,
      step_b (le (con v1) (con v2), env) (bcon (Z.leb v1 v2), env)
  | eval_not : forall b env,
      step_b (not (bcon b), env) (bcon (negb b), env)
  | eval_and : forall b e env,
      step_b (and (bcon b) e, env) (if b then e else bcon false, env)
  | cong_le_r : cong_r le con step_e step_b
  | cong_le_l : cong_l le step_e step_b
  | cong_not : cong_1 not step_b step_b
  | cong_and : cong_l and step_b step_b
  .

Inductive step_s : (Stmt * Env) -> (Stmt * Env) -> Prop :=
  | exec_assign : forall x v v0 env,  get x env = Some v0 ->
      step_s (assign x (con v),env) (skip, set x v env)
  | cong_assign : forall x,
      cong_1 (assign x) step_e step_s
  | exec_seq : forall s env,
      step_s (seq skip s,env) (s,env)
  | cong_seq : cong_l seq step_s step_s
  | exec_cond : forall b s1 s2 env,
      step_s (cond (bcon b) s1 s2, env) (if b then s1 else s2, env)
  | cong_cond : forall b b' env env' s1 s2, step_b (b,env) (b',env') ->
      step_s (cond b s1 s2,env) (cond b' s1 s2,env')
  | exec_while : forall b s env,
      step_s (while b s,env) (cond b (seq s (while b s)) skip, env)
  .

Inductive step_p : (Pgm * Env) -> (Pgm * Env) -> Prop :=
  | exec_init: forall x xs s env,
      step_p (pgm (x::xs) s,env) (pgm xs s, (x,0)::env)
  | exec_body: forall s env s' env',
      step_s (s, env) (s', env') ->
      step_p (pgm nil s, env) (pgm nil s', env')
  .

(** Now we verify the program *)
Require Import proof_system.

(** The claim about the loop says that running the loop in any enviroment
    with a non-negative n finishes with n set to zero, and sum increased
    from it's original value by the sum of numbers 0+1+...+n.
 *)
Inductive sum_spec : Spec (Pgm * Env) :=
 | sum_claim: forall n, 0 <= n ->
   sum_spec
     (pgm ["n"; "sum"]
       (seq (assign "n" (con n))
       (seq (assign "sum" (con 0))
            (while (not (le (var "n") (con 0)))
            (seq (assign "sum" (plus (var "sum") (var "n")))
                 (assign "n" (plus (var "n") (con (-1))))))))
     ,[])
     (fun cfg' => cfg' = (pgm [] skip, [("sum",((n + 1) * n)/2);("n",0)]))
 | sum_loop_claim : forall env n, get "n" env = Some n -> 0 <= n ->
                    forall s, get "sum" env = Some s ->
      sum_spec
         (pgm []
           (while (not (le (var "n") (con 0)))
             (seq (assign "sum" (plus (var "sum") (var "n")))
                  (assign "n" (plus (var "n") (con (-1))))))
         ,env)
      (fun cfg' => fst cfg' = pgm [] skip /\
         snd cfg' = set "n" 0 (set "sum" (s + ((n + 1) * n)/2) env)).

(* Some lemmas about enviroment stuff *)
Ltac env_ind_tac env :=
  induction env as [|[]];try reflexivity;simpl;
  repeat match goal with
  | [ |- context [string_dec ?a ?b]] => destruct (string_dec a b);simpl;try congruence
  end.

Lemma env_set_id: forall x v env,
    get x env = Some v ->
    set x v env = env.
Proof.
  env_ind_tac env.
  intro. f_equal. tauto.
Qed.

Lemma env_set_eq:
  forall x v1 v2 env,
    set x v1 (set x v2 env) = set x v1 env.
Proof. env_ind_tac env. Qed.

Lemma env_set_ne_comm:
  forall x1 x2, x1 <> x2 ->
  forall v1 v2 env,
    set x1 v1 (set x2 v2 env) = set x2 v2 (set x1 v1 env).
Proof. env_ind_tac env. Qed.

Lemma env_set_set: forall x1 x2 v1 v2 env,
    set x1 v1 (set x2 v2 env) =
    if string_dec x1 x2
    then set x1 v1 env
    else set x2 v2 (set x1 v1 env).
Proof. env_ind_tac env. Qed.

Definition env_has x env: bool :=
  match get x env with
  | Some _ => true
  | None => false
  end.

Lemma env_has_get x v env:
  get x env = Some v ->
  env_has x env = true.
Proof.
  unfold env_has;intros ->;reflexivity.
Qed.

Lemma env_get_set x x' v env:
  get x (set x' v env) =
  if string_dec x x'
  then if env_has x env then Some v else None
  else get x env.
Proof. unfold env_has; env_ind_tac env. Qed.

Lemma env_has_set x x' v env:
  env_has x (set x' v env) = env_has x env.
Proof.
  unfold env_has.
  rewrite env_get_set.
  unfold env_has.
  destruct (string_dec x x');[|reflexivity].
  destruct (get x env);reflexivity.
Qed.
```

```
Ltac step_tac :=
  match goal with
  | [ |- step_p _ _] => econstructor;step_tac
  | [ |- step_s _ _] => econstructor;step_tac
  | [ |- step_b _ _] => econstructor;step_tac
  | [ |- step_e _ _] => econstructor;step_tac
  | [ |- get _ _ = _] => rewrite ?env_get_set;(reflexivity || eassumption)
  end.

Ltac run := repeat first[
   eapply dtrans;[constructor|]
  |eapply ddone;simpl;split;[reflexivity|]
  |eapply dstep;[step_tac|]].

Require Import Recdef.
Function sum_to (n:Z) { wf (fun x y => 0 <= x < y) n } : Z :=
  if Z_lt_ge_dec 0 n then n + sum_to (n - 1) else 0.
intros;omega.
exact (Z.lt_wf 0).
Defined.

Lemma sum_algebra: forall s n, 0 < n ->
  s + n + (n + -1 + 1) * (n + -1) / 2
           = s + (n + 1) * n / 2.
Proof.
  intros s n H.
  rewrite <- Z.add_assoc.
  f_equal.
  rewrite <- Z.add_assoc, Z.add_0_r.
  rewrite <- Z.div_add_l by omega.
  f_equal.
  rewrite Z.mul_add_distr_r, Z.mul_add_distr_l.
  omega.
Qed.

Lemma sum_ok : sound step_p sum_spec.
apply proved_sound;destruct 1.

{ (* Overall claim, easily proved with loop claim *)
  eapply sstep;[solve[step_tac]|].
  run;[reflexivity || assumption ..|].
  destruct k';simpl.
  destruct 1 as [-> ->].
  apply ddone.
  reflexivity.
}


eapply sstep;[solve[step_tac]|].
run.
destruct (Z.leb_spec n 0);simpl.

(* when n = 0, loop exits.
   To conclude, need to prove that the initial
   environment env is an acceptable result *)
run.
replace n with 0 in H |- * by auto with zarith.
rewrite (env_set_id "sum") by (rewrite H1;f_equal;auto with zarith).
rewrite (env_set_id "n") by assumption.
```

```
reflexivity.

(* when n > 0, execution goes through the loop body,
   then sum_loop_claim is applied by transitivity,
   which takes us to a state satisfying the goal *)
run.
{ rewrite env_get_set, ?env_has_set.
  simpl.
  erewrite env_has_get by eassumption;reflexivity. }
  omega.
{ rewrite !env_get_set.
  simpl.
  erewrite env_has_get by eassumption;reflexivity. }
destruct k';simpl;intros [-> ->].

run.
rewrite (env_set_set "sum" "n"). simpl.
rewrite 2 env_set_eq.
f_equal.
f_equal.
apply sum_algebra.
assumption.
Qed.
```

The proof

```
Require Import String.
Require Import ZArith.
Require Import imp.

Local Open Scope Z.
Import List.ListNotations.
Local Open Scope list.
Local Open Scope string.

(** An execution trace is a sequence of zero or more steps.
    We will take the reflexive-transitive closure of the step
    relation using the library type clos_refl_trans_1n *)
Import Relation_Operators.

(** Execution tests can be written as a theorem claiming that
    an initial state reaches an expected final state *)
Lemma test_execution :
  clos_refl_trans_1n _ step_p (sum_pgm 100,[]) (pgm nil skip,[("sum",5050);("n",0)]).
Proof.
  Time repeat (eapply rt1n_trans;[once solve[repeat econstructor]|];simpl).
  apply rt1n_refl.
Time Qed.

(** The final state can also be filled in by the search.
    This statement uses a sort of existential quantification over the
    final environment e2 *)
Lemma test_execution2 :
  {e2:Env | clos_refl_trans_1n _ step_p (sum_pgm 100,[]) (pgm nil skip,e2)}.
Proof.
  eexists.
  Time repeat (eapply rt1n_trans;[once solve[repeat econstructor]|simpl]).
  apply rt1n_refl.
```

```
Time Defined.

(** The final environment can be extracted from this trace *)
Eval simpl in proj1_sig test_execution2.
```

Proving reachability

```
(*
  This file contains the main soundness proof allowing
  the reachability logic proof strategy to be used in Coq.

  'reaches x P' holds
  if any execution path from configuration x is either infinite or
  reaches a configuration satisfying P.

  'reaches' is shown to be the greatest fixpoint of the 'step' function.

  'stable_sound' is plain coinduction theorem for 'reaches' and 'step',
  'proved_sound' is the generalized coinduction theorem also allowing
  the "proof" rules defined in 'trans'.
 *)
Set Implicit Arguments.

Section relations.
Variables (cfg : Type) (cstep : cfg -> cfg -> Prop).

Definition Spec : Type := cfg -> (cfg -> Prop) -> Prop.

(* Soundness *)
CoInductive reaches (k : cfg) (P : cfg -> Prop) : Prop :=
  (* reaches : Spec, but defining k and P as parameters
     gives a cleaner definition and induction principle. *)
| rdone : P k -> reaches k P
| rstep : forall k', cstep k k' -> reaches k' P -> reaches k P.

Definition sound (Rules : Spec) : Prop :=
  forall x P, Rules x P -> reaches x P.

Inductive step (X : Spec) (k : cfg) (P : cfg -> Prop) : Prop :=
  (* step : Spec -> Spec *)
| sdone : P k -> step X k P
| sstep : forall k', cstep k k' -> X k' P -> step X k P.

Lemma reaches_stable :
  (forall x P, reaches x P -> step reaches x P).
Proof. destruct 1;econstructor;eassumption. Qed.

CoFixpoint stable_sound (Rules : Spec)
  (Hstable : forall x P, Rules x P -> step Rules x P)
  : sound Rules :=
  fun x P H =>
  match Hstable _ _ H with
    | sdone _ _ _ pf => rdone _ _ pf
    | sstep _ _ Hstep H' =>
        rstep Hstep (stable_sound Hstable _ _ H')
  end.

(*
Inductive derived (X : Spec) k P : Prop :=
| dclaim : X k P -> derived X k P
| ddone : P k -> derived X k P
| dstep : forall k', cstep k k' -> derived X k' P -> derived X k P
| dtrans' : forall Q, derived X k Q -> (forall k', Q k' -> derived X k' P)
    -> derived X k P
| dproved : reaches k P -> derived X k P.
*)

Inductive trans (X : Spec) (k : cfg) (P : cfg -> Prop) : Prop :=
  (* trans : Spec -> Spec *)
  | ddone : P k -> trans X k P
  | dtrans' : forall Q, trans X k Q -> (forall k', Q k' -> trans X k' P) -> trans X k P
  | drule : X k P -> trans X k P
  | dstep : forall k', cstep k k' -> trans X k' P -> trans X k P
  | dvalid : reaches k P -> trans X k P
  .
Definition dtrans_valid (X : Spec) (k : cfg) (P Q : cfg -> Prop)
  (rule : reaches k Q) (rest : forall k', Q k' -> trans X k' P) : trans X k P :=
  @dtrans' X k P Q (dvalid _ rule) rest.
Definition dtrans (X : Spec) (k : cfg) (P Q : cfg -> Prop)
  (rule : X k Q) (rest : forall k', Q k' -> trans X k' P) : trans X k P :=
  @dtrans' X k P Q (drule _ _ _ rule) rest.

Lemma trans_stable (Rules : Spec) :
  (forall x P, Rules x P -> step (trans Rules) x P)
  -> (forall x P, trans Rules x P -> step (trans Rules) x P).
induction 2;eauto using step.
destruct IHtrans; eauto using step, dtrans'.
destruct H0; eauto using step,dvalid.
Qed.

Lemma proved_sound (Rules : Spec) :
  (forall x P, Rules x P -> step (trans Rules) x P)
  -> sound Rules.
unfold sound.
intros H x P R.
eapply stable_sound.
apply trans_stable. eassumption.
apply drule. assumption.
Qed.

End relations.
```

Bibliography

Bibliography

[Acze13] Peter et al. Aczel. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

Link: https://hott.github.io/book/nightly/hott-letter-1075-g3c53219.pdf

[Alte05] Thorsten Altenkirch and Conor McBride. Why Dependent Types Matter, 2005.

Abstract: We exhibit the rationale behind the design of Epigram, a dependently typed programming language and interactive program development system, using refinements of a well known program – merge sort – as a running example. We discuss its relationship with other proposals to introduce aspects of dependent types into functional programming languages and sketch some topics for further work in this area.

Link: http://www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf

[Alte18] Thorsten Altenkirch. Naive Type Theory, 2018.

Link: https://www.youtube.com/watch?v=bNG53SA4n48p

[Altu19] Musab A. Alturki and Brandon Moore. K vs Coq as Language Verification Frameworks, 2019.

Link: https://runtimeverification.com/blog/k-vs-coq-as-language-verification-frameworks-pa

[Avig12a] Jeremy Avigad. Type Inference in Mathematics. *European Association of Theoretical Computer Science*, 106:78–98, 2012.

Abstract: In the theory of programming languages, type inference is the process of inferring the type of an expression automatically, often making use of information from the context in which the expression appears. Such mechanisms turn out to be extremely useful in the practice of interactive theorem proving, whereby users interact with a computational proof assistant to construct formal axiomatic derivations of mathematical theorems. This article explains some of the mechanisms for type inference used by the Mathematical Components project, which is working towards a verification of the Feit-Thompson theorem.

[Bake84] Henry Baker. The Nimble Type Inferencer for Common Lisp-84, 1984.

Abstract: We describe a framework and an algorithm for doing type inference analysis on programs written in full Common Lisp-84 (Common Lisp without the CLOS object-oriented extensions). The objective of type inference is to determine tight lattice upper bounds on the range of runtime data types for Common Lisp program variables and temporaries. Depending upon the lattice used, type inference can also provide range analysis information for numeric variables. This lattice upper bound information can be used by an optimizing compiler to choose more restrictive, and hence more efficient, representations for these program variables. Our analysis also produces tighter control flow information, which can be used to eliminate redundant tests which result in dead code. The overall goal of type inference is to mechanically extract from Common Lisp programs the same degree of representation information that is usually provided by the programmer in traditiional strongly-typed languages. In this way, we can provide some classes of Common Lisp programs execution time efficiency expected only for more strongly-typed compiled languages. The Nimble type inference system follows the traditional lattice/algebraic data flow techniques [Kaplan80], rather than the logical/theorem-proving unification techniques of ML [Milner78]. It can handle polymorphic variables and functions in a natural way, and provides for "case-based" analysis that is quite similar to that used intuitively by programmers. Additionally, this inference system can deduce the termination of some simple loops, thus providing surprisingly tight upper lattice bounds for many loop variables. By using a higher resolution lattice, more precise typing of primitive functions, polymorphic types and case analysis, the Nimble type inference algorithm can often produce sharper bounds than unification-based type inference techniques. At the present time, however, our treatment of higher-order data structures and functions is not as elegant as that of the unification techniques.

Link: `http://home.pipeline.com/~hbaker1/TInference.html`

[Bare92] Henk Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science (vol 2)*. Oxford University Press, 1992.

[Baue07] Andrej Bauer and Christopher A. Stone. RZ: a Tool for Bringing Constructive and Computational Mathematics Closer to Programming Practice, 2007.

Abstract: Realizability theory is not only a fundamental tool in logic and computability, but also has direct application to the design and implementation of programs: it can produce interfaces for the data structure corresponding to a mathematical theory. Our tool, called RS, serves as a bridge between the worlds of constructive mathematics and programming. By using the realizability interpretation of constructive mathematics, RZ translates specifications in constructive logic into annotated interface code in Objective Caml. The system supports a rich input language allowing descriptions of complex mathematical structures. RZ does not extract code from proofs, but allows any implementation method, from handwritten code to code extracted from proofs by other tools.

Link: `http://math.andrej.com/wp-content/uploads/2007/cie-long.pdf`

[Baue19] Andrej Bauer. How to Implement Type Theory in an Hour, 2019.

Comment: `https://github.com/andrejbauer/spartan-type-theory`

Link: `https://vimeo.com/286652934`

[Baue19a] Andrej Bauer. Derivations as computations, 2019.

Link: `https://youtube.com/watch?v=YZqOVsuyQyQ`

[Baze14] Gershom Bazerman. Homotopy Type Theory: What's the Big Idea, 2014.

Comment: Lambda Jam 2014

Link: https://www.youtube.com/watch?v=OupcXmLER7I

[Blai70a] Fred W. Blair, James H. Griesmer, and Richard D. Jenks. An interactive facility for symbolic mathematics. *ACM SIGSAM*, 14:17–18, 1970.

Abstract: This paper describes a system designed to provide an interactive symbolic computational facility for the mathematician user. To carry out this objective, an experimental LISP system has been implemented for IBM System/360 computers. Using this LISP system as a base, portions of several systems have been combined and augmented to provide the following facilities to a user:(1) rational function manipulation and simplification; symbolic differentiation (Anthony Hearn's REDUCE)(2) symbolic integration (Joel Moses' SIN)(3) polynomial factorization, solution of linear differential equations, direct and inverse symbolic Laplace transforms (Carl Engelman's MATHLAB, including Knut Korsvold's simplification system)(4) unlimited precision integer arithmetic(5) manipulation of arrays containing symbolic entries(6) two-dimensional output on IBM 2741 terminals or IBM 2250 displays (William Martin's Symbolic Mathematical Laboratory, and Jonathan Millen's CHARYBDIS program from MATHLAB)(7) self-extending language facility (META/LISP).The user language created for the system incorporates a subset of "customary" mathematical notation. Data objects include sequences (both finite and infinite) and arrays of arbitrary rank. Assignment statements are the fundamental commands in the user language; they may contain "for"-clauses which restrict the domain for which the assignment is valid and permit "piecewise" and recursive definition of new operators and functions. The user may also enter syntax definition statements in order to introduce new notations into the system.Expressions appearing in assignment statements may include "where"-clauses which allow user control over the "environment" used in evaluation. Otherwise, evaluation of expressions occurs in the current environment created by the successive user commands, with certain operations such as integration, differentiation, and simplification performed automatically.Translators for the user language and for a resident higher-level procedural language facility are written in META/LISP, a new self-compiling translator-writing system. As a result, all input language facilities as well as the underlying manipulation routines may be interactively extended by an experienced user.

[Bott19] Gert-Jan Bottu, Ningning Xie, Klara Mardirosian, and Tom Schrijvers. Coherence of Type Class Resolution, 2019.

Abstract: Elaboration-based type class resolution, as found in languages like Haskell, Mercury and PureScript, is generally *nondeterministic*: there can be multiple ways to satisfy a wanted constraint in terms of global instances and locally given constraints. Coherence is the key property that keeps this sane; it guarantees that, despite the nondeterminism, programs will behave predictably. even though elaboration-based resolution is generally assumed coherent, as far as we know, there is no formal proof of this property in the presence of sources of nondeterminism, like suerclasses and flexible contexts. This paper provides a formal proof to remedy the situation. The proof is non-trivial because the semantics elaborates resolution into a target language where different elaborations can be distinguished by contexts that do not have a source language counterpart. Inspired by the notion of full abstraction, we present a two-stop strategy that first elaborates nondeterministically into an intermediate language that preserves contextual equaivalence, and the deterministically elaborates from there into the target language. We use an approach based on logical relations to establish contextual equivalence and thus coherence for the first step of elaboration, while the second step's determinism straightforwardly preserves this coherence property.

Link: http://youtube.com/watch?v=bmHd0MoCliM

[Brad20] Sean Brady. Drop Your Tools – Does Expertise Have A Dark Side, 2020.

Link: https://www.youtube.com/watch?v=Yv4tI6939q0

[Brui68a] N.G. de Bruijn. AUTOMATH, A Language for Mathematics. technical report 68-WSK-05, Technische Hogeschool Eindhoven, 1968.

[Carn19a] Mario Carneiro. The Type Theory of Lean, 2019.

Abstract: This thesis is a presentation of dependent type theory with inductive types, a hierarchy of universes, with an impredicative universe of propositions, proof irrelevance, and subsingleton elimination, along with axioms for propositional extensionality, quotient types, and the axiom of choice. This theory is notable for being the axiomatic framework of the Lean theorem prover. The axiom system is given here in complete detail, including "optional" features of the type system such as **let** binders and definitions. We provide a reduction of the theory to a finitely axiomatized fragment utilizing a fixed set of inductive types (the **W**-type plus a few others), to ease the study of this framework. The metatheory of this theory (which we will Lean) is studied. In particular, we prove unique typing of the definitional equality, and use this to construct the expected set-theoretic model, from which we derive consistency of Lean relative to **ZFC+** $\{$there are n inaccessible cardinals $\mid n < \omega\}$ (a relatively weak large cardinal assumption). As Lean supports models of **ZFC** with n inaccessible cardinals, this is optimal. We also show a number of negative results, where the theory is less nice than we would like. In particular, type checking is undecidable, and the type checking as implemented by the Lean theorem prover is a decideable non-transitive underapproximation of the typing judgment. Non-transitivity also leads to lack of subject reduction, and the reduction relation does not satisfy the Church-Rosser property, so reduction to a normal form does not produce a decision procedure for definitional equality. However, a modified reduction relation allows us to restore the Church-Rosser property at the expense of guaranteed termination, so that unique typing is shown to hold.

Comment: https://www.youtube.com/watch?v=3sKrSNhSxik

Link: https://github.com/digama0/lean-type-theory/releases/v1.0

[Carn19b] Mario Carneiro. The Type Theory of Lean (slides), 2019.

Link: https://github.com/digama0/lean-type-theory/releases/v1.0

[Chli17] Adam Chlipala. *Formal Reasoning About Programs*. MIT, 2017.

Abstract: Briefly, this book is about an approach to bringing software engineering up to speed with more traditional engineering disciplines, providing a mathematical foundation for rigorous analysis of realistic computer systems. As civil engineers apply their mathematical canon to reach high certainty that bridges will not fall down, the software engineer should apply a different canon to argue that programs behave properly. As other engineering disciplines have their computer-aided design tools, computer science has proof assistants, IDEs for logical arguments. We will learn how to apply these tools to certify that programs behave as expected. More specifically: Introductions to two intertangled subjects: the Coq proof assistant, a tool for machine-checked mathematical theorem proving; and formal logical reasoning about the correctness of programs.

Link: http://adam.chlipala.net/frap/frap_book.pdf

[Cleb19] Sylvan Clebsch. Starting with Semantics, 2019.

Link: https://www.youtube.com/watch?v=JbS8a-Ba0Ck

[Comm16] Unknown. A very comprehensive and precise spec, 2016.

Link: http://www.commitstrip.com/en/2016/08/25/a-very-comprehensive-and-precise-spec

[Crua20] Simon Cruanes. Axiom of Choice, 2020.

Comment: Axiom: That's why we have the Axiom of Choice

Link: https://leanprover.zulipchat.com/#narrow/stream/113488-general

[Daly17] Timothy Daly and Mark Botch. Axiom Developer Website, 2017.

Link: http://axiom-developer.org

[Daly18] Timothy Daly. Proving Axiom Sane: Survey of CAS and Proof Cooperation, 2018.

Comment: In Preparation

[Dave02] James H. Davenport. Equality in computer algebra and beyond. *J. Symbolic Computing*, 34(4):259–270, 2002.

Abstract: Equality is such a fundamental concept in mathematics that, in fact, we seldom explore it in detail, and tend to regard it as trivial. When it is shown to be non-trivial, we are often surprised. As is often the case, the computerization of mathematical computation in computer algebra systems on the one hand, and mathematical reasoning in theorem provers on the other hand, forces us to explore the issue of equality in greater detail.In practice, there are also several ambiguities in the definition of equality. For example, we refer to $\mathbb{Q}(x)$ as "rational functions", even though $\frac{x^2-1}{x-1}$ and $x+1$ are not equal as functions from $\mathbb{R}$ to $\mathbb{R}$, since the former is not defined at $x=1$, even though they are equal as elements of $\mathbb{Q}(x)$. The aim of this paper is to point out some of the problems, both with mathematical equality and with data structure equality, and to explain how necessary it is to keep a clear distintion between the two.

Link: http://www.calculemus.net/meetings/siena01/Papers/Davenport.pdf

[Dave84] James H. Davenport, Patrizia Gianni, Richard D. Jenks, Victor Miller, Scott C. Morrison, Michael Rothstein, Christine Sundaresan, Robert S. Sutor, and Barry M. Trager. *Scratchpad*. Mathematical Sciences Department, IBM Thomas Watson Research Center, Yorktown Heights, NY, 1984.

[Dave85] James H. Davenport. The LISP/VM Foundation of Scratchpad II. *The Scratchpad II Newsletter*, 1(1), September 1985.

[Demi79] Richard A. DeMilo, Richard J. Lipton, and Alan J. Perlis. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22(5):271–280, 1979.

Abstract: It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.

[Dijk00] Edsger W. Dijkstra. EWD1305 Archive: Answers to questions from students of Software Engineering, 2000.

Link: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1305.html

[Dijk72] Edsger Dijkstra. The Humble Programmer, 1972.

Comment: ACM Turing Lecture 1972

[Dijk78] Edsger W. Dijkstra. A Political Pamphlet From The Middle Ages, 1978.

Comment: EWD638

Link: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD638.html

[Dijk90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990, 0-387-96957-8.

[Dosr11] Gabriel Dos Reis, David Matthews, and Yue Li. *Retargeting OpenAxiom to Poly/ML: Towards an Integrated Proof Assistants and Computer Algebra System Framework*, pages 15–29. Springer, 2011, 978-3-642-22673-1.

Abstract: This paper presents an ongoing effort to integrate the Axiom family of computer algebra systems with Poly/ML-based proof assistants in the same framework. A long term goal is to make a large set of efficient implementations of algebraic algorithms available to popular proof assistants, and also to bring the power of mechanized formal verification to a family of strongly typed computer algebra systems at a modest cost. Our approach is based on retargeting the code generator of the OpenAxiom compiler to the Poly/ML abstract machine.

Link: http://paradise.caltech.edu/~yli/paper/oa-polyml.pdf

[Fade17] Unknown. Strongly Connected and Completely Specified Moore Equivalent of a Mealy Machine, 2017.

Link: https://cs.stackexchange.com/questions/81127/strongly-connected-and-completely-specif

[Fetz88] James H. Fetzer. Program Verification: The Very Idea. *Communications of the ACM*, 31(9):1048–1063, 1988.

Abstract: The notion of program verification appears to trade upon an equvocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.

[Ford04] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Notices*, 39(1):111–122, 2004.

> **Abstract:** For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use *prioritized choice*. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for constructing grammars, they are reducible to two minimal recognition schemas developed around 1970, TS/TDPL and gTS/GTDPL, which are here proven equivalent in effective recognition power.

[Gabr85] Richard Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

> **Link:** `https://www.dreamsongs.com/Files/Timrep.pdf`

[Gabr96] Richard Gabriel. *Patterns of Software*. Oxford University Press, 1996.

> **Link:** `https://www.dreamsongs.com/Files/PatternsOfSoftware.pdf`

[Gray18] Daniel R. Grayson. An Introduction to Univalent Foundations for Mathematicians, 2018.

> **Abstract:** We offer an introduction for mathematicians to the univalent foundations of Vladimir Voevodsky, aiming to explain how he chose to encode mathematics in type theory and how the encoding reveals a potentially viable foundation for all of modern mathematics that can serve as an alternative to set theory
>
> **Link:** `http://arxiv.org/pdfs/1711.01477v3`

[Gray18a] Daniel R. Grayson. The Mathematical Work of Vladimir Voevodsky, 2018.

> **Link:** `https://www.youtube.com/watch?v=BanMgvdKP8E`

[Guen19] Armael Gueneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. PhD thesis, University of Paris, 2019.

> **Abstract:** This dissertation is concerned with the question of formally verifying that the implementation of an algorithm is not only functionally correct (it always returns the right result), but also has the right asymptotic complexity (it reliably computes the result in the expected amount of time). In the algorithms literature, it is standard practice to characterize the performance of an algorithm by indicating its asymptotic time complexity, typically using Landau's "big-O" notation. We first argue informally that asymptotic complexity bounds are equally useful as formal specifications, because they enable modular reasoning: a O bound abstracts over

the concrete cost expression of a program, and therefore abstracts over the specifics of its implementation. We illustrate – with the help of small illustrative examples – a number of challenges with the use of the O notation, in particular in the multivariate case, that might be overlooked when reasoning informally. We put these considerations into practice by formalizing the O notation in the Coq proof assistant, and by extending an existing program verification framework, CFML, with support for a methodology enabling robust and modular proofs of asymptotic complexity bounds. We extend the existing framework of Separation Logic with Time Credits, which allows to reason at the same time about correctness and time complexity, and introduce negative time credits. Negative time credits increase the expressiveness of the logic, and enable convenient reasoning principles as well as elegant specifications. At the level of specifications, we show how asymptotic complexity specifications using O can be integrated and composed within Separation Logic with Time Credits. Then, in order to establish such specifications, we develop a methodology that allows proofs of complexity in Separation Logic to be robust and carried out at a relatively high level of abstraction, by relying on two key elements: a mechanism for collecting and deferring constraints during the proof, and a mechanism for semi-automatically synthesizing cost expressions without loss of generality. We demonstrate the usefulness and practicality of our approach on a number of increasingly challenging case studies. These include algorithms whose complexity analysis is relatively simple (such as binary search, which is nonetheless out of the scope of many automated complexity analysis tools) and data structures (such as Okasaki's binary random access lists). In our most challenging case study, we establish the correctness and amortized complexity of a state-of-the-art incremental cycle detection algorithm: our methodology scales up to highly non-trivial algorithms whose complexity analysis intimately depends on subtle functional invariants, and furthermore makes it possible to formally verify OCaml code which can then actually be used as part of real world programs.

Link: http://gallium.inria.fr/~agueneau/phd/manuscript.pdf

[HEAR80] Anthony C. Hearn. Symbolic Computation and its Application to High Energy Physics. In *Proc. 1980 CERN School of Computing*, pages 390–406, 1980.

Abstract: It is clear that we are in the middle of an electronic revolution whose effect will be as profound as the in dustrial revolution. The continuing advances in computing technology will provide us with devices which will make present day computers appear primitive. In this environment, the algebraic and other non-numerical capabilities of such devices will become increasingly important. These lectures will review the present state of the field of algebraic computation and its potential for problem solving in high energy physics and related areas. We shall begin with a brief description of the available systems and examine the data objects which they consider. As an example of the facilities which these systems can offer, we shall then consider the problem of analytic integration, since this is so fundamental to many of the calculational techniques used by high energy physicists. Finally, we shall study the implications which the current developments in hardware technology hold for scientific problem solving.

Link: http://www.iaea.org/inis/collection/NCLCollectionStore/_Public/12/631/12631585.pdf

[Harp18] Robert Harper. Computational Type Theory, 2018.

Comment: OPLSS 2018

Link: http://www.cs.uoregon.edu/research/summerschool/summer18/topics.php

[Harp18a] Robert Harper. Computational Type Theory Lectures, 2018.

Comment: OPLSS 2018

Link: http://www.youtube.com/watch?v=LE0SSLizYUI

[Hoar03] Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50(1):63–69, 2003.

Abstract: This contribution proposes a set of criteria that distinguish a grand challenge in science or engineering from the many other kinds of short-term or long-term research problems that engage the interest of scientists and engineers. As an example drawn from Computer Science, it revives an old challenge: the contruction and application of a verifying compiler that guarantees correctness of a program before running it.

Link: https://vimeo.com/39256698

[Hoar69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–580, 1969.

Abstract: In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics

Link: https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf

[Huds19] Andrew Dana Hudson. A Priest, a Rabbi, and a Robot Walk Into a Bar, 2019.

Abstract: A new short story looks at how artificial intelligence could support, and distort, faith.

Link: https://slate.com/technology/future-tense-fiction-priest-rabbi-robot-bar-andrew-huds

[Hugh19] John Hughes. How to Specify it!, 2019.

Abstract: Property-based testing tools test software against a specification, rather than a set of examples. This tutorial paper presents five generic approaches to writing such specifications (for purely functional code). We discuss costs, benefits, and bug-finding power of each approach, with reference to a simple example with eight buggy variants. The lessons learned should help the reader to develope effective property-based tests in the future.

Link: https://www.dropbox.com/s/tx2b84kae4bw1p4/paper.pdf

[IBMx91] Computer Algebra Group. The AXIOM Users Guide, 1991.

[Jaff93] Arthur Jaffe and Frank Quinn. "Theoretical Mathematics": Towards a Cultural Synthesis of Mathematics and Theoretical Physics. *Bull. of the American Mathematical Society*, 29(1):1–13, 1993.

Abstract: Is specualative mathematics dangerous? Recent interactions between physics and mathematics pose the question with some force: traditional mathematical norms discourage speculation, but it is the fabric of theoretical physics. In practice there can be benefits, but there can also be unpleasant and destructive consequences. Serious caution is required, and the issue should be considered before, rather than after, obvious damage occurs. With the hazards carefully in mind, we propose a framework that should allow a healthy and positive role for speculation.

[Jenk18] Kris Jenkins. Communicating with Types, 2018.

Abstract: Modern type systems have come a long way from the days of C and Java. Far from being nit-pickers that berate us for making mistakes, type systems like the ones found in Haskell, PureScript and Elm form a language in themselves. A language for expressing high-level ideas about our software to our colleagues and to the computer. A design language. In this talk, we'll take a look at the way the right kind of type signatures let us talk about software. We'll survey how to state our assumptions about the domain we're coding in and how each part fits together. We'll show how it can highlight problems, and surface opportunities for reuse. And most importantly, we'll see how types can help you communicate to your coworkers, and to future maintainers, with little effort. You've probably heard the phrase "programs are meant to be read by humans and only incidentally for computers to execute". Come and see how a modern type system is about communicating ideas to humans, and only incidentally about proving correctness.

Link: `https://vimeo.com/302682323`

[Kahr98] Stefan Kahrs and Donald Sannella. Reflections on the Design of a Specification Language. *LNCS*, 1382:154–170, 1998.

Abstract: We reflect on our experiences from work on the design and semantic underpinnings of Extended ML, a specification language which supports the specification and formal development of Standard ML programs. Our aim is to isolate problems and issues that are intrinsic to the general enterprise of designing a specification language for use with a given programming language. Consequently the lessons learned go far beyond our original aim of designing a specification language for ML.

[Kayx19] Alan Kay. Alan Kay Speaks at ATLAS Institute, 2019.

Link: `https://www.youtube.com/watch?v=nOrdzDaPYV4`

[Knut19] Donald E. Knuth. Donald Knuth: Algorithms, Complexity, Life, and The Art of Computer Programmig, 2019.

Comment: Lex Fridman AI Podcast

Link: `https://www.youtube.com/watch?v=2Bd8fsXbST8`

[Knut92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford CA, 1992, 0-937073-81-4.

[Lemx68] Stanislaw Lem. *His Master's Voice*. MIT Press, 1968.

[Lewi20] Robert Y. Lewis. The Lean Mathematical Library, 2020.

Link: https://www.youtube.com/watch?v=fnRrwsaFUM8

[Lica16] Dan Licata. A Functional Programmer's Guide to Homotopy Type Theory, 2016.

Comment: ICFP conference

Link: https://www.youtube.com/watch?v=caSOTjr1z18

[Lica16a] Dan Licata. Computing with Univalence, 2016.

Comment: Institute for Advanced Study

Link: https://www.youtube.com/watch?v=YDdDuq2AGw4

[Mack01] Donald MacKenzei. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 2001, 0-262-13393-8.

[Mart79] P. Martin-Lof. Constructive Mathematics and Computer Programming. In *Proc. 6th Int. Congress for Logic, Methodology and Philosophy of Science*, pages 153–179. North-Holland, 1979.

Abstract: If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute (a difference that Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics. This explains why the intuitionistic theory of types, which was originally developed as a symbolism for the precise codification of constructive mathematics, may equally well be viewed as a programming language. As such it provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Moreover, the inference rules of the theory of types, which are again completely formal, appear as rules of correct program synthesis. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.

[Mcca63] John McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*. Elsevier, 1963.

[Mcco79] Pamela McCorduck. *Machines Who Think*. Freeman, 1979.

[Miln84] R. Milner. The Use of Machines to Assist in Rigorous Proof. *Philosophical Transactions of the Royal Society*, 312(1522):411–422, 1984.

Abstract: A methodology for computer assisted proof is presented with an example. A central ingredient in the method is the presentation of tactics (or strategies) in an algorithmic metalanguage. Further, the same language is also used to express combinators, by which simple elementary tactics - which often correspond to the inference rules of the logic employed - are combined into more complex tactics, which may even be strategies complete for a class of problems. However, the emphasis is not upon completeness but upon providing a metalogical framework within which a user may express his insight into proof methods and may delegate routine (but error-prone) work to the computer. This method of tactic composition is presented at the start of the paper in the form of an elementary theory of goal-seeking. A second ingredient of the methodology is the stratification of machine-assisted proof by an ancestry graph of applied theories, and the example illustrates this stratification. In the final section, some recent developments and applications of the method are cited.

[Miln90] Robin Milner, Mads Torte, and Robert Harper. *The Definition of Standard ML*. Lab for Foundations of Computer Science, Univ. Edinburgh, 1990.

Link: http://sml-family.org/sml90-defn.pdf

[Miln91] Robin Milner and Mads Torte. *Commentary on Standard ML*. Lab for Foundations of Computer Science, Univ. Edinburgh, 1991.

Link: https://pdfs.semanticscholar.org/d199/16cbbda01c06b6eafa0756416e8b6f15ff44.pdf

[Moor18] Brandon Moore, Lucas Pena, and Grigore Rosu. Program Verification by Coinduction. In *Programming Languages and Systems*, pages 589–618. Springer, 2018.

Abstract: We present a novel program verification approach based on coinduction, which takes as input an operational semantics. No intermediates like program logics or verifcation condition generators are needed. Specifications can be written using any state predicates. We implement our approach in Coq, giving a certifying language-independent verification framework. Our proof system is implemented as a single module imported unchanged into language-specific proofs. Automation is reached by instantiating a generic heuristic with language-specific tactics. Manual assistance is also smoothly allowed at points the automation cannot handle. We demonstrate the power and versatility of our approach by verifying algoirthms as complicated as Schorr-Waite graph marking and instantiating our framework for object languages in several styles of semantics. Finally, we show that our coinductive approach subsumes reachability logic, a recent language-independent soudn and (relatively) complete logic for program verification that has been instantiated with operational semantics of languages as complex as C, Java and Javascript.

Link: https://link.springer.com/content/pdf/10.1007%2F978-3-319-89884-1.pdf

[Mose71] Joel Moses. Symbolic Integration: The Stormy Decade. *CACM*, 14(8):548–560, 1971.

Abstract: Three approaches to symbolic integration in the 1960's are described. The first, from artificial intelligence, led to Slagle's SAINT and to a large degree to Moses' SIN. The second, from algebraic manipulation, led to Monove's implementation and to Horowitz' and Tobey's reexamination of the Hermite algorithm for integrating rational functions. The third, from mathematics, led to Richardson's proof of the unsolvability of the problem for a class of functions and for Risch's decision procedure for the elementary functions. Generalizations of Risch's algorithm to a class of special functions and programs for solving differential equations and for finding the definite integral are also described.

Link: http://www-inst.eecs.berkeley.edu/~cs282/sp02/readings/moses-int.pdf

[Pear15] David J. Pearce and Lindsay Groves. Designing a Verifying Compiler: Lessons Learned from developing Whiley, 2015.

Abstract: An ongoing challenge for computer science is the development of a tool which automatically verifies programms meet their specifications, and are free from runtime errors such as divide-by-zero, array out-of-bounds and null dereferences. Several impressive systems have been developed to this end, such as ESC/Java and Spec#, which build on existing programming languages (e.g., Java, C#). We have been developing a programming language from scratch to simplify verification, called Whiley, and an accompanying verifying compiler. In this paper, we present a technical overview of the verifying compiler and document the numerous design decisions made. Indeed, many of our designs reflect thos of similar tools. However, they have often been ignored in the literature and/or spread thinly throughout. In doing this, we hope to provide a useful resource for those building verifying compilers.

Link: https://homepages.ecs.vuw.ac.nz/~djp/files/SCP15-preprint.pdf

[Prat73] Vaughan R. Pratt. Top Down Operator Precedence. In *Proc. 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'73, pages 41–51, 1973.

Link: http://hall.org.ua/halls/wizzard/pdf/Vaughan.Pratt.TDOP.pdf

[Pres19] Ron Pressler. Correctness and Complexity, 2019.

Link: https://pron.github.io/posts/correctness-and-complexity

[Robi65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM*, 12:23–41, 1965.

Abstract: Theorem-proving on the computer, using procedures based on the fundamental theorem of Herbrand concerning the first-order predicate calculus, is examined with a view towards improving the efficiency and widening the range of practical applicability of these procedures. A elose analysis of the process of substitution (of terms for variables), and the process of truth-functional analysis of the results of such substitutions, reveals that both processes can be combined into a single new process (called resolution), iterating which is vastty more effieient than the older cyclic procedures consisting of substitution stages alternating with truth-functional analysis stages. The theory of the resolution process is presented in the form of a system of first-order logic with .just one inference principle (the resolution principle). The completeness of the system is proved; the simplest proof-procedure based on the system is then the direct implementation of the proof of completeness. However, this procedure is quite inefficient, and the paper concludes with a discussion of several principles (called search principles) which are applicable to the design of efficient proof-procedures employing resolution as the basic logical process.

[Russ1920] Bertrand Russell. *Introduction to Mathematical Philosophy*. George Allen and Unwin, Ltd, 1920.

[Rute20] Jason Rute. Communicating with Lean, 2020.

Link: https://github.com/jasonrute/communicating-with-lean/blob/master/communicate_with_lea

[Sann91] D. Sannella and A. Tarlecki. Formal Program Development in Extended ML for the Working Programmer. In *3rd BCS/FACS Workshop on Refinement*, pages 99–130. Springer, 1991.

Abstract: Extened ML is a framework for the formal development of programs in the Standard ML programming language from high-level specifications of their required input/output behavior. It strongly supports the development of modular programs consisting of an interconnected collection of generic and reusable units. The Extended ML framework includes a methodology for formal program development which establishes a number of ways of proceeding from a given specification of a programming task towards a program. Each such step gives rise to one or more proof oblisgations which must be proved in order to establish the correctness of that step. This paper is inteded as a user-oriented summary of the Extended ML language and mthodology. Theoretical technicalities are avoided whenever possible, with emphasis placed on the practical aspects of formal program development. An extended example of a complete program development in Extended ML is included.

[Sann99] Donald Sannella and Andrzej Tarlecki. Algebraic Methods for Specification and Formal Development of Programs. *ACM Computing Surveys*, 31, 1999.

[Scho16] Arthur Schopenhauer. Essays of Schopenhauer: On Reading and Books, 2016.

Link: `https://ebooks.adelaide.edu.au/s/schopenhauer/arthur/essays/chapter3.html`

[Stee90] Guy L. Steele. *Common Lisp, The Language*. Digital Equipment Corporation, 1990, 1-555558-041-6.

Link: `http://daly.axiom-developer.org/clm.pdf`

Algebra:
(p525) package DFSFUN DoubleFloatSpecialFunctions

[Thur94] William P. Thurston. On Proof and Progress in Mathematics. *Bulletin AMS*, 30(2), April 1994.

Link: `https://arxiv.org/pdf/math/9404236.pdf`

[Unkn20] Unknown. Programming is Terrible – Lessons Learned from a life wasted, 2020.

Link: `https://programmingisterrible.com/post/65781074112/devils-dictionary-of-programming`

[Warn16] Henrik Warne. More Good Programming Quotes, 2019.

Link: `https://henrikwarne.com/2016/04/17/more-good-programming-quotes`

[Warn19] Henrik Warne. More Good Programming Quotes, Part 3, 2019.

Link: `https://henrikwarne.com/2019/04/03/more-good-programming-quotes-part-3`

[Wiki19] Wikipedia. Algebraic data type, 2019.

Link: `https://en.wikipedia.org/wiki/Algebraic_data_type`

[Wood20] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. CMU PDF, 2020.

Link: `http://www.cs.cmu.edu/~15819/zedbook.pdf`

[Xena19] Kevin Buzzard. Xena, 2019.

Link: `https://xenaproject.wordpress.com`

[Zeil02] Doron Zeilberger. Topology: The Slum of Combinatorics, 2002.

Link: http://sites.math.rutgers.edu/~zeilberg/Opinion1.html

[verb20] unknown. Verbal Regular Expressions, 2020.

Link: https://github.com/VerbalExpressions/implementation/wiki/List-of-methods-to-implement

Signatures Index

59 compileSpad : theparse → values
57 explode : string → list char
51 FSM-abbrev : linelist → (linelist,abbreviation)
54 FSM-catBody : (linelist,theparse) → list(tokens)
54 FSM-cdpsig : (linelist,theparse) → (linelist,CDPSigClass)
52 FSM-comment : linelist → linelist,list(comments)
55 FSM-dpBody : (linelist,theparse) → nil
73 FSM-extract1Macro : linelist → (linelist,gather)
64 FSM-gatherList : linelist → (linelist,list(args)
66 FSM-gatherVars : argslist → list(vars)
54 FSM-isCategory? : (linelist,theparse) → (linelist,boolean)
53 FSM-macroNames : theparse → list(macronames)
52 FSM-macros : linelist → (linelist,macroAlist)
51 FSM-symbol : linelist → (linelist,symboliation)
55 FSM-VAR : linelist → (linelist,var)
56 FSM-variables : (linelist,theparse) → nil
72 FSMSIG : debug → values
35 getAncestors : symbol → list(ancestors)
67 indent : line → depth
57 joinsplit : (list(token),char) → string
35 leaves : survive → survive
30 make-hasclause : (type,condition,siglist) → hasclause
31 make-haslist : clauses → hasclause
50 make-instance : (class,abbrev,string) → abbreviation
40 make-operation : name → operation
36 make-signature : (funcname return arglist comment examples infix? → signature
47 make-Sourcecode : filename → sourcecode
42 operations-show : operationAlist → values
41 operationToSignature : alistEntry → signature
46 Parse : filename → (theparse,pile)
74 parseSignature : linelist → values
69 pile2tree : pileform → treeform
70 pretty : treeform → values
83 sane : nil → nil
68 spawn : tree → list(indentedLines)
57 splitchar : (string,char) → list(token)

Figure Index

==== **A** ====

ABBREV syntax, 50

==== **F** ====

FLOAT syntax, 58
FOR syntax, 58
FSMSIG syntax, 70

==== **I** ====

IF syntax, 58

==== **L** ====

LEAVE syntax, 71

==== **S** ====

SYMBOL syntax, 57

==== **T** ====

TYPESPEC syntax, 56

==== **U** ====

UABBREV syntax, 50
UNSIGNED syntax, 58

==== **V** ====

VAR syntax, 55

Documentation Index

==== **Symbols ====**

place, 45

==== **A ====**

assert
    keyword, 16

==== **B ====**

birth, 67

==== **C ====**

compileSpad, 59

==== **D ====**

deep, 69
depthof, 67

==== **E ====**

ensures
    keyword, 16
explode, 57

==== **F ====**

forall
    keyword, 16
FSM-abbrev, 50
FSM-cdpname, 63
FSM-cdpsig, 54
FSM-extract1Macro, 73
FSM-gatherList, 64
FSM-gatherVars, 65
FSM-macroNames, 53
FSM-macros, 52
FSM-variables, 56

==== **G ====**

gather, 73

==== **I ====**

indent, 67
intern-object, 21

==== **J ====**

joinsplit, 57

==== **K ====**

keyword
    assert, 16
    ensures, 16
    forall, 16
    requires, 16
    where, 16

==== **L ====**

labels, 22
leaves, 35
level, 35

==== **M ====**

make-Sourcecode, 47
make-sourcecode
    caller, 46

==== **N ====**

nameof, 68

==== **O ====**

OperationAlist, 36

==== **P** ====

ParserClass, 46
pretty, 70
print-object, 21
Product Types, 22

==== **R** ====

Record, 22
requires
    keyword, 16

==== **S** ====

showSig, 33
Signatures, 36
SigSort, 759
spawn, 68
splitchar, 57
Sum Types, 22

==== **T** ====

theparse, 46

==== **U** ====

Union, 22

==== **W** ====

where
    keyword, 16

Code Index

==== **Symbols ====**

Categories
    defvar, 29
Category, 29
Domains
    defvar, 29
Packages
    defvar, 29
defaultdomain-list
    defvar, 259
indent
    defvar, 30
operation-hash, 260
place
    defvar, 45

==== **A ====**

abbreviation
    defclass, 49
    defmethod
        print-object, 50
AddClass
    defclass, 33
amacro
    defclass, 39
AxiomClass
    Category, 22
    defclass, 31
    defmethod
        print-object, 32
    methods
        print-object, 32

==== **B ====**

birth
    defmacro, 67

==== **C ====**

Category
    AxiomClass, 22
CDPSigClass
    defclass, 53
    defmethod
        print-object, 53
compileSpad
    defun, 59

==== **D ====**

database
    defstruct, 259
defclass
    abbreviation, 49
    AddClass, 33
    amacro, 39
    AxiomClass, 31
    CDPSigClass, 53
    FSM-VAR, 56
    gather, 61
    hasclause, 30
    haslist, 31
    operation, 39
    ParserClass, 45
    signature, 36
    sourcecode, 49
    specification, 26
    WithClass, 33
defgeneric
    operation-add, 40
    showSig, 33
defmacro
    birth, 67
    defunt, 34
    depthof, 68
    expect, 83
    FSM-abbname, 64
    FSM-AND, 60
    FSM-cdpname, 64
    FSM-dword, 63
    FSM-gather, 61, 62
    FSM-match, 63

FSM-OR, 60
FSM-startsWith, 62
FSM-uword, 63
FSM-word, 63
ll, 35
must, 83
nameof, 68
trim, 34
defmethod
operation-add, 40
print-object
abbreviation, 50
AxiomClass, 32
CDPSigClass, 53
FSM-VAR, 56
gather, 61
hasclause, 31
operation, 40
ParserClass, 45
signature, 37
sourcecode, 49
showSig, 34
defstruct
database, 259
defun
compileSpad, 59
explode, 57
FSM-abbrev, 51
FSM-catBody, 55
FSM-cdpsig, 54
FSM-comment, 52
FSM-dpBody, 55
FSM-extract1Macro, 73
FSM-gatherList, 64
FSM-gatherVars, 66
FSM-isCategory?, 54
FSM-macroNames, 53
FSM-macros, 52
FSM-symbol, 51
FSM-VAR, 55
FSM-variables, 56
FSMSIG, 72
getAncestors, 35
getDomains, 36
indent, 67
joinsplit, 57
leaves, 35
make-abbreviation, 50
make-hasclause, 30
make-haslist, 31
make-operation, 40
make-signature, 37
make-Sourcecode, 47
operations-show, 42
operationToSignature, 41
Parse, 46
pile2tree, 69
pretty, 70
sane, 83
signatureToOperation, 43
SigSort, 759
spawn, 68
splitchar, 57
defunt
defmacro, 34
defvar
Categories, 29
Domains, 29
Packages, 29
defaultdomain-list, 259
indent, 30
place, 45
forcelongs, 42
infixOperations, 40
theparse, 46
depthof
defmacro, 68

==== **E** ====

expect
defmacro, 83
explode
defun, 57

==== **F** ====

forcelongs
defvar, 42
FSM
defun-variables, 56
FSM-abbname
defmacro, 64
FSM-abbrev
defun, 51
function, 49
FSM-AND
defmacro, 60
FSM-catBody
defun, 55
FSM-cdpname
defmacro, 64
FSM-cdpsig
defun, 54
FSM-comment
defun, 52
FSM-dpBody
defun, 55

FSM-dword
  defmacro, 63
FSM-extract1Macro
  defun, 73
FSM-gather
  defmacro, 61, 62
FSM-gatherList
  defun, 64
FSM-gatherVars
  defun, 66
FSM-isCategory?
  defun, 54
FSM-macroNames
  defun, 53
FSM-macros
  defun, 52
FSM-match
  defmacro, 63
FSM-OR
  defmacro, 60
FSM-startsWith
  defmacro, 62
FSM-symbol
  defun, 51
FSM-uword
  defmacro, 63
FSM-VAR
  defclass, 56
  defmethod
    print-object, 56
  defun, 55
FSM-word
  defmacro, 63
FSMSIG
  defun, 72
function
  FSM-abbrev, 49

==== **G** ====

gather
  defclass, 61
  defmethod
    print-object, 61
getAncestors
  defun, 35
getDomains
  defun, 36

==== **H** ====

hasclause
  defclass, 30
  defmethod
    print-object, 31
haslist
  defclass, 31

==== **I** ====

indent
  defun, 67
infixOperations
  defvar, 40

==== **J** ====

joinsplit
  defun, 57

==== **L** ====

leaves
  defun, 35
ll
  defmacro, 35

==== **M** ====

make-abbreviation
  defun, 50
make-hasclause
  defun, 30
make-haslist
  defun, 31
make-operation
  defun, 40
make-signature
  defun, 37
make-Sourcecode
  defun, 47
must
  defmacro, 83

==== **N** ====

nameof
  defmacro, 68

==== **O** ====

operation
  defclass, 39
  defmethod
    print-object, 40
operation-add
  defgeneric, 40
  defmethod, 40
operations-show

defun, 42
operationToSignature
defun, 41

==== P ====

Parse
defun, 46
parser
parseSignature, 74
ParserClass
defclass, 45
defmethod
print-object, 45
parseSignature
parser, 74
pile2tree, 68
defun, 69
pretty
defun, 70
print-object
AxiomClass, 32
defmethod
abbreviation, 50
AxiomClass, 32
CDPSigClass, 53
FSM-VAR, 56
gather, 61
hasclause, 31
operation, 40
ParserClass, 45
signature, 37
sourcecode, 49

==== R ====

result
variable, 60, 62

==== S ====

sane
defun, 83
showSig
defgeneric, 33
defmethod, 34
signature
defclass, 36
defmethod
print-object, 37
signatureToOperation
defun, 43
SigSort
defun, 759
sourcecode
defclass, 49
defmethod
print-object, 49
spawn
defun, 68
specification
defclass, 26
split
variable, 60, 62
splitchar
defun, 57

==== T ====

theparse
defvar, 46
trim
defmacro, 34

==== V ====

variable
result, 60, 62
split, 60, 62

==== W ====

WithClass
defclass, 33

Error Index

==== **F** ====

FSM-gatherList 1, 64

==== **P** ====

Parse 1, 46

Category Index

==== **A** ====

A1AGG, 193
ABELGRP, 182
AbelianGroup
    Category, 182
AbelianMonoid
    Category, 163
AbelianMonoidRing
    Category, 223
AbelianSemiGroup
    Category, 133
ABELMON, 163
ABELSG, 133
ACF, 239
ACFS, 248
AdditiveValuationAttribute
    Category, 90
AffineSpaceCategory
    Category, 164
AFSPCAT, 164
AGG, 123
Aggregate
    Category, 123
AHYP, 92
ALAGG, 211
ALGEBRA, 210
Algebra
    Category, 210
AlgebraicallyClosedField
    Category, 239
AlgebraicallyClosedFunctionSpace
    Category, 248
AMR, 223
ApproximateAttribute
    Category, 90
ArbitraryExponentAttribute
    Category, 91
ArbitraryPrecisionAttribute
    Category, 91
ArcHyperbolicFunctionCategory
    Category, 92
ArcTrigonometricFunctionCategory
    Category, 93
ARR2CAT, 173
AssociationListAggregate
    Category, 211
ATADDVA, 90
ATAPPRO, 90
ATARBEX, 91
ATARBPR, 91
ATCANCL, 103
ATCANON, 103
ATCENRL, 104
ATCS, 106
ATCUNOR, 104
ATFINAG, 108
ATJACID, 110
ATLR, 110
ATLUNIT, 111
ATMULVA, 113
ATNOTHR, 114
ATNULSQ, 114
ATNZDIV, 113
ATPOSET, 117
ATRIG, 93
ATRUNIT, 119
ATSHMUT, 120
ATTREG, 94
AttributeRegistry
    Category, 94
ATUNIKN, 122

==== **B** ====

BagAggregate
    Category, 164
BasicType
    Category, 95
BASTYPE, 95
BGAGG, 164
BiModule
    Category, 197

BinaryRecursiveAggregate
  Category, 174
BinaryTreeCategory
  Category, 183
BitAggregate
  Category, 197
BLMETCT, 134
BlowUpMethodCategory
  Category, 134
BMODULE, 197
BRAGG, 174
BTAGG, 197
BTCAT, 183

==== C ====

CABMON, 174
CachableSet
  Category, 165
CACHSET, 165
CancellationAbelianMonoid
  Category, 174
CanonicalAttribute
  Category, 103
CanonicalClosedAttribute
  Category, 103
CanonicalUnitNormalAttribute
  Category, 104
Category
  AbelianGroup, 182
  AbelianMonoid, 163
  AbelianMonoidRing, 223
  AbelianSemiGroup, 133
  AdditiveValuationAttribute, 90
  AffineSpaceCategory, 164
  Aggregate, 123
  Algebra, 210
  AlgebraicallyClosedField, 239
  AlgebraicallyClosedFunctionSpace, 248
  ApproximateAttribute, 90
  ArbitraryExponentAttribute, 91
  ArbitraryPrecisionAttribute, 91
  ArcHyperbolicFunctionCategory, 92
  ArcTrigonometricFunctionCategory, 93
  AssociationListAggregate, 211
  AttributeRegistry, 94
  BagAggregate, 164
  BasicType, 95
  BiModule, 197
  BinaryRecursiveAggregate, 174
  BinaryTreeCategory, 183
  BitAggregate, 197
  BlowUpMethodCategory, 134
  CachableSet, 165
  CancellationAbelianMonoid, 174
  CanonicalAttribute, 103
  CanonicalClosedAttribute, 103
  CanonicalUnitNormalAttribute, 104
  CentralAttribute, 104
  CharacteristicNonZero, 203
  CharacteristicZero, 203
  CoercibleTo, 105
  Collection, 165
  CombinatorialFunctionCategory, 105
  CombinatorialOpsCategory, 124
  CommutativeRing, 204
  CommutativeStarAttribute, 106
  Comparable, 135
  ComplexCategory, 255
  ConvertibleTo, 106
  DequeueAggregate, 183
  DesingTreeCategory, 175
  Dictionary, 189
  DictionaryOperations, 184
  DifferentialExtension, 211
  DifferentialPolynomialCategory, 240
  DifferentialRing, 204
  DifferentialVariableCategory, 166
  DirectProductCategory, 216
  DivisionRing, 217
  DivisorCategory, 212
  DoublyLinkedAggregate, 175
  ElementaryFunctionCategory, 107
  Eltable, 108
  EltableAggregate, 125
  EntireRing, 205
  EuclideanDomain, 233
  Evalable, 126
  ExpressionSpace, 167
  ExtensibleLinearAggregate, 184
  ExtensionField, 248
  Field, 236
  FieldOfPrimeCharacteristic, 241
  FileCategory, 135
  FileNameCategory, 136
  Finite, 137
  FiniteAbelianMonoidRing, 230
  FiniteAggregateAttribute, 108
  FiniteAlgebraicExtensionField, 251
  FiniteDivisorCategory, 189
  FiniteFieldCategory, 249
  FiniteLinearAggregate, 185
  FiniteRankAlgebra, 217
  FiniteRankNonAssociativeAlgebra, 218
  FiniteSetAggregate, 198
  FloatingPointSystem, 249
  FortranFunctionCategory, 137
  FortranMachineTypeCategory, 223

FortranMatrixCategory, 139
FortranMatrixFunctionCategory, 140
FortranProgramCategory, 126
FortranVectorCategory, 141
FortranVectorFunctionCategory, 142
FramedAlgebra, 224
FramedNonAssociativeAlgebra, 224
FreeAbelianMonoidCategory, 186
FreeLieAlgebra, 218
FreeModuleCat, 212
FullyEvalableOver, 143
FullyLinearlyExplicitRingOver, 213
FullyPatternMatchable, 167
FullyRetractableTo, 127
FunctionFieldCategory, 255
FunctionSpace, 241
GcdDomain, 225
GradedAlgebra, 168
GradedModule, 144
Group, 176
HomogeneousAggregate, 145
HyperbolicFunctionCategory, 109
IndexedAggregate, 168
IndexedDirectProductCategory, 145
InfinitlyClosePointCategory, 169
InnerEvalable, 109
IntegerNumberSystem, 236
IntegralDomain, 219
IntervalCategory, 230
JacobiIdentityAttribute, 110
KeyedDictionary, 198
LazyRepresentationAttribute, 110
LazyStreamAggregate, 190
LeftAlgebra, 205
LeftModule, 190
LeftOreRing, 213
LeftUnitaryAttribute, 111
LieAlgebra, 214
LinearAggregate, 177
LinearlyExplicitRingOver, 206
LinearOrdinaryDifferentialOperator-
Category,
227
LiouvillianFunctionCategory, 146
ListAggregate, 191
LocalPowerSeriesCategory, 242
Logic, 128
MatrixCategory, 177
ModularAlgebraicGcdOperations, 111
Module, 206
Monad, 147
MonadWithUnit, 169
MonogenicAlgebra, 252
MonogenicLinearOperator, 219
Monoid, 170
MultiDictionary, 191
MultiplicativeValuationAttribute, 113
MultisetAggregate, 192
MultivariateTaylorSeriesCategory, 234
NonAssociativeAlgebra, 214
NonAssociativeRing, 199
NonAssociativeRng, 192
NormalizedTriangularSetCategory, 199
NotherianAttribute, 114
NoZeroDivisorsAttribute, 113
NullSquareAttribute, 114
NumericalIntegrationCategory, 148
NumericalOptimizationCategory, 148
OctonionCategory, 220
OneDimensionalArrayAggregate, 193
OpenMath, 115
OrderedAbelianGroup, 194
OrderedAbelianMonoid, 178
OrderedAbelianMonoidSup, 194
OrderedAbelianSemiGroup, 170
OrderedCancellationAbelianMonoid,
186
OrderedFinite, 171
OrderedIntegralDomain, 228
OrderedMonoid, 178
OrderedMultisetAggregate, 200
OrderedRing, 207
OrderedSet, 149
OrdinaryDifferentialEquationsSolver-
Category,
150
PAdicIntegerCategory, 237
PartialDifferentialEquationsSolverCate-
gory,
150
PartialDifferentialRing, 207
PartiallyOrderedSetAttribute, 117
PartialTranscendentalFunctions, 115
Patternable, 128
PatternMatchable, 151
PermutationCategory, 187
PlacesCategory, 171
PlottablePlaneCurveCategory, 129
PlottableSpaceCurveCategory, 129
PointCategory, 208
PolynomialCategory, 238
PolynomialFactorizationExplicit, 234
PolynomialSetCategory, 179
PowerSeriesCategory, 231
PrimitiveFunctionCategory, 117
PrincipalIdealDomain, 231
PriorityQueueAggregate, 180
ProjectiveSpaceCategory, 172

PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory, 256
PseudoAlgebraicClosureOfFiniteFieldCategory, 253
PseudoAlgebraicClosureOfPerfectFieldCategory, 242
PseudoAlgebraicClosureOfRationalNumberCategory, 254
QuaternionCategory, 228
QueueAggregate, 180
QuotientFieldCategory, 243
RadicalCategory, 118
RealClosedField, 244
RealConstant, 130
RealNumberSystem, 245
RealRootCharacterizationCategory, 152
RectangularMatrixCategory, 215
RecursiveAggregate, 172
RecursivePolynomialCategory, 245
RegularTriangularSetCategory, 195
RetractableTo, 118
RightModule, 196
RightUnitaryAttribute, 119
Ring, 201
Rng, 196
SegmentCategory, 131
SegmentExpansionCategory, 154
SemiGroup, 155
SetAggregate, 181
SetCategory, 132
SetCategoryWithDegree, 156
SExpressionCategory, 153
ShallowlyMutableAttribute, 120
SpecialFunctionCategory, 120
SquareFreeNormalizedTriangularSetCategory, 208
SquareFreeRegularTriangularSetCategory, 201
SquareMatrixCategory, 221
StackAggregate, 181
StepThrough, 156
StreamAggregate, 187
StringAggregate, 202
StringCategory, 209
TableAggregate, 209
ThreeSpaceCategory, 157
TranscendentalFunctionCategory, 132
TriangularSetCategory, 188
TrigonometricFunctionCategory, 121
TwoDimensionalArrayCategory, 173
Type, 122
UnaryRecursiveAggregate, 182
UniqueFactorizationDomain, 232
UnitsKnownAttribute, 122
UnivariateLaurentSeriesCategory, 246
UnivariateLaurentSeriesConstructorCategory, 250
UnivariatePolynomialCategory, 246
UnivariatePowerSeriesCategory, 235
UnivariatePuiseuxSeriesCategory, 247
UnivariatePuiseuxSeriesConstructorCategory, 251
UnivariateSkewPolynomialCategory, 221
UnivariateTaylorSeriesCategory, 238
VectorCategory, 202
VectorSpace, 215
XAlgebra, 222
XFreeAlgebra, 229
XPolynomialsCat, 232
CentralAttribute
  Category, 104
CFCAT, 105
CharacteristicNonZero
  Category, 203
CharacteristicZero
  Category, 203
CHARNZ, 203
CHARZ, 203
CLAGG, 165
CoercibleTo
  Category, 105
Collection
  Category, 165
CombinatorialFunctionCategory
  Category, 105
CombinatorialOpsCategory
  Category, 124
COMBOPC, 124
CommutativeRing
  Category, 204
CommutativeStarAttribute
  Category, 106
COMPAR, 135
Comparable
  Category, 135
COMPCAT, 255
ComplexCategory
  Category, 255

COMRING, 204
ConvertibleTo
    Category, 106

==== **D** ====

DequeueAggregate
    Category, 183
DesingTreeCategory
    Category, 175
DIAGG, 189
Dictionary
    Category, 189
DictionaryOperations
    Category, 184
DIFEXT, 211
DifferentialExtension
    Category, 211
DifferentialPolynomialCategory
    Category, 240
DifferentialRing
    Category, 204
DifferentialVariableCategory
    Category, 166
DIFRING, 204
DIOPS, 184
DirectProductCategory
    Category, 216
DIRPCAT, 216
DIVCAT, 212
DivisionRing
    Category, 217
DivisorCategory
    Category, 212
DIVRING, 217
DLAGG, 175
DoublyLinkedAggregate
    Category, 175
DPOLCAT, 240
DQAGG, 183
DSTRCAT, 175
DVARCAT, 166

==== **E** ====

ELAGG, 184
ElementaryFunctionCategory
    Category, 107
ELEMFUN, 107
ELTAB, 108
Eltable
    Category, 108
EltableAggregate
    Category, 125
ELTAGG, 125
ENTIRER, 205
EntireRing
    Category, 205
ES, 167
EUCDOM, 233
EuclideanDomain
    Category, 233
EVALAB, 126
Evalable
    Category, 126
ExpressionSpace
    Category, 167
ExtensibleLinearAggregate
    Category, 184
ExtensionField
    Category, 248

==== **F** ====

FAMONC, 186
FAMR, 230
FAXF, 251
FDIVCAT, 189
FEVALAB, 143
FFCAT, 255
FFIELDC, 249
FIELD, 236
Field
    Category, 236
FieldOfPrimeCharacteristic
    Category, 241
FILECAT, 135
FileCategory
    Category, 135
FileNameCategory
    Category, 136
FINAALG, 218
FINITE, 137
Finite
    Category, 137
FiniteAbelianMonoidRing
    Category, 230
FiniteAggregateAttribute
    Category, 108
FiniteAlgebraicExtensionField
    Category, 251
FiniteDivisorCategory
    Category, 189
FiniteFieldCategory
    Category, 249
FiniteLinearAggregate
    Category, 185
FiniteRankAlgebra

Category, 217
FiniteRankNonAssociativeAlgebra
Category, 218
FiniteSetAggregate
Category, 198
FINRALG, 217
FLAGG, 185
FLALG, 218
FLINEXP, 213
FloatingPointSystem
Category, 249
FMC, 139
FMCAT, 212
FMFUN, 140
FMTC, 223
FNCAT, 136
FORTCAT, 126
FORTFN, 137
FortranFunctionCategory
Category, 137
FortranMachineTypeCategory
Category, 223
FortranMatrixCategory
Category, 139
FortranMatrixFunctionCategory
Category, 140
FortranProgramCategory
Category, 126
FortranVectorCategory
Category, 141
FortranVectorFunctionCategory
Category, 142
FPATMAB, 167
FPC, 241
FPS, 249
FRAMALG, 224
FramedAlgebra
Category, 224
FramedNonAssociativeAlgebra
Category, 224
FreeAbelianMonoidCategory
Category, 186
FreeLieAlgebra
Category, 218
FreeModuleCat
Category, 212
FRETRCT, 127
FRNAALG, 224
FS, 241
FSAGG, 198
FullyEvalableOver
Category, 143
FullyLinearlyExplicitRingOver
Category, 213
FullyPatternMatchable
Category, 167
FullyRetractableTo
Category, 127
FunctionFieldCategory
Category, 255
FunctionSpace
Category, 241
FVC, 141
FVFUN, 142

==== **G** ====

GCDDOM, 225
GcdDomain
Category, 225
GradedAlgebra
Category, 168
GradedModule
Category, 144
GRALG, 168
GRMOD, 144
GROUP, 176
Group
Category, 176

==== **H** ====

HOAGG, 145
HomogeneousAggregate
Category, 145
HYPCAT, 109
HyperbolicFunctionCategory
Category, 109

==== **I** ====

IDPC, 145
IEVALAB, 109
IndexedAggregate
Category, 168
IndexedDirectProductCategory
Category, 145
INFCLCT, 169
InfinitlyClosePointCategory
Category, 169
InnerEvalable
Category, 109
INS, 236
INTCAT, 230
INTDOM, 219
IntegerNumberSystem
Category, 236
IntegralDomain
Category, 219

IntervalCategory
Category, 230
IXAGG, 168

==== J ====

JacobiIdentityAttribute
Category, 110

==== K ====

KDAGG, 198
KeyedDictionary
Category, 198
KOERCE, 105
KONVERT, 106

==== L ====

LALG, 205
LazyRepresentationAttribute
Category, 110
LazyStreamAggregate
Category, 190
LeftAlgebra
Category, 205
LeftModule
Category, 190
LeftOreRing
Category, 213
LeftUnitaryAttribute
Category, 111
LFCAT, 146
LieAlgebra
Category, 214
LIECAT, 214
LinearAggregate
Category, 177
LinearlyExplicitRingOver
Category, 206
LinearOrdinaryDifferentialOperatorCategory
Category, 227
LINEXP, 206
LiouvillianFunctionCategory
Category, 146
ListAggregate
Category, 191
LMODULE, 190
LNAGG, 177
LocalPowerSeriesCategory
Category, 242
LOCPOWC, 242
LODOCAT, 227
LOGIC, 128
Logic
Category, 128
LORER, 213
LSAGG, 191
LZSTAGG, 190

==== M ====

MAGCDOC, 111
MATCAT, 177
MatrixCategory
Category, 177
MDAGG, 191
MLO, 219
ModularAlgebraicGcdOperations
Category, 111
MODULE, 206
Module
Category, 206
MONAD, 147
Monad
Category, 147
MonadWithUnit
Category, 169
MONADWU, 169
MONOGEN, 252
MonogenicAlgebra
Category, 252
MonogenicLinearOperator
Category, 219
MONOID, 170
Monoid
Category, 170
MSETAGG, 192
MTSCAT, 234
MultiDictionary
Category, 191
MultiplicativeValuationAttribute
Category, 113
MultisetAggregate
Category, 192
MultivariateTaylorSeriesCategory
Category, 234

==== N ====

NAALG, 214
NARNG, 192
NASRING, 199
NonAssociativeAlgebra
Category, 214
NonAssociativeRing
Category, 199
NonAssociativeRng
Category, 192

NormalizedTriangularSetCategory
    Category, 199
NotherianAttribute
    Category, 114
NoZeroDivisorsAttribute
    Category, 113
NTSCAT, 199
NullSquareAttribute
    Category, 114
NumericalIntegrationCategory
    Category, 148
NumericalOptimizationCategory
    Category, 148
NUMINT, 148

==== **O** ====

OAGROUP, 194
OAMON, 178
OAMONS, 194
OASGP, 170
OC, 220
OCAMON, 186
OctonionCategory
    Category, 220
ODECAT, 150
OINTDOM, 228
OM, 115
OMSAGG, 200
OneDimensionalArrayAggregate
    Category, 193
OpenMath
    Category, 115
OPTCAT, 148
OrderedAbelianGroup
    Category, 194
OrderedAbelianMonoid
    Category, 178
OrderedAbelianMonoidSup
    Category, 194
OrderedAbelianSemiGroup
    Category, 170
OrderedCancellationAbelianMonoid
    Category, 186
OrderedFinite
    Category, 171
OrderedIntegralDomain
    Category, 228
OrderedMonoid
    Category, 178
OrderedMultisetAggregate
    Category, 200
OrderedRing
    Category, 207
OrderedSet
    Category, 149
ORDFIN, 171
OrdinaryDifferentialEquationsSolverCategory
    Category, 150
ORDMON, 178
ORDRING, 207
ORDSET, 149
OREPCAT, 221

==== **P** ====

PACEXTC, 256
PACFFC, 253
PACPERC, 242
PACRATC, 254
PADICCT, 237
PAdicIntegerCategory
    Category, 237
PartialDifferentialEquationsSolverCategory
    Category, 150
PartialDifferentialRing
    Category, 207
PartiallyOrderedSetAttribute
    Category, 117
PartialTranscendentalFunctions
    Category, 115
PATAB, 128
PATMAB, 151
Patternable
    Category, 128
PatternMatchable
    Category, 151
PDECAT, 150
PDRING, 207
PERMCAT, 187
PermutationCategory
    Category, 187
PFECAT, 234
PID, 231
PLACESC, 171
PlacesCategory
    Category, 171
PlottablePlaneCurveCategory
    Category, 129
PlottableSpaceCurveCategory
    Category, 129
PointCategory
    Category, 208
POLYCAT, 238
PolynomialCategory
    Category, 238
PolynomialFactorizationExplicit
    Category, 234

PolynomialSetCategory
Category, 179
PowerSeriesCategory
Category, 231
PPCURVE, 129
PRIMCAT, 117
PrimitiveFunctionCategory
Category, 117
PrincipalIdealDomain
Category, 231
PriorityQueueAggregate
Category, 180
ProjectiveSpaceCategory
Category, 172
PRQAGG, 180
PRSPCAT, 172
PSCAT, 231
PSCURVE, 129
PSETCAT, 179
PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory
Category, 256
PseudoAlgebraicClosureOfFiniteFieldCategory
Category, 253
PseudoAlgebraicClosureOfPerfectFieldCategory
Category, 242
PseudoAlgebraicClosureOfRationalNumberCategory
Category, 254
PTCAT, 208
PTRANFN, 115

==== **Q** ====

QFCAT, 243
QUAGG, 180
QUATCAT, 228
QuaternionCategory
Category, 228
QueueAggregate
Category, 180
QuotientFieldCategory
Category, 243

==== **R** ====

RADCAT, 118
RadicalCategory
Category, 118
RCAGG, 172
RCFIELD, 244
REAL, 130
RealClosedField
Category, 244
RealConstant
Category, 130
RealNumberSystem
Category, 245
RealRootCharacterizationCategory
Category, 152
RectangularMatrixCategory
Category, 215
RecursiveAggregate
Category, 172
RecursivePolynomialCategory
Category, 245
RegularTriangularSetCategory
Category, 195
RETRACT, 118
RetractableTo
Category, 118
RightModule
Category, 196
RightUnitaryAttribute
Category, 119
RING, 201
Ring
Category, 201
RMATCAT, 215
RMODULE, 196
RNG, 196
Rng
Category, 196
RNS, 245
RPOLCAT, 245
RRCC, 152
RSETCAT, 195

==== **S** ====

SEGCAT, 131
SegmentCategory
Category, 131
SegmentExpansionCategory
Category, 154
SEGXCAT, 154
SemiGroup
Category, 155
SETAGG, 181
SetAggregate
Category, 181
SETCAT, 132
SETCATD, 156
SetCategory
Category, 132
SetCategoryWithDegree
Category, 156
SEXCAT, 153
SExpressionCategory
Category, 153

SFRTCAT, 201
SGROUP, 155
ShallowlyMutableAttribute
    Category, 120
SKAGG, 181
SMATCAT, 221
SNTSCAT, 208
SPACEC, 157
SpecialFunctionCategory
    Category, 120
SPFCAT, 120
SquareFreeNormalizedTriangularSetCategory
    Category, 208
SquareFreeRegularTriangularSetCategory
    Category, 201
SquareMatrixCategory
    Category, 221
SRAGG, 202
StackAggregate
    Category, 181
STAGG, 187
STEP, 156
StepThrough
    Category, 156
StreamAggregate
    Category, 187
STRICAT, 209
StringAggregate
    Category, 202
StringCategory
    Category, 209

==== **T** ====

TableAggregate
    Category, 209
TBAGG, 209
ThreeSpaceCategory
    Category, 157
TRANFUN, 132
TranscendentalFunctionCategory
    Category, 132
TriangularSetCategory
    Category, 188
TRIGCAT, 121
TrigonometricFunctionCategory
    Category, 121
TSETCAT, 188
TwoDimensionalArrayCategory
    Category, 173
TYPE, 122
Type
    Category, 122

==== **U** ====

UFD, 232
ULSCAT, 246
ULSCCAT, 250
UnaryRecursiveAggregate
    Category, 182
UniqueFactorizationDomain
    Category, 232
UnitsKnownAttribute
    Category, 122
UnivariateLaurentSeriesCategory
    Category, 246
UnivariateLaurentSeriesConstructorCategory
    Category, 250
UnivariatePolynomialCategory
    Category, 246
UnivariatePowerSeriesCategory
    Category, 235
UnivariatePuiseuxSeriesCategory
    Category, 247
UnivariatePuiseuxSeriesConstructorCategory
    Category, 251
UnivariateSkewPolynomialCategory
    Category, 221
UnivariateTaylorSeriesCategory
    Category, 238
UPOLYC, 246
UPSCAT, 235
UPXSCAT, 247
UPXSCCA, 251
URAGG, 182
UTSCAT, 238

==== **V** ====

VECTCAT, 202
VectorCategory
    Category, 202
VectorSpace
    Category, 215
VSPACE, 215

==== **X** ====

XALG, 222
XAlgebra
    Category, 222
XF, 248
XFALG, 229
XFreeAlgebra
    Category, 229
XPOLYC, 232
XPolynomialsCat
    Category, 232

Domain Index

==== **A** ====

ACPLOT, 430
AffinePlane
    Domain, 265
AffinePlaneOverPseudoAlgebraicClosureOfFiniteField
    Domain, 265
AffineSpace
    Domain, 266
AFFPL, 265
AFFPLPS, 265
AFFSP, 266
AlgebraGivenByStructuralConstants
    Domain, 266
AlgebraicFunctionField
    Domain, 267
AlgebraicNumber
    Domain, 267
ALGFF, 267
ALGSC, 266
ALIST, 297
AN, 267
ANON, 268
AnonymousFunction
    Domain, 268
ANTISYM, 268
AntiSymm
    Domain, 268
ANY, 269
Any
    Domain, 269
ARRAY1, 412
ARRAY2, 479
ArrayStack
    Domain, 269
Asp1, 270
    Domain, 270
ASP10, 270
Asp10
    Domain, 270
ASP12, 271
Asp12
    Domain, 271
ASP19, 272
Asp19
    Domain, 272
ASP20, 274
Asp20
    Domain, 274
ASP24, 275
Asp24
    Domain, 275
ASP27, 275
Asp27
    Domain, 275
ASP28, 276
Asp28
    Domain, 276
ASP29, 279
Asp29
    Domain, 279
ASP30, 279
Asp30
    Domain, 279
ASP31, 280
Asp31
    Domain, 280
ASP33, 281
Asp33
    Domain, 281
ASP34, 282
Asp34
    Domain, 282
ASP35, 283
Asp35
    Domain, 283
ASP4, 283
Asp4
    Domain, 283
ASP41, 284
Asp41
    Domain, 284

ASP42, 285
Asp42
    Domain, 285
ASP49, 286
Asp49
    Domain, 286
ASP50, 287
Asp50
    Domain, 287
ASP55, 288
Asp55
    Domain, 288
ASP6, 289
Asp6
    Domain, 289
ASP7, 290
Asp7
    Domain, 290
ASP73, 291
Asp73
    Domain, 291
ASp74, 291
Asp74
    Domain, 291
ASP77, 292
Asp77
    Domain, 292
ASP78, 293
Asp78
    Domain, 293
ASP8, 293
Asp8
    Domain, 293
ASP80, 294
Asp80
    Domain, 294
ASP9, 295
Asp9
    Domain, 295
AssociatedJordanAlgebra
    Domain, 296
AssociatedLieAlgebra
    Domain, 296
AssociationList
    Domain, 297
ASTACK, 269
ATTRBUT, 298
AttributeButtons
    Domain, 298
AUTOMOR, 299
Automorphism
    Domain, 299

==== **B** ====

BalancedBinaryTree
    Domain, 299
BalancedPAdicInteger
    Domain, 300
BalancedPAdicRational
    Domain, 300
BasicFunctions
    Domain, 301
BasicOperator
    Domain, 301
BasicStochasticDifferential
    Domain, 302
BBTREE, 299
BFUNCT, 301
BINARY, 302
BinaryExpansion
    Domain, 302
BinaryFile
    Domain, 303
BinarySearchTree
    Domain, 303
BinaryTournament
    Domain, 304
BinaryTree
    Domain, 304
BINFILE, 303
BITS, 305
Bits
    Domain, 305
BLHN, 305
BlowUpWithHamburgerNoether
    Domain, 305
BlowUpWithQuadTrans
    Domain, 305
BLQT, 305
BOOLEAN, 306
Boolean
    Domain, 306
BOP, 301
BPADIC, 300
BPADICRT, 300
BSD, 302
BSTREE, 303
BTOURN, 304
BTREE, 304

==== **C** ====

CARD, 306
CardinalNumber
    Domain, 306
CARTEN, 307
CartesianTensor

Domain, 307
CCLASS, 309
CDFMAT, 311
CDFVEC, 312
CELL, 308
Cell
Domain, 308
CHAR, 308
Character
Domain, 308
CharacterClass
Domain, 309
CLIF, 309
CliffordAlgebra
Domain, 309
COLOR, 310
Color
Domain, 310
COMM, 310
Commutator
Domain, 310
COMPLEX, 311
Complex
Domain, 311
ComplexDoubleFloatMatrix
Domain, 311
ComplexDoubleFloatVector
Domain, 312
COMPPROP, 470
CONTFRAC, 312
ContinuedFraction
Domain, 312

==== **D** ====

D01AJFA, 323
d01ajfAnnaType
Domain, 323
D01AKFA, 323
d01akfAnnaType
Domain, 323
D01ALFA, 324
d01alfAnnaType
Domain, 324
D01AMFA, 324
d01amfAnnaType
Domain, 324
D01ANFA, 325
d01anfAnnaType
Domain, 325
D01APFA, 325
d01apfAnnaType
Domain, 325
D01AQFA, 326
d01aqfAnnaType
Domain, 326
D01ASFA, 327
d01asfAnnaType
Domain, 327
D01FCFA, 327
d01fcfAnnaType
Domain, 327
D01GBFA, 328
d01gbfAnnaType
Domain, 328
d01TransformFunctionType
Domain, 328
D01TRNS, 328
D02BBFA, 329
d02bbfAnnaType
Domain, 329
D02BHFA, 329
d02bhfAnnaType
Domain, 329
D02CJFA, 330
d02cjfAnnaType
Domain, 330
D02EJFA, 331
d02ejfAnnaType
Domain, 331
D03EEFA, 331
d03eefAnnaType
Domain, 331
D03FAFA, 332
d03fafAnnaType
Domain, 332
Database
Domain, 313
DataList
Domain, 313
DBASE, 313
DECIMAL, 314
DecimalExpansion
Domain, 314
DenavitHartenbergMatrix
Domain, 314
DEQUEUE, 315
Dequeue
Domain, 315
DERHAM, 316
DeRhamComplex
Domain, 316
DesingTree
Domain, 316
DFLOAT, 320
DFMAT, 321
DFVEC, 322
DHMATRIX, 314

DifferentialSparseMultivariatePolynomial
  Domain, 317
DirectProduct
  Domain, 317
DirectProductMatrixModule
  Domain, 318
DirectProductModule
  Domain, 318
DirichletRing
  Domain, 319
DIRPROD, 317
DIRRING, 319
DistributedMultivariatePolynomial
  Domain, 319
DIV, 320
Divisor
  Domain, 320
DLIST, 313
DMP, 319
Domain
  AffinePlane, 265
  AffinePlaneOverPseudoAlgebraicClosureOfFiniteField, 265
  AffineSpace, 266
  AlgebraGivenByStructuralConstants, 266
  AlgebraicFunctionField, 267
  AlgebraicNumber, 267
  AnonymousFunction, 268
  AntiSymm, 268
  Any, 269
  ArrayStack, 269
  Asp1, 270
  Asp10, 270
  Asp12, 271
  Asp19, 272
  Asp20, 274
  Asp24, 275
  Asp27, 275
  Asp28, 276
  Asp29, 279
  Asp30, 279
  Asp31, 280
  Asp33, 281
  Asp34, 282
  Asp35, 283
  Asp4, 283
  Asp41, 284
  Asp42, 285
  Asp49, 286
  Asp50, 287
  Asp55, 288
  Asp6, 289
  Asp7, 290
  Asp73, 291
  Asp74, 291
  Asp77, 292
  Asp78, 293
  Asp8, 293
  Asp80, 294
  Asp9, 295
  AssociatedJordanAlgebra, 296
  AssociatedLieAlgebra, 296
  AssociationList, 297
  AttributeButtons, 298
  Automorphism, 299
  BalancedBinaryTree, 299
  BalancedPAdicInteger, 300
  BalancedPAdicRational, 300
  BasicFunctions, 301
  BasicOperator, 301
  BasicStochasticDifferential, 302
  BinaryExpansion, 302
  BinaryFile, 303
  BinarySearchTree, 303
  BinaryTournament, 304
  BinaryTree, 304
  Bits, 305
  BlowUpWithHamburgerNoether, 305
  BlowUpWithQuadTrans, 305
  Boolean, 306
  CardinalNumber, 306
  CartesianTensor, 307
  Cell, 308
  Character, 308
  CharacterClass, 309
  CliffordAlgebra, 309
  Color, 310
  Commutator, 310
  Complex, 311
  ComplexDoubleFloatMatrix, 311
  ComplexDoubleFloatVector, 312
  ContinuedFraction, 312
  d01ajfAnnaType, 323
  d01akfAnnaType, 323
  d01alfAnnaType, 324
  d01amfAnnaType, 324
  d01anfAnnaType, 325
  d01apfAnnaType, 325
  d01aqfAnnaType, 326
  d01asfAnnaType, 327
  d01fcfAnnaType, 327
  d01gbfAnnaType, 328
  d01TransformFunctionType, 328
  d02bbfAnnaType, 329
  d02bhfAnnaType, 329
  d02cjfAnnaType, 330

d02ejfAnnaType, 331
d03eefAnnaType, 331
d03fafAnnaType, 332
Database, 313
DataList, 313
DecimalExpansion, 314
DenavitHartenbergMatrix, 314
Dequeue, 315
DeRhamComplex, 316
DesingTree, 316
DifferentialSparseMultivariatePolynomial,
317
DirectProduct, 317
DirectProductMatrixModule, 318
DirectProductModule, 318
DirichletRing, 319
DistributedMultivariatePolynomial,
319
Divisor, 320
DoubleFloat, 320
DoubleFloatMatrix, 321
DoubleFloatVector, 322
DrawOption, 322
e04dgfAnnaType, 338
e04fdfAnnaType, 338
e04gcfAnnaType, 339
e04jafAnnaType, 340
e04mbfAnnaType, 340
e04nafAnnaType, 341
e04ucfAnnaType, 341
ElementaryFunctionsUnivariateLaurentSeries,
332
ElementaryFunctionsUnivariatePuiseuxSeries,
333
EqTable, 334
Equation, 333
EuclideanModularRing, 334
Exit, 335
ExponentialExpansion, 335
ExponentialOfUnivariatePuiseuxSeries,
337
Expression, 336
ExtAlgBasis, 337
Factored, 342
File, 343
FileName, 343
FiniteDivisor, 344
FiniteField, 344
FiniteFieldCyclicGroup, 345
FiniteFieldCyclicGroupExtension, 345
FiniteFieldCyclicGroupExtensionByPolynomial,
346
FiniteFieldExtension, 346
FiniteFieldExtensionByPolynomial, 347
FiniteFieldNormalBasis, 347
FiniteFieldNormalBasisExtension, 348
FiniteFieldNormalBasisExtensionByPolynomial,
349
FlexibleArray, 349
Float, 350
FortranCode, 351
FortranExpression, 352
FortranProgram, 352
FortranScalarType, 353
FortranTemplate, 353
FortranType, 354
FourierComponent, 354
FourierSeries, 355
Fraction, 355
FractionalIdeal, 356
FramedModule, 356
FreeAbelianGroup, 357
FreeAbelianMonoid, 357
FreeGroup, 358
FreeModule, 358
FreeModule1, 359
FreeMonoid, 359
FreeNilpotentLie, 360
FullPartialFractionExpansion, 360
FunctionCalled, 361
GeneralDistributedMultivariatePolynomial,
361
GeneralModulePolynomial, 362
GeneralPolynomialSet, 363
GeneralSparseTable, 363
GeneralTriangularSet, 364
GeneralUnivariatePowerSeries, 364
GenericNonAssociativeAlgebra, 362
GraphImage, 365
GuessOption, 365
GuessOptionFunctions0, 366
HashTable, 366
Heap, 367
HexadecimalExpansion, 367
HomogeneousDirectProduct, 368
HomogeneousDistributedMultivariatePolynomial,
369
HTMLFormat, 368
HyperellipticFiniteDivisor, 369
IndexCard, 370

IndexedBits, 371
IndexedDirectProductAbelianGroup, 371
IndexedDirectProductAbelianMonoid, 372
IndexedDirectProductObject, 372
IndexedDirectProductOrderedAbelian-Monoid, 373
IndexedDirectProductOrderedAbelian-MonoidSup, 373
IndexedExponents, 374
IndexedFlexibleArray, 374
IndexedList, 375
IndexedMatrix, 376
IndexedOneDimensionalArray, 376
IndexedString, 377
IndexedTwoDimensionalArray, 377
IndexedVector, 378
InfClsPt, 370
InfiniteTuple, 378
InfinitlyClosePoint, 379
InfinitlyClosePointOverPseudoAlge-braicClosureOfFiniteField, 379
InnerAlgebraicNumber, 380
InnerFiniteField, 380
InnerFreeAbelianMonoid, 381
InnerIndexedTwoDimensionalArray, 381
InnerPAdicInteger, 382
InnerPrimeField, 382
InnerSparseUnivariatePowerSeries, 383
InnerTable, 383
InnerTaylorSeries, 384
InputForm, 384
Integer, 385
IntegerMod, 385
IntegrationFunctionsTable, 386
IntegrationResult, 386
Interval, 387
Kernel, 387
KeyedAccessFile, 388
LaurentPolynomial, 388
Library, 389
LieExponentials, 389
LiePolynomial, 390
LieSquareMatrix, 390
LinearOrdinaryDifferentialOperator, 391
LinearOrdinaryDifferentialOperator1, 392
LinearOrdinaryDifferentialOperator2, 392
List, 393
ListMonoidOps, 393
ListMultiDictionary, 394
LocalAlgebra, 394
Localize, 395
LyndonWord, 395
MachineComplex, 396
MachineFloat, 396
MachineInteger, 397
Magma, 397
MakeCachableSet, 398
MathMLFormat, 398
Matrix, 399
ModMonic, 399
ModularField, 400
ModularRing, 400
ModuleMonomial, 401
ModuleOperator, 401
MoebiusTransform, 402
MonoidRing, 402
Multiset, 403
MultivariatePolynomial, 403
MyExpression, 404
MyUnivariatePolynomial, 404
NeitherSparseOrDensePowerSeries, 405
NewSparseMultivariatePolynomial, 405
NewSparseUnivariatePolynomial, 406
None, 406
NonNegativeInteger, 407
NottinghamGroup, 407
NumericalIntegrationProblem, 408
NumericalODEProblem, 409
NumericalOptimizationProblem, 409
NumericalPDEProblem, 410
Octonion, 411
ODEIntensityFunctionsTable, 411
OneDimensionalArray, 412
OnePointCompletion, 412
OpenMathConnection, 413
OpenMathDevice, 413
OpenMathEncoding, 414
OpenMathError, 414
OpenMathErrorKind, 415
Operator, 415
OppositeMonogenicLinearOperator, 416
OrderedCompletion, 416
OrderedDirectProduct, 417
OrderedFreeMonoid, 417
OrderedVariableList, 418
OrderlyDifferentialPolynomial, 418

OrderlyDifferentialVariable, 419
OrdinaryDifferentialRing, 420
OrdinaryWeightedPolynomials, 420
OrdSetInts, 421
OutputForm, 421
PAdicInteger, 422
PAdicRational, 422
PAdicRationalConstructor, 423
Palette, 423
ParametricPlaneCurve, 423
ParametricSpaceCurve, 424
ParametricSurface, 424
PartialFraction, 425
Partition, 426
Pattern, 426
PatternMatchListResult, 427
PatternMatchResult, 427
PendantTree, 428
Permutation, 428
PermutationGroup, 429
Pi, 429
Places, 430
PlacesOverPseudoAlgebraicClosureOfFiniteField, 431
PlaneAlgebraicCurvePlot, 430
Plcs, 431
Plot, 432
Plot3D, 432
PoincareBirkhoffWittLyndonBasis, 433
Point, 433
Polynomial, 434
PolynomialIdeals, 434
PolynomialRing, 435
PositiveInteger, 435
PrimeField, 436
PrimitiveArray, 436
Product, 437
ProjectivePlane, 437
ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField, 438
ProjectiveSpace, 438
PseudoAlgebraicClosureOfAlgExtOfRationalNumber, 439
PseudoAlgebraicClosureOfFiniteField, 439
PseudoAlgebraicClosureOfRationalNumber, 440
QuadraticForm, 441
QuasiAlgebraicSet, 441
Quaternion, 442
QueryEquation, 443
Queue, 443
RadicalFunctionField, 444
RadixExpansion, 444
RealClosure, 445
RectangularMatrix, 445
Reference, 446
RegularChain, 446
RegularTriangularSet, 447
ResidueRing, 447
Result, 448
RewriteRule, 448
RightOpenIntervalRootCharacterization, 449
RomanNumeral, 449
RoutinesTable, 450
RuleCalled, 450
Ruleset, 451
ScriptFormulaFormat, 451
Segment, 452
SegmentBinding, 452
SequentialDifferentialPolynomial, 454
SequentialDifferentialVariable, 454
Set, 453
SetOfMIntegersInOneToN, 453
SExpression, 455
SExpressionOf, 456
SimpleAlgebraicExtension, 456
SimpleCell, 457
SimpleFortranProgram, 457
SingleInteger, 458
SingletonAsOrderedSet, 458
SparseEchelonMatrix, 459
SparseMultivariatePolynomial, 459
SparseMultivariateTaylorSeries, 460
SparseTable, 460
SparseUnivariateLaurentSeries, 461
SparseUnivariatePolynomial, 461
SparseUnivariatePolynomialExpressions, 462
SparseUnivariatePuiseuxSeries, 462
SparseUnivariateSkewPolynomial, 463
SparseUnivariateTaylorSeries, 463
SplitHomogeneousDirectProduct, 464
SplittingNode, 464
SplittingTree, 465
SquareFreeRegularTriangularSet, 466
SquareMatrix, 466
Stack, 467
StochasticDifferential, 467
Stream, 468
String, 468

StringTable, 469
SubSpace, 469
SubSpaceComponentProperty, 470
SuchThat, 470
Switch, 471
Symbol, 471
SymbolTable, 472
SymmetricPolynomial, 472
Table, 473
Tableau, 473
TaylorSerieso, 474
TexFormat, 474
TextFile, 475
TheSymbolTable, 475
ThreeDimensionalMatrix, 476
ThreeDimensionalViewport, 476
ThreeSpace, 477
Tree, 477
TubePlot, 478
Tuple, 478
TwoDimensionalArray, 479
TwoDimensionalViewport, 479
U16Matrix, 486
U16Vector, 487
U32Matrix, 486
U32Vector, 488
U8Matrix, 485
U8Vector, 487
UnivariateFormalPowerSeries, 480
UnivariateLaurentSeries, 480
UnivariateLaurentSeriesConstructor, 481
UnivariatePolynomial, 481
UnivariatePuiseuxSeries, 482
UnivariatePuiseuxSeriesConstructor, 482
UnivariatePuiseuxSeriesWithExponentialSingularity, 483
UnivariateSkewPolynomial, 483
UnivariateTaylorSeries, 484
UnivariateTaylorSeriesCZero, 485
UniversalSegment, 485
Variable, 489
Vector, 489
Void, 490
WeightedPolynomials, 490
WuWenTsunTriangularSet, 491
XDistributedPolynomial, 491
XPBWPolynomial, 492
XPolynomial, 492
XPolynomialRing, 493
XRecursivePolynomial, 493

DoubleFloat
Domain, 320
DoubleFloatMatrix
Domain, 321
DoubleFloatVector
Domain, 322
DPMM, 318
DPMO, 318
DrawOption
Domain, 322
DROPT, 322
DSMP, 317
DSTREE, 316

==== **E** ====

E04DGFA, 338
e04dgfAnnaType
Domain, 338
E04FDFA, 338
e04fdfAnnaType
Domain, 338
E04GCFA, 339
e04gcfAnnaType
Domain, 339
E04JAFA, 340
e04jafAnnaType
Domain, 340
E04MBFA, 340
e04mbfAnnaType
Domain, 340
E04NAFA, 341
e04nafAnnaType
Domain, 341
E04UCFA, 341
e04ucfAnnaType
Domain, 341
EAB, 337
EFULS, 332
EFUPXS, 333
ElementaryFunctionsUnivariateLaurentSeries
Domain, 332
ElementaryFunctionsUnivariatePuiseuxSeries
Domain, 333
EMR, 334
EQ, 333
EqTable
Domain, 334
EQTBL, 334
Equation
Domain, 333
EuclideanModularRing
Domain, 334
EXIT, 335
Exit

Domain, 335
EXPEXPAN, 335
ExponentialExpansion
Domain, 335
ExponentialOfUnivariatePuiseuxSeries
Domain, 337
EXPR, 336
Expression
Domain, 336
EXPUPXS, 337
ExtAlgBasis
Domain, 337

==== **F** ====

Factored
Domain, 342
FAGROUP, 357
FAMONOID, 357
FARRAY, 349
FC, 351
FCOMP, 354
FDIV, 344
FEXPR, 352
FF, 344
FFCG, 345
FFCGP, 346
FFCGX, 345
FFNB, 347
FFNBP, 349
FFNBX, 348
FFP, 347
FFX, 346
FGROUP, 358
FILE, 343
File
Domain, 343
FileName
Domain, 343
FiniteDivisor
Domain, 344
FiniteField
Domain, 344
FiniteFieldCyclicGroup
Domain, 345
FiniteFieldCyclicGroupExtension
Domain, 345
FiniteFieldCyclicGroupExtensionByPolynomial
Domain, 346
FiniteFieldExtension
Domain, 346
FiniteFieldExtensionByPolynomial
Domain, 347
FiniteFieldNormalBasis
Domain, 347
FiniteFieldNormalBasisExtension
Domain, 348
FiniteFieldNormalBasisExtensionByPolynomial
Domain, 349
FlexibleArray
Domain, 349
FLOAT, 350
Float
Domain, 350
FM, 358
FM1, 359
FMONOID, 359
FNAME, 343
FNLA, 360
FORMULA, 451
FORTRAN, 352
FortranCode
Domain, 351
FortranExpression
Domain, 352
FortranProgram
Domain, 352
FortranScalarType
Domain, 353
FortranTemplate
Domain, 353
FortranType
Domain, 354
FourierComponent
Domain, 354
FourierSeries
Domain, 355
FPARFRAC, 360
FR, 342
FRAC, 355
Fraction
Domain, 355
FractionalIdeal
Domain, 356
FramedModule
Domain, 356
FreeAbelianGroup
Domain, 357
FreeAbelianMonoid
Domain, 357
FreeGroup
Domain, 358
FreeModule
Domain, 358
FreeModule1
Domain, 359
FreeMonoid
Domain, 359

FreeNilpotentLie
 Domain, 360
FRIDEAL, 356
FRMOD, 356
FSERIES, 355
FST, 353
FT, 354
FTEM, 353
FullPartialFractionExpansion
 Domain, 360
FUNCTION, 361
FunctionCalled
 Domain, 361

==== **G** ====

GCNAALG, 362
GDMP, 361
GeneralDistributedMultivariatePolynomial
 Domain, 361
GeneralModulePolynomial
 Domain, 362
GeneralPolynomialSet
 Domain, 363
GeneralSparseTable
 Domain, 363
GeneralTriangularSet
 Domain, 364
GeneralUnivariatePowerSeries
 Domain, 364
GenericNonAssociativeAlgebra
 Domain, 362
GMODPOL, 362
GOPT, 365
GOPT0, 366
GPOLSET, 363
GraphImage
 Domain, 365
GRIMAGE, 365
GSERIES, 364
GSTBL, 363
GTSET, 364
GuessOption
 Domain, 365
GuessOptionFunctions0
 Domain, 366

==== **H** ====

HACKPI, 429
HashTable
 Domain, 366
HASHTBL, 366
HDMP, 369
HDP, 368
HEAP, 367
Heap
 Domain, 367
HELLFDIV, 369
HEXADEC, 367
HexadecimalExpansion
 Domain, 367
HomogeneousDirectProduct
 Domain, 368
HomogeneousDistributedMultivariatePolynomial
 Domain, 369
HTMLFORM, 368
HTMLFormat
 Domain, 368
HyperellipticFiniteDivisor
 Domain, 369

==== **I** ====

IAN, 380
IARRAY1, 376
IARRAY2, 377
IBITS, 371
ICARD, 370
ICP, 370
IDEAL, 434
IDPAG, 371
IDPAM, 372
IDPO, 372
IDPOAM, 373
IDPOAMS, 373
IFAMON, 381
IFARRAY, 374
IFF, 380
IIARRAY2, 381
ILIST, 375
IMATRIX, 376
INDE, 374
IndexCard
 Domain, 370
IndexedBits
 Domain, 371
IndexedDirectProductAbelianGroup
 Domain, 371
IndexedDirectProductAbelianMonoid
 Domain, 372
IndexedDirectProductObject
 Domain, 372
IndexedDirectProductOrderedAbelianMonoid
 Domain, 373
IndexedDirectProductOrderedAbelianMonoidSup
 Domain, 373
IndexedExponents

Domain, 374
IndexedFlexibleArray
Domain, 374
IndexedList
Domain, 375
IndexedMatrix
Domain, 376
IndexedOneDimensionalArray
Domain, 376
IndexedString
Domain, 377
IndexedTwoDimensionalArray
Domain, 377
IndexedVector
Domain, 378
INFCLSPS, 379
INFCLSPT, 379
InfClsPt
Domain, 370
InfiniteTuple
Domain, 378
InfinitlyClosePoint
Domain, 379
InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField
Domain, 379
INFORM, 384
InnerAlgebraicNumber
Domain, 380
InnerFiniteField
Domain, 380
InnerFreeAbelianMonoid
Domain, 381
InnerIndexedTwoDimensionalArray
Domain, 381
InnerPAdicInteger
Domain, 382
InnerPrimeField
Domain, 382
InnerSparseUnivariatePowerSeries
Domain, 383
InnerTable
Domain, 383
InnerTaylorSeries
Domain, 384
InputForm
Domain, 384
INT, 385
INTABL, 383
Integer
Domain, 385
IntegerMod
Domain, 385
IntegrationFunctionsTable
Domain, 386
IntegrationResult
Domain, 386
Interval
Domain, 387
INTFTBL, 386
INTRVL, 387
IPADIC, 382
IPF, 382
IR, 386
ISTRING, 377
ISUPS, 383
ITAYLOR, 384
ITUPLE, 378
IVECTOR, 378

==== **J** ====

JORDAN, 296

==== **K** ====

KAFILE, 388
KERNEL, 387
Kernel
Domain, 387
KeyedAccessFile
Domain, 388

==== **L** ====

LA, 394
LAUPOL, 388
LaurentPolynomial
Domain, 388
LEXP, 389
LIB, 389
Library
Domain, 389
LIE, 296
LieExponentials
Domain, 389
LiePolynomial
Domain, 390
LieSquareMatrix
Domain, 390
LinearOrdinaryDifferentialOperator
Domain, 391
LinearOrdinaryDifferentialOperator1
Domain, 392
LinearOrdinaryDifferentialOperator2
Domain, 392
LIST, 393
List
Domain, 393
ListMonoidOps

Domain, 393
ListMultiDictionary
Domain, 394
LMDICT, 394
LMOPS, 393
LO, 395
LocalAlgebra
Domain, 394
Localize
Domain, 395
LODO, 391
LODO1, 392
LODO2, 392
LPOLY, 390
LSQM, 390
LWORD, 395
LyndonWord
Domain, 395

==== **M** ====

M3D, 476
MachineComplex
Domain, 396
MachineFloat
Domain, 396
MachineInteger
Domain, 397
MAGMA, 397
Magma
Domain, 397
MakeCachableSet
Domain, 398
MathMLFormat
Domain, 398
MATRIX, 399
Matrix
Domain, 399
MCMPLX, 396
MFLOAT, 396
MINT, 397
MKCHSET, 398
MMLFORM, 398
MODFIELD, 400
MODMON, 399
ModMonic
Domain, 399
MODMONOM, 401
MODOP, 401
MODRING, 400
ModularField
Domain, 400
ModularRing
Domain, 400
ModuleMonomial
Domain, 401
ModuleOperator
Domain, 401
MOEBIUS, 402
MoebiusTransform
Domain, 402
MonoidRing
Domain, 402
MPOLY, 403
MRING, 402
MSET, 403
Multiset
Domain, 403
MultivariatePolynomial
Domain, 403
MYEXPR, 404
MyExpression
Domain, 404
MyUnivariatePolynomial
Domain, 404
MYUP, 404

==== **N** ====

NeitherSparseOrDensePowerSeries
Domain, 405
NewSparseMultivariatePolynomial
Domain, 405
NewSparseUnivariatePolynomial
Domain, 406
NIPROB, 408
NNI, 407
NONE, 406
None
Domain, 406
NonNegativeInteger
Domain, 407
NOTTING, 407
NottinghamGroup
Domain, 407
NSDPS, 405
NSMP, 405
NSUP, 406
NumericalIntegrationProblem
Domain, 408
NumericalODEProblem
Domain, 409
NumericalOptimizationProblem
Domain, 409
NumericalPDEProblem
Domain, 410

==== **O** ====

OCT, 411
Octonion
    Domain, 411
ODEIFTBL, 411
ODEIntensityFunctionsTable
    Domain, 411
ODEPROB, 409
ODP, 417
ODPOL, 418
ODR, 420
ODVAR, 419
OFMONOID, 417
OMCONN, 413
OMDEV, 413
OMENC, 414
OMERR, 414
OMERRK, 415
OMLO, 416
ONECOMP, 412
OneDimensionalArray
    Domain, 412
OnePointCompletion
    Domain, 412
OP, 415
OpenMathConnection
    Domain, 413
OpenMathDevice
    Domain, 413
OpenMathEncoding
    Domain, 414
OpenMathError
    Domain, 414
OpenMathErrorKind
    Domain, 415
Operator
    Domain, 415
OppositeMonogenicLinearOperator
    Domain, 416
OPTPROB, 409
ORDCOMP, 416
OrderedCompletion
    Domain, 416
OrderedDirectProduct
    Domain, 417
OrderedFreeMonoid
    Domain, 417
OrderedVariableList
    Domain, 418
OrderlyDifferentialPolynomial
    Domain, 418
OrderlyDifferentialVariable
    Domain, 419
OrdinaryDifferentialRing
    Domain, 420
OrdinaryWeightedPolynomials
    Domain, 420
OrdSetInts
    Domain, 421
ORESUP, 463
OREUP, 483
OSI, 421
OUTFORM, 421
OutputForm
    Domain, 421
OVAR, 418
OWP, 420

==== **P** ====

PACEXT, 439
PACOFF, 439
PACRAT, 440
PADIC, 422
PAdicInteger
    Domain, 422
PADICRAT, 422
PAdicRational
    Domain, 422
PAdicRationalConstructor
    Domain, 423
PADICRC, 423
PALETTE, 423
Palette
    Domain, 423
ParametricPlaneCurve
    Domain, 423
ParametricSpaceCurve
    Domain, 424
ParametricSurface
    Domain, 424
PARPCURV, 423
PARSCURV, 424
PARSURF, 424
PartialFraction
    Domain, 425
Partition
    Domain, 426
PATLRES, 427
PATRES, 427
PATTERN, 426
Pattern
    Domain, 426
PatternMatchListResult
    Domain, 427
PatternMatchResult
    Domain, 427
PBWLB, 433

PDEPROB, 410
PendantTree
    Domain, 428
PENDTREE, 428
PERM, 428
PERMGRP, 429
Permutation
    Domain, 428
PermutationGroup
    Domain, 429
PF, 436
PFR, 425
PI, 435
Pi
    Domain, 429
PLACES, 430
Places
    Domain, 430
PlacesOverPseudoAlgebraicClosureOfFiniteField
    Domain, 431
PLACESPS, 431
PlaneAlgebraicCurvePlot
    Domain, 430
PLCS, 431
Plcs
    Domain, 431
PLOT, 432
Plot
    Domain, 432
PLOT3D, 432
Plot3D
    Domain, 432
PoincareBirkhoffWittLyndonBasis
    Domain, 433
POINT, 433
Point
    Domain, 433
POLY, 434
Polynomial
    Domain, 434
PolynomialIdeals
    Domain, 434
PolynomialRing
    Domain, 435
PositiveInteger
    Domain, 435
PR, 435
PRIMARR, 436
PrimeField
    Domain, 436
PrimitiveArray
    Domain, 436
PRODUCT, 437
Product
    Domain, 437
ProjectivePlane
    Domain, 437
ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField
    Domain, 438
ProjectiveSpace
    Domain, 438
PROJPL, 437
PROJPLPS, 438
PROJSP, 438
PRTITION, 426
PseudoAlgebraicClosureOfAlgExtOfRationalNumber
    Domain, 439
PseudoAlgebraicClosureOfFiniteField
    Domain, 439
PseudoAlgebraicClosureOfRationalNumber
    Domain, 440

==== **Q** ====

QALGSET, 441
QEQUAT, 443
QFORM, 441
QuadraticForm
    Domain, 441
QuasiAlgebraicSet
    Domain, 441
QUAT, 442
Quaternion
    Domain, 442
QueryEquation
    Domain, 443
QUEUE, 443
Queue
    Domain, 443

==== **R** ====

RADFF, 444
RadicalFunctionField
    Domain, 444
RADIX, 444
RadixExpansion
    Domain, 444
RealClosure
    Domain, 445
RECLOS, 445
RectangularMatrix
    Domain, 445
REF, 446
Reference
    Domain, 446
REGSET, 447
RegularChain

Domain, 446
RegularTriangularSet
Domain, 447
ResidueRing
Domain, 447
RESRING, 447
RESULT, 448
Result
Domain, 448
RewriteRule
Domain, 448
RGCHAIN, 446
RightOpenIntervalRootCharacterization
Domain, 449
RMATRIX, 445
ROIRC, 449
ROMAN, 449
RomanNumeral
Domain, 449
ROUTINE, 450
RoutinesTable
Domain, 450
RULE, 448
RuleCalled
Domain, 450
RULECOLD, 450
RULESET, 451
Ruleset
Domain, 451

==== **S** ====

SAE, 456
SAOS, 458
SCELL, 457
ScriptFormulaFormat
Domain, 451
SD, 467
SDPOL, 454
SDVAR, 454
SEG, 452
SEGBIND, 452
Segment
Domain, 452
SegmentBinding
Domain, 452
SEM, 459
SequentialDifferentialPolynomial
Domain, 454
SequentialDifferentialVariable
Domain, 454
SET, 453
Set
Domain, 453
SETMN, 453
SetOfMIntegersInOneToN
Domain, 453
SEX, 455
SEXOF, 456
SExpression
Domain, 455
SExpressionOf
Domain, 456
SFORT, 457
SHDP, 464
SimpleAlgebraicExtension
Domain, 456
SimpleCell
Domain, 457
SimpleFortranProgram
Domain, 457
SingleInteger
Domain, 458
SingletonAsOrderedSet
Domain, 458
SINT, 458
SMP, 459
SMTS, 460
SPACE3, 477
SparseEchelonMatrix
Domain, 459
SparseMultivariatePolynomial
Domain, 459
SparseMultivariateTaylorSeries
Domain, 460
SparseTable
Domain, 460
SparseUnivariateLaurentSeries
Domain, 461
SparseUnivariatePolynomial
Domain, 461
SparseUnivariatePolynomialExpressions
Domain, 462
SparseUnivariatePuiseuxSeries
Domain, 462
SparseUnivariateSkewPolynomial
Domain, 463
SparseUnivariateTaylorSeries
Domain, 463
SplitHomogeneousDirectProduct
Domain, 464
SplittingNode
Domain, 464
SplittingTree
Domain, 465
SPLNODE, 464
SPLTREE, 465
SQMATRIX, 466

SquareFreeRegularTriangularSet
  Domain, 466
SquareMatrix
  Domain, 466
SREGSET, 466
STACK, 467
Stack
  Domain, 467
STBL, 460
StochasticDifferential
  Domain, 467
STREAM, 468
Stream
  Domain, 468
STRING, 468
String
  Domain, 468
StringTable
  Domain, 469
STRTBL, 469
SUBSPACE, 469
SubSpace
  Domain, 469
SubSpaceComponentProperty
  Domain, 470
SUCH, 470
SuchThat
  Domain, 470
SULS, 461
SUP, 461
SUPEXPR, 462
SUPXS, 462
SUTS, 463
SWITCH, 471
Switch
  Domain, 471
SYMBOL, 471
Symbol
  Domain, 471
SymbolTable
  Domain, 472
SymmetricPolynomial
  Domain, 472
SYMPOLY, 472
SYMS, 475
SYMTAB, 472

==== **T** ====

TABLE, 473
Table
  Domain, 473
TABLEAU, 473
Tableau
  Domain, 473
TaylorSerieso
  Domain, 474
TEX, 474
TexFormat
  Domain, 474
TEXTFILE, 475
TextFile
  Domain, 475
TheSymbolTable
  Domain, 475
ThreeDimensionalMatrix
  Domain, 476
ThreeDimensionalViewport
  Domain, 476
ThreeSpace
  Domain, 477
TREE, 477
Tree
  Domain, 477
TS, 474
TUBE, 478
TubePlot
  Domain, 478
TUPLE, 478
Tuple
  Domain, 478
TwoDimensionalArray
  Domain, 479
TwoDimensionalViewport
  Domain, 479

==== **U** ====

U16MAT, 486
U16Matrix
  Domain, 486
U16VEC, 487
U16Vector
  Domain, 487
U32MAT, 486
U32Matrix
  Domain, 486
U32VEC, 488
U32Vector
  Domain, 488
U8MAT, 485
U8Matrix
  Domain, 485
U8VEC, 487
U8Vector
  Domain, 487
UFPS, 480
ULS, 480

ULSCONS, 481
UNISEG, 485
UnivariateFormalPowerSeries
  Domain, 480
UnivariateLaurentSeries
  Domain, 480
UnivariateLaurentSeriesConstructor
  Domain, 481
UnivariatePolynomial
  Domain, 481
UnivariatePuiseuxSeries
  Domain, 482
UnivariatePuiseuxSeriesConstructor
  Domain, 482
UnivariatePuiseuxSeriesWithExponentialSingularity
  Domain, 483
UnivariateSkewPolynomial
  Domain, 483
UnivariateTaylorSeries
  Domain, 484
UnivariateTaylorSeriesCZero
  Domain, 485
UniversalSegment
  Domain, 485
UP, 481
UPXS, 482
UPXSCONS, 482
UPXSSING, 483
UTS, 484
UTSZ, 485

==== **V** ====

VARIABLE, 489
Variable
  Domain, 489
VECTOR, 489
Vector
  Domain, 489
VIEW2D, 479
VIEW3D, 476
VOID, 490
Void
  Domain, 490

==== **W** ====

WeightedPolynomials
  Domain, 490
WP, 490
WUTSET, 491
WuWenTsunTriangularSet
  Domain, 491

==== **X** ====

XDistributedPolynomial
  Domain, 491
XDPOLY, 491
XPBWPOLY, 492
XPBWPolynomial
  Domain, 492
XPOLY, 492
XPolynomial
  Domain, 492
XPolynomialRing
  Domain, 493
XPR, 493
XRecursivePolynomial
  Domain, 493
XRPOLY, 493

==== **Z** ====

ZMOD, 385

Package Index

==== **A** ====

AF, 496
AFALGGRO, 495
AFALGRES, 495
AffineAlgebraicSetComputeWithGroebnerBasis
    Domain, 495
AffineAlgebraicSetComputeWithResultant
    Domain, 495
AlgebraicFunction
    Domain, 496
AlgebraicHermiteIntegration
    Domain, 496
AlgebraicIntegrate
    Domain, 497
AlgebraicIntegration
    Domain, 497
AlgebraicManipulations
    Domain, 498
AlgebraicMultFact
    Domain, 498
AlgebraPackage
    Domain, 499
ALGFACT, 499
AlgFactor
    Domain, 499
ALGMANIP, 498
ALGMFACT, 498
ALGPKG, 499
AnnaNumericalIntegrationPackage
    Domain, 500
AnnaNumericalOptimizationPackage
    Domain, 500
AnnaOrdinaryDifferentialEquationPackage
    Domain, 501
AnnaPartialDifferentialEquationPackage
    Domain, 501
ANY1, 502
AnyFunctions1
    Domain, 502
API, 502
ApplicationProgramInterface
    Domain, 502
APPLYORE, 503
ApplyRules
    Domain, 503
ApplyUnivariateSkewPolynomial
    Domain, 503
APPRULE, 503
ARRAY12, 655
ASSOCEQ, 503
AssociatedEquations
    Domain, 503
AttachPredicates
    Domain, 504
AxiomServer
    Domain, 504
AXSERV, 504

==== **B** ====

BalancedFactorisation
    Domain, 505
BALFACT, 505
BasicOperatorFunctions1
    Domain, 505
BEZIER, 506
Bezier
    Domain, 506
BEZOUT, 506
BezoutMatrix
    Domain, 506
BlowUpPackage
    Domain, 507
BLUPPACK, 507
BOP1, 505
BoundIntegerRoots
    Domain, 507
BOUNDZRO, 507
BRILL, 508
BrillhartTests
    Domain, 508

==== **C** ====

CAD, 520
CADU, 520
CARTEN2, 508
CartesianTensorFunctions2
 Domain, 508
CDEN, 512
ChangeOfVariable
 Domain, 509
CharacteristicPolynomialInMonogenicalAlgebra
 Domain, 509
CharacteristicPolynomialPackage
 Domain, 510
CHARPOL, 510
ChineseRemainderToolsForIntegralBases
 Domain, 510
CHVAR, 509
CINTSLPE, 514
CLIP, 743
CMPLXRT, 516
CoerceVectorMatrixPackage
 Domain, 511
COMBF, 511
COMBINAT, 595
CombinatorialFunction
 Domain, 511
CommonDenominator
 Domain, 512
COMMONOP, 512
CommonOperators
 Domain, 512
COMMUPC, 513
CommuteUnivariatePolynomialCategory
 Domain, 513
COMPFACT, 513
COMPLEX2, 514
ComplexFactorization
 Domain, 513
ComplexFunctions2
 Domain, 514
ComplexIntegerSolveLinearPolynomialEquation
 Domain, 514
ComplexPattern
 Domain, 515
ComplexPatternMatch
 Domain, 515
ComplexRootFindingPackage
 Domain, 515
ComplexRootPackage
 Domain, 516
ComplexTrigonometricManipulations
 Domain, 517
COMPLPAT, 515
ConstantLODE
 Domain, 517
CoordinateSystems
 Domain, 518
COORDSYS, 518
CPIMA, 509
CPMATCH, 515
CRAPACK, 518
CRApackage
 Domain, 518
CRFP, 515
CSTTOOLS, 519
CTRIGMNP, 517
CVMP, 511
CycleIndicators
 Domain, 519
CYCLES, 519
CyclicStreamTools
 Domain, 519
CYCLOTOM, 520
CyclotomicPolynomialPackage
 Domain, 520
CylindricalAlgebraicDecompositionPackage
 Domain, 520
CylindricalAlgebraicDecompositionUtilities
 Domain, 520

==== **D** ====

d01AgentsPackage
 Domain, 528
D01AGNT, 528
d01WeightsPackage
 Domain, 529
D01WGTS, 529
d02AgentsPackage
 Domain, 529
D02AGNT, 529
d03AgentsPackage
 Domain, 530
D03AGNT, 530
DBLRESP, 526
DDFACT, 525
DefiniteIntegrationTools
 Domain, 521
DEFINTEF, 531
DEFINTRF, 701
DEGRED, 521
DegreeReductionPackage
 Domain, 521
DesingTreePackage
 Domain, 522
DFINTTLS, 521
DFSFUN, 525

DiophantineSolutionPackage
    Domain, 522
DIOSP, 522
DirectProductFunctions2
    Domain, 523
DIRPROD2, 523
DiscreteLogarithmPackage
    Domain, 524
DISPLAY, 524
DisplayPackage
    Domain, 524
DistinctDegreeFactorize
    Domain, 525
DLP, 524
Domain
    AffineAlgebraicSetComputeWith-GroebnerBasis, 495
    AffineAlgebraicSetComputeWithResultant, 495
    AlgebraicFunction, 496
    AlgebraicHermiteIntegration, 496
    AlgebraicIntegrate, 497
    AlgebraicIntegration, 497
    AlgebraicManipulations, 498
    AlgebraicMultFact, 498
    AlgebraPackage, 499
    AlgFactor, 499
    AnnaNumericalIntegrationPackage, 500
    AnnaNumericalOptimizationPackage, 500
    AnnaOrdinaryDifferentialEquation-Package, 501
    AnnaPartialDifferentialEquationPackage, 501
    AnyFunctions1, 502
    ApplicationProgramInterface, 502
    ApplyRules, 503
    ApplyUnivariateSkewPolynomial, 503
    AssociatedEquations, 503
    AttachPredicates, 504
    AxiomServer, 504
    BalancedFactorisation, 505
    BasicOperatorFunctions1, 505
    Bezier, 506
    BezoutMatrix, 506
    BlowUpPackage, 507
    BoundIntegerRoots, 507
    BrillhartTests, 508
    CartesianTensorFunctions2, 508
    ChangeOfVariable, 509
    CharacteristicPolynomialInMonogenicalAlgebra, 509
    CharacteristicPolynomialPackage, 510
    ChineseRemainderToolsForIntegral-Bases, 510
    CoerceVectorMatrixPackage, 511
    CombinatorialFunction, 511
    CommonDenominator, 512
    CommonOperators, 512
    CommuteUnivariatePolynomialCategory, 513
    ComplexFactorization, 513
    ComplexFunctions2, 514
    ComplexIntegerSolveLinearPolynomialEquation, 514
    ComplexPattern, 515
    ComplexPatternMatch, 515
    ComplexRootFindingPackage, 515
    ComplexRootPackage, 516
    ComplexTrigonometricManipulations, 517
    ConstantLODE, 517
    CoordinateSystems, 518
    CRApackage, 518
    CycleIndicators, 519
    CyclicStreamTools, 519
    CyclotomicPolynomialPackage, 520
    CylindricalAlgebraicDecomposition-Package, 520
    CylindricalAlgebraicDecompositionUtilities, 520
    d01AgentsPackage, 528
    d01WeightsPackage, 529
    d02AgentsPackage, 529
    d03AgentsPackage, 530
    DefiniteIntegrationTools, 521
    DegreeReductionPackage, 521
    DesingTreePackage, 522
    DiophantineSolutionPackage, 522
    DirectProductFunctions2, 523
    DiscreteLogarithmPackage, 524
    DisplayPackage, 524
    DistinctDegreeFactorize, 525
    DoubleFloatSpecialFunctions, 525
    DoubleResultantPackage, 526
    DrawComplex, 526
    DrawNumericHack, 526

DrawOptionFunctions0, 527
DrawOptionFunctions1, 528
e04AgentsPackage, 545
EigenPackage, 530
ElementaryFunction, 531
ElementaryFunctionDefiniteIntegration, 531
ElementaryFunctionLODESolver, 532
ElementaryFunctionODESolver, 532
ElementaryFunctionSign, 533
ElementaryFunctionStructurePackage, 533
ElementaryIntegration, 534
ElementaryRischDE, 534
ElementaryRischDESystem, 534
EllipticFunctionsUnivariateTaylorSeries, 535
EquationFunctions2, 535
ErrorFunctions, 536
EuclideanGroebnerBasisPackage, 537
EvaluateCycleIndicators, 538
ExpertSystemContinuityPackage, 538
ExpertSystemContinuityPackage1, 538
ExpertSystemToolsPackage, 539
ExpertSystemToolsPackage1, 539
ExpertSystemToolsPackage2, 540
Export3D, 544
ExpressionFunctions2, 540
ExpressionSolve, 541
ExpressionSpaceFunctions1, 541
ExpressionSpaceFunctions2, 542
ExpressionSpaceODESolver, 542
ExpressionToOpenMath, 543
ExpressionToUnivariatePowerSeries, 543
ExpressionTubePlot, 544
FactoredFunctions, 545
FactoredFunctions2, 546
FactoredFunctionUtilities, 546
FactoringUtilities, 547
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber, 547
FactorisationOverPseudoAlgebraicClosureOfRationalNumber, 548
FGLMIfCanPackage, 548
FindOrderFinite, 549
FiniteAbelianMonoidRingFunctions2, 549
FiniteDivisorFunctions2, 550
FiniteFieldFactorization, 550
FiniteFieldFactorizationWithSizeParseBySideEffect, 551
FiniteFieldFunctions, 551
FiniteFieldHomomorphisms, 552
FiniteFieldPolynomialPackage, 552
FiniteFieldPolynomialPackage2, 553
FiniteFieldSolveLinearPolynomialEquation, 553
FiniteFieldSquareFreeDecomposition, 554
FiniteLinearAggregateFunctions2, 554
FiniteLinearAggregateSort, 555
FiniteSetAggregateFunctions2, 555
FloatingComplexPackage, 556
FloatingRealPackage, 556
FloatSpecialFunctions, 557
FortranCodePackage1, 557
FortranOutputStackPackage, 558
FortranPackage, 558
FractionalIdealFunctions2, 559
FractionFreeFastGaussian, 559
FractionFreeFastGaussianFractions, 560
FractionFunctions2, 560
FramedNonAssociativeAlgebraFunctions2, 561
FunctionalSpecialFunction, 561
FunctionFieldCategoryFunctions2, 562
FunctionFieldIntegralBasis, 562
FunctionSpaceAssertions, 563
FunctionSpaceAttachPredicates, 563
FunctionSpaceComplexIntegration, 564
FunctionSpaceFunctions2, 564
FunctionSpaceIntegration, 565
FunctionSpacePrimitiveElement, 565
FunctionSpaceReduce, 566
FunctionSpaceSum, 566
FunctionSpaceToExponentialExpansion, 567
FunctionSpaceToUnivariatePowerSeries, 567
FunctionSpaceUnivariatePolynomialFactor, 568
GaloisGroupFactorizationUtilities, 568
GaloisGroupFactorizer, 569
GaloisGroupPolynomialUtilities, 569
GaloisGroupUtilities, 570

GaussianFactorizationPackage, 570
GeneralHenselPackage, 571
GeneralizedMultivariateFactorize, 571
GeneralPackageForAlgebraicFunction-Field,
572
GeneralPolynomialGcdPackage, 572
GenerateUnivariatePowerSeries, 573
GenExEuclid, 573
GenUFactorize, 573
GenusZeroIntegration, 574
GnuDraw, 575
GosperSummationMethod, 575
GraphicsDefaults, 576
Graphviz, 576
GrayCode, 576
GroebnerFactorizationPackage, 577
GroebnerInternalPackage, 578
GroebnerPackage, 578
GroebnerSolve, 579
Guess, 579
GuessAlgebraicNumber, 580
GuessFinite, 580
GuessFiniteFunctions, 581
GuessInteger, 581
GuessPolynomial, 582
GuessUnivariatePolynomial, 582
HallBasis, 583
HeuGcd, 583
IdealDecompositionPackage, 584
IncrementingMaps, 584
InfiniteProductCharacteristicZero, 585
InfiniteProductFiniteField, 585
InfiniteProductPrimeField, 586
InfiniteTupleFunctions2, 586
InfiniteTupleFunctions3, 587
Infinity, 587
InnerAlgFactor, 588
InnerCommonDenominator, 588
InnerMatrixLinearAlgebraFunctions,
589
InnerMatrixQuotientFieldFunctions,
589
InnerModularGcd, 590
InnerMultFact, 590
InnerNormalBasisFieldFunctions, 591
InnerNumericEigenPackage, 591
InnerNumericFloatSolvePackage, 592
InnerPolySign, 592
InnerPolySum, 593
InnerTrigonometricManipulations, 593
InputFormFunctions1, 594
IntegerBits, 594
IntegerCombinatoricFunctions, 595
IntegerFactorizationPackage, 595
IntegerLinearDependence, 596
IntegerNumberTheoryFunctions, 596
IntegerPrimesPackage, 597
IntegerRetractions, 597
IntegerRoots, 598
IntegerSolveLinearPolynomialEquation,
598
IntegralBasisPolynomialTools, 599
IntegralBasisTools, 599
IntegrationResultFunctions2, 600
IntegrationResultRFToFunction, 600
IntegrationResultToFunction, 601
IntegrationTools, 601
InterfaceGroebnerPackage, 594
InternalPrintPackage, 602
InternalRationalUnivariateRepresenta-tionPackage,
602
InterpolateFormsPackage, 603
IntersectionDivisorPackage, 603
InverseLaplaceTransform, 605
IrredPolyOverFiniteField, 604
IrrRepSymNatPackage, 604
KernelFunctions2, 605
Kovacic, 606
LaplaceTransform, 606
LazardSetSolvingPackage, 607
LeadingCoefDetermination, 607
LexTriangularPackage, 608
LinearDependence, 608
LinearOrdinaryDifferentialOperator-Factorizer,
609
LinearOrdinaryDifferentialOperator-sOps,
609
LinearPolynomialEquationByFractions,
610
LinearSystemFromPowerSeriesPack-age,
610
LinearSystemMatrixPackage, 611
LinearSystemMatrixPackage1, 611
LinearSystemPolynomialPackage, 612
LinesOpPack, 613
LinGroebnerPackage, 612
LiouvillianFunction, 613
ListFunctions2, 614
ListFunctions3, 614
ListToMap, 615
LocalParametrizationOfSimplePoint-Package,
615

MakeBinaryCompiledFunction, 616
MakeFloatCompiledFunction, 616
MakeFunction, 617
MakeRecord, 617
MakeUnaryCompiledFunction, 618
MappingPackage1, 620
MappingPackage2, 620
MappingPackage3, 621
MappingPackage4, 621
MappingPackageInternalHacks1, 618
MappingPackageInternalHacks2, 619
MappingPackageInternalHacks3, 619
MatrixCategoryFunctions2, 622
MatrixCommonDenominator, 622
MatrixLinearAlgebraFunctions, 623
MatrixManipulation, 623
MergeThing, 624
MeshCreationRoutinesForThreeDimensions, 624
ModularDistinctDegreeFactorizer, 624
ModularHermitianRowReduction, 625
MonoidRingFunctions2, 625
MonomialExtensionTools, 626
MoreSystemCommands, 626
MPolyCatFunctions2, 628
MPolyCatFunctions3, 628
MPolyCatPolyFactorizer, 627
MPolyCatRationalFunctionFactorizer, 627
MRationalFactorize, 629
MultFiniteFactorize, 629
MultipleMap, 630
MultiVariableCalculusFunctions, 630
MultivariateFactorize, 631
MultivariateLifting, 631
MultivariateSquareFree, 632
NagEigenPackage, 632
NagFittingPackage, 633
NagIntegrationPackage, 635
NagInterpolationPackage, 635
NagLapack, 636
NagLinearEquationSolvingPackage, 634
NAGLinkSupportPackage, 634
NagMatrixOperationsPackage, 636
NagOptimisationPackage, 637
NagOrdinaryDifferentialEquationsPackage, 637
NagPartialDifferentialEquationsPackage, 638
NagPolynomialRootsPackage, 639
NagRootFindingPackage, 639
NagSeriesSummationPackage, 640
NagSpecialFunctionsPackage, 640
NewSparseUnivariatePolynomialFunctions2, 641
NewtonInterpolation, 641
NewtonPolygon, 642
NonCommutativeOperatorDivision, 642
NoneFunctions1, 643
NonLinearFirstOrderODESolver, 643
NonLinearSolvePackage, 644
NormalizationPackage, 644
NormInMonogenicAlgebra, 645
NormRetractPackage, 645
NPCoef, 645
NumberFieldIntegralBasis, 646
NumberFormats, 646
NumberTheoreticPolynomialFunctions, 647
Numeric, 647
NumericalOrdinaryDifferentialEquations, 648
NumericalQuadrature, 650
NumericComplexEigenPackage, 651
NumericContinuedFraction, 652
NumericRealEigenPackage, 652
NumericTubePlot, 653
OctonionCategoryFunctions2, 653
ODEIntegration, 654
ODETools, 654
OneDimensionalArrayFunctions2, 655
OnePointCompletionFunctions2, 655
OpenMathPackage, 656
OpenMathServerPackage, 656
OperationsQuery, 657
OrderedCompletionFunctions2, 657
OrderingFunctions, 658
OrthogonalPolynomialFunctions, 658
OutputPackage, 659
PackageForAlgebraicFunctionField, 659
PackageForAlgebraicFunctionFieldOverFiniteField, 660
PackageForPoly, 660
PadeApproximantPackage, 660
PadeApproximants, 661
PAdicWildFunctionFieldIntegralBasis, 661
ParadoxicalCombinatorsForStreams, 662

ParametricLinearEquations, 662
ParametricPlaneCurveFunctions2, 663
ParametricSpaceCurveFunctions2, 664
ParametricSurfaceFunctions2, 664
ParametrizationPackage, 664
PartialFractionPackage, 665
PartitionsAndPermutations, 665
PatternFunctions1, 666
PatternFunctions2, 666
PatternMatch, 667
PatternMatchAssertions, 667
PatternMatchFunctionSpace, 668
PatternMatchIntegerNumberSystem, 668
PatternMatchIntegration, 669
PatternMatchKernel, 669
PatternMatchListAggregate, 670
PatternMatchPolynomialCategory, 670
PatternMatchPushDown, 671
PatternMatchQuotientFieldCategory, 671
PatternMatchResultFunctions2, 672
PatternMatchSymbol, 672
PatternMatchTools, 672
Permanent, 673
PermutationGroupExamples, 673
PiCoercions, 674
PlotFunctions1, 674
PlotTools, 675
PointFunctions2, 676
PointPackage, 676
PointsOfFiniteOrder, 677
PointsOfFiniteOrderRational, 677
PointsOfFiniteOrderTools, 678
PolToPol, 679
PolyGroebner, 679
PolynomialAN2Expression, 680
PolynomialCategoryLifting, 680
PolynomialCategoryQuotientFunctions, 681
PolynomialComposition, 681
PolynomialDecomposition, 682
PolynomialFactorizationByRecursion, 682
PolynomialFactorizationByRecursionUnivariate, 683
PolynomialFunctions2, 683
PolynomialGcdPackage, 684
PolynomialInterpolation, 684
PolynomialInterpolationAlgorithms, 685
PolynomialNumberTheoryFunctions, 685
PolynomialPackageForCurve, 678
PolynomialRoots, 686
PolynomialSetUtilitiesPackage, 686
PolynomialSolveByFormulas, 687
PolynomialSquareFree, 687
PolynomialToUnivariatePolynomial, 688
PowerSeriesLimitPackage, 688
PrecomputedAssociatedEquations, 689
PrimitiveArrayFunctions2, 689
PrimitiveElement, 690
PrimitiveRatDE, 690
PrimitiveRatRicDE, 691
PrintPackage, 691
ProjectiveAlgebraicSetPackage, 675
PseudoLinearNormalForm, 692
PseudoRemainderSequence, 692
PureAlgebraicIntegration, 693
PureAlgebraicLODE, 693
PushVariables, 694
QuasiAlgebraicSet2, 694
QuasiComponentPackage, 695
QuaternionCategoryFunctions2, 696
QuotientFieldCategoryFunctions2, 696
RadicalEigenPackage, 697
RadicalSolvePackage, 697
RadixUtilities, 698
RandomDistributions, 698
RandomFloatDistributions, 699
RandomIntegerDistributions, 699
RandomNumberSource, 699
RationalFactorize, 700
RationalFunction, 701
RationalFunctionDefiniteIntegration, 701
RationalFunctionFactor, 702
RationalFunctionFactorizer, 702
RationalFunctionIntegration, 703
RationalFunctionLimitPackage, 703
RationalFunctionSign, 703
RationalFunctionSum, 704
RationalIntegration, 704
RationalInterpolation, 705
RationalLODE, 705
RationalRetractions, 706
RationalRicDE, 706
RationalUnivariateRepresentationPackage, 707
RealPolynomialUtilitiesPackage, 707
RealSolvePackage, 708
RealZeroPackage, 708

RealZeroPackageQ, 709
RectangularMatrixCategoryFunctions2, 709
RecurrenceOperator, 710
ReducedDivisor, 710
ReduceLODE, 711
ReductionOfOrder, 711
RegularSetDecompositionPackage, 712
RegularTriangularSetGcdPackage, 713
RepeatedDoubling, 713
RepeatedSquaring, 713
RepresentationPackage1, 714
RepresentationPackage2, 714
ResolveLatticeCompletion, 715
RetractSolvePackage, 716
RootsFindingPackage, 716
SAERationalFunctionAlgFactor, 717
ScriptFormulaFormat1, 717
SegmentBindingFunctions2, 718
SegmentFunctions2, 718
SimpleAlgebraicExtensionAlgFactor, 719
SimplifyAlgebraicNumberConvertPackage, 719
SmithNormalForm, 720
SortedCache, 720
SortPackage, 721
SparseUnivariatePolynomialFunctions2, 721
SpecialOutputPackage, 722
SquareFreeQuasiComponentPackage, 722
SquareFreeRegularSetDecompositionPackage, 723
SquareFreeRegularTriangularSetGcdPackage, 723
StorageEfficientMatrixOperations, 724
StreamFunctions1, 724
StreamFunctions2, 725
StreamFunctions3, 725
StreamInfiniteProduct, 726
StreamTaylorSeriesOperations, 726
StreamTensor, 727
StreamTranscendentalFunctions, 727
StreamTranscendentalFunctionsNonCommutative, 728
StructuralConstantsPackage, 728
SturmHabichtPackage, 729
SubResultantPackage, 729
SupFractionFactorizer, 730
SymmetricFunctions, 732
SymmetricGroupCombinatoricFunctions, 731
SystemODESolver, 730
SystemSolvePackage, 731
TableauxBumpers, 732
TabulatedComputationPackage, 733
TangentExpansions, 733
TaylorSolve, 734
TemplateUtilities, 734
TexFormat1, 735
ToolsForSign, 735
TopLevelDrawFunctions, 736
TopLevelDrawFunctionsForAlgebraicCurves, 736
TopLevelDrawFunctionsForCompiledFunctions, 737
TopLevelDrawFunctionsForPoints, 737
TopLevelThreeSpace, 738
TranscendentalHermiteIntegration, 738
TranscendentalIntegration, 739
TranscendentalManipulations, 739
TranscendentalRischDE, 740
TranscendentalRischDESystem, 740
TransSolvePackage, 741
TransSolvePackageService, 741
TriangularMatrixOperations, 742
TrigonometricManipulations, 742
TubePlotTools, 743
TwoDimensionalPlotClipping, 743
TwoFactorize, 744
U32VectorPolynomialOperations, 753
UnivariateFactorize, 744
UnivariateFormalPowerSeriesFunctions, 745
UnivariateLaurentSeriesFunctions2, 745
UnivariatePolynomialCategoryFunctions2, 746
UnivariatePolynomialCommonDenominator, 746
UnivariatePolynomialDecompositionPackage, 747
UnivariatePolynomialDivisionPackage, 747
UnivariatePolynomialFunctions2, 748

UnivariatePolynomialMultiplication-Package, 748
UnivariatePolynomialSquareFree, 749
UnivariatePuiseuxSeriesFunctions2, 750
UnivariateSkewPolynomialCategory-Ops, 750
UnivariateTaylorSeriesFunctions2, 751
UnivariateTaylorSeriesODESolver, 751
UniversalSegmentFunctions2, 752
UserDefinedPartialOrdering, 752
UserDefinedVariableOrdering, 752
UTSodetools, 753
VectorFunctions2, 754
ViewDefaultsPackage, 755
ViewportPackage, 755
WeierstrassPreparation, 756
WildFunctionFieldIntegralBasis, 756
XExponentialPackage, 757
ZeroDimensionalSolvePackage, 757
DoubleFloatSpecialFunctions
Domain, 525
DoubleResultantPackage
Domain, 526
DRAW, 736
DRAWCFUN, 737
DrawComplex
Domain, 526
DRAWCURV, 736
DRAWCX, 526
DRAWHACK, 526
DrawNumericHack
Domain, 526
DrawOptionFunctions0
Domain, 527
DrawOptionFunctions1
Domain, 528
DRAWPT, 737
DROPT0, 527
DROPT1, 528
DTP, 522

==== **E** ====

e04AgentsPackage
Domain, 545
E04AGNT, 545
EF, 531
EFSTRUC, 533
EigenPackage
Domain, 530
ElementaryFunction
Domain, 531
ElementaryFunctionDefiniteIntegration
Domain, 531
ElementaryFunctionLODESolver
Domain, 532
ElementaryFunctionODESolver
Domain, 532
ElementaryFunctionSign
Domain, 533
ElementaryFunctionStructurePackage
Domain, 533
ElementaryIntegration
Domain, 534
ElementaryRischDE
Domain, 534
ElementaryRischDESystem
Domain, 534
ELFUTS, 535
EllipticFunctionsUnivariateTaylorSeries
Domain, 535
EP, 530
EQ2, 535
EquationFunctions2
Domain, 535
ERROR, 536
ErrorFunctions
Domain, 536
ES1, 541
ES2, 542
ESCONT, 538
ESCONT1, 538
ESTOOLS, 539
ESTOOLS1, 539
ESTOOLS2, 540
EuclideanGroebnerBasisPackage
Domain, 537
EVALCYC, 538
EvaluateCycleIndicators
Domain, 538
EXP3D, 544
ExpertSystemContinuityPackage
Domain, 538
ExpertSystemContinuityPackage1
Domain, 538
ExpertSystemToolsPackage
Domain, 539
ExpertSystemToolsPackage1
Domain, 539
ExpertSystemToolsPackage2
Domain, 540
Export3D
Domain, 544
EXPR2, 540
EXPR2UPS, 543

ExpressionFunctions2
  Domain, 540
ExpressionSolve
  Domain, 541
ExpressionSpaceFunctions1
  Domain, 541
ExpressionSpaceFunctions2
  Domain, 542
ExpressionSpaceODESolver
  Domain, 542
ExpressionToOpenMath
  Domain, 543
ExpressionToUnivariatePowerSeries
  Domain, 543
ExpressionTubePlot
  Domain, 544
EXPRODE, 542
EXPRSOL, 541
EXPRTUBE, 544

==== **F** ====

FACTEXT, 547
FACTFUNC, 545
FactoredFunctions
  Domain, 545
FactoredFunctions2
  Domain, 546
FactoredFunctionUtilities
  Domain, 546
FactoringUtilities
  Domain, 547
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
  Domain, 547
FactorisationOverPseudoAlgebraicClosureOfRationalNumber
  Domain, 548
FACTRN, 548
FACUTIL, 547
FAMR2, 549
FCPAK1, 557
FDIV2, 550
FFCAT2, 562
FFF, 551
FFFACTOR, 550
FFFACTSE, 551
FFFG, 559
FFFGF, 560
FFHOM, 552
FFINTBAS, 562
FFPOLY, 552
FFPOLY2, 553
FFSLPE, 553
FFSQFR, 554
FGLMICPK, 548
FGLMIfCanPackage
  Domain, 548
FindOrderFinite
  Domain, 549
FiniteAbelianMonoidRingFunctions2
  Domain, 549
FiniteDivisorFunctions2
  Domain, 550
FiniteFieldFactorization
  Domain, 550
FiniteFieldFactorizationWithSizeParseBySideEffect
  Domain, 551
FiniteFieldFunctions
  Domain, 551
FiniteFieldHomomorphisms
  Domain, 552
FiniteFieldPolynomialPackage
  Domain, 552
FiniteFieldPolynomialPackage2
  Domain, 553
FiniteFieldSolveLinearPolynomialEquation
  Domain, 553
FiniteFieldSquareFreeDecomposition
  Domain, 554
FiniteLinearAggregateFunctions2
  Domain, 554
FiniteLinearAggregateSort
  Domain, 555
FiniteSetAggregateFunctions2
  Domain, 555
FLAGG2, 554
FLASORT, 555
FLOATCP, 556
FloatingComplexPackage
  Domain, 556
FloatingRealPackage
  Domain, 556
FLOATRP, 556
FloatSpecialFunctions
  Domain, 557
FOP, 558
FORDER, 549
FORMULA1, 717
FORT, 558
FortranCodePackage1
  Domain, 557
FortranOutputStackPackage
  Domain, 558
FortranPackage
  Domain, 558
FR2, 546
FRAC2, 560
FractionalIdealFunctions2
  Domain, 559

FractionFreeFastGaussian
  Domain, 559
FractionFreeFastGaussianFractions
  Domain, 560
FractionFunctions2
  Domain, 560
FramedNonAssociativeAlgebraFunctions2
  Domain, 561
FRIDEAL2, 559
FRNAAF2, 561
FRUTIL, 546
FS2, 564
FS2EXPXP, 567
FS2UPS, 567
FSAGG2, 555
FSCINT, 564
FSFUN, 557
FSINT, 565
FSPECF, 561
FSPRMELT, 565
FSRED, 566
FSUPFACT, 568
FunctionalSpecialFunction
  Domain, 561
FunctionFieldCategoryFunctions2
  Domain, 562
FunctionFieldIntegralBasis
  Domain, 562
FunctionSpaceAssertions
  Domain, 563
FunctionSpaceAttachPredicates
  Domain, 563
FunctionSpaceComplexIntegration
  Domain, 564
FunctionSpaceFunctions2
  Domain, 564
FunctionSpaceIntegration
  Domain, 565
FunctionSpacePrimitiveElement
  Domain, 565
FunctionSpaceReduce
  Domain, 566
FunctionSpaceSum
  Domain, 566
FunctionSpaceToExponentialExpansion
  Domain, 567
FunctionSpaceToUnivariatePowerSeries
  Domain, 567
FunctionSpaceUnivariatePolynomialFactor
  Domain, 568

==== **G** ====

GALFACT, 569
GALFACTU, 568
GaloisGroupFactorizationUtilities
  Domain, 568
GaloisGroupFactorizer
  Domain, 569
GaloisGroupPolynomialUtilities
  Domain, 569
GaloisGroupUtilities
  Domain, 570
GALPOLYU, 569
GALUTIL, 570
GAUSSFAC, 570
GaussianFactorizationPackage
  Domain, 570
GB, 578
GBEUCLID, 537
GBF, 577
GBINTERN, 578
GDRAW, 575
GENEEZ, 573
GeneralHenselPackage
  Domain, 571
GeneralizedMultivariateFactorize
  Domain, 571
GeneralPackageForAlgebraicFunctionField
  Domain, 572
GeneralPolynomialGcdPackage
  Domain, 572
GenerateUnivariatePowerSeries
  Domain, 573
GenExEuclid
  Domain, 573
GENMFACT, 571
GENPGCD, 572
GENUFACT, 573
GenUFactorize
  Domain, 573
GENUPS, 573
GenusZeroIntegration
  Domain, 574
GHENSEL, 571
GnuDraw
  Domain, 575
GOSPER, 575
GosperSummationMethod
  Domain, 575
GPAFF, 572
GraphicsDefaults
  Domain, 576
GRAPHVIZ, 576
Graphviz
  Domain, 576
GRAY, 576
GrayCode

Domain, 576
GRDEF, 576
GroebnerFactorizationPackage
Domain, 577
GroebnerInternalPackage
Domain, 578
GroebnerPackage
Domain, 578
GroebnerSolve
Domain, 579
GROEBSOL, 579
GUESS, 579
Guess
Domain, 579
GuessAlgebraicNumber
Domain, 580
GUESSAN, 580
GUESSF, 580
GUESSF1, 581
GuessFinite
Domain, 580
GuessFiniteFunctions
Domain, 581
GUESSINT, 581
GuessInteger
Domain, 581
GUESSP, 582
GuessPolynomial
Domain, 582
GuessUnivariatePolynomial
Domain, 582
GUESSUP, 582

==== **H** ====

HallBasis
Domain, 583
HB, 583
HEUGCD, 583
HeuGcd
Domain, 583

==== **I** ====

IALGFACT, 588
IBACHIN, 510
IBATOOL, 599
IBPTOOLS, 599
ICDEN, 588
IdealDecompositionPackage
Domain, 584
IDECOMP, 584
IMATLIN, 589
IMATQF, 589
INBFF, 591
IncrementingMaps
Domain, 584
INCRMAPS, 584
INEP, 591
InfiniteProductCharacteristicZero
Domain, 585
InfiniteProductFiniteField
Domain, 585
InfiniteProductPrimeField
Domain, 586
InfiniteTupleFunctions2
Domain, 586
InfiniteTupleFunctions3
Domain, 587
INFINITY, 587
Infinity
Domain, 587
INFORM1, 594
INFPROD0, 585
INFSP, 592
INMODGCD, 590
InnerAlgFactor
Domain, 588
InnerCommonDenominator
Domain, 588
InnerMatrixLinearAlgebraFunctions
Domain, 589
InnerMatrixQuotientFieldFunctions
Domain, 589
InnerModularGcd
Domain, 590
InnerMultFact
Domain, 590
InnerNormalBasisFieldFunctions
Domain, 591
InnerNumericEigenPackage
Domain, 591
InnerNumericFloatSolvePackage
Domain, 592
InnerPolySign
Domain, 592
InnerPolySum
Domain, 593
InnerTrigonometricManipulations
Domain, 593
INNMFACT, 590
INPRODFF, 585
INPRODPF, 586
INPSIGN, 592
InputFormFunctions1
Domain, 594
INTAF, 497
INTALG, 497

INTBIT, 594
INTDIVP, 603
INTEF, 534
IntegerBits
    Domain, 594
IntegerCombinatoricFunctions
    Domain, 595
IntegerFactorizationPackage
    Domain, 595
IntegerLinearDependence
    Domain, 596
IntegerNumberTheoryFunctions
    Domain, 596
IntegerPrimesPackage
    Domain, 597
IntegerRetractions
    Domain, 597
IntegerRoots
    Domain, 598
IntegerSolveLinearPolynomialEquation
    Domain, 598
IntegralBasisPolynomialTools
    Domain, 599
IntegralBasisTools
    Domain, 599
IntegrationResultFunctions2
    Domain, 600
IntegrationResultRFToFunction
    Domain, 600
IntegrationResultToFunction
    Domain, 601
IntegrationTools
    Domain, 601
InterfaceGroebnerPackage
    Domain, 594
INTERGB, 594
InternalPrintPackage
    Domain, 602
InternalRationalUnivariateRepresentationPackage
    Domain, 602
InterpolateFormsPackage
    Domain, 603
IntersectionDivisorPackage
    Domain, 603
INTFACT, 595
INTFRSP, 603
INTG0, 574
INTHEORY, 596
INTHERAL, 496
INTHERTR, 738
INTPACK, 500
INTPAF, 693
INTPM, 669
INTRAT, 704
INTRET, 597
INTRF, 703
INTSLPE, 598
INTTOOLS, 601
INTTR, 739
InverseLaplaceTransform
    Domain, 605
INVLAPLA, 605
IPRNTPK, 602
IR2, 600
IR2F, 601
IROOT, 598
IRREDFFX, 604
IrredPolyOverFiniteField
    Domain, 604
IRRF2F, 600
IrrRepSymNatPackage
    Domain, 604
IRSN, 604
IRURPK, 602
ISUMP, 593
ITFUN2, 586
ITFUN3, 587
ITRIGMNP, 593

==== **K** ====

KERNEL2, 605
KernelFunctions2
    Domain, 605
KOVACIC, 606
Kovacic
    Domain, 606

==== **L** ====

LAPLACE, 606
LaplaceTransform
    Domain, 606
LazardSetSolvingPackage
    Domain, 607
LAZM3PK, 607
LEADCDET, 607
LeadingCoefDetermination
    Domain, 607
LexTriangularPackage
    Domain, 608
LEXTRIPK, 608
LF, 613
LGROBP, 612
LIMITPS, 688
LIMITRF, 703
LINDEP, 608
LinearDependence

Domain, 608
LinearOrdinaryDifferentialOperatorFactorizer
Domain, 609
LinearOrdinaryDifferentialOperatorsOps
Domain, 609
LinearPolynomialEquationByFractions
Domain, 610
LinearSystemFromPowerSeriesPackage
Domain, 610
LinearSystemMatrixPackage
Domain, 611
LinearSystemMatrixPackage1
Domain, 611
LinearSystemPolynomialPackage
Domain, 612
LinesOpPack
Domain, 613
LinGroebnerPackage
Domain, 612
LiouvillianFunction
Domain, 613
LIST2, 614
LIST2MAP, 615
LIST3, 614
ListFunctions2
Domain, 614
ListFunctions3
Domain, 614
ListToMap
Domain, 615
LISYSER, 610
LocalParametrizationOfSimplePointPackage
Domain, 615
LODEEF, 532
LODOF, 609
LODOOPS, 609
LOP, 613
LPARSPT, 615
LPEFRAC, 610
LSMP, 611
LSMP1, 611
LSPP, 612

==== **M** ====

MakeBinaryCompiledFunction
Domain, 616
MakeFloatCompiledFunction
Domain, 616
MakeFunction
Domain, 617
MakeRecord
Domain, 617
MakeUnaryCompiledFunction
Domain, 618
MAMA, 623
MAPHACK1, 618
MAPHACK2, 619
MAPHACK3, 619
MappingPackage1
Domain, 620
MappingPackage2
Domain, 620
MappingPackage3
Domain, 621
MappingPackage4
Domain, 621
MappingPackageInternalHacks1
Domain, 618
MappingPackageInternalHacks2
Domain, 619
MappingPackageInternalHacks3
Domain, 619
MAPPKG1, 620
MAPPKG2, 620
MAPPKG3, 621
MAPPKG4, 621
MATCAT2, 622
MATLIN, 623
MatrixCategoryFunctions2
Domain, 622
MatrixCommonDenominator
Domain, 622
MatrixLinearAlgebraFunctions
Domain, 623
MatrixManipulation
Domain, 623
MATSTOR, 724
MCALCFN, 630
MCDEN, 622
MDDFACT, 624
MergeThing
Domain, 624
MESH, 624
MeshCreationRoutinesForThreeDimensions
Domain, 624
MFINFACT, 629
MHROWRED, 625
MKBCFUNC, 616
MKFLCFN, 616
MKFUNC, 617
MKRECORD, 617
MKUCFUNC, 618
MLIFT, 631
MMAP, 630
ModularDistinctDegreeFactorizer
Domain, 624
ModularHermitianRowReduction

Domain, 625
MonoidRingFunctions2
Domain, 625
MonomialExtensionTools
Domain, 626
MONOTOOL, 626
MoreSystemCommands
Domain, 626
MPC2, 628
MPC3, 628
MPCPF, 627
MPolyCatFunctions2
Domain, 628
MPolyCatFunctions3
Domain, 628
MPolyCatPolyFactorizer
Domain, 627
MPolyCatRationalFunctionFactorizer
Domain, 627
MPRFF, 627
MRATFAC, 629
MRationalFactorize
Domain, 629
MRF2, 625
MSYSCMD, 626
MTHING, 624
MULTFACT, 631
MultFiniteFactorize
Domain, 629
MultipleMap
Domain, 630
MultiVariableCalculusFunctions
Domain, 630
MultivariateFactorize
Domain, 631
MultivariateLifting
Domain, 631
MultivariateSquareFree
Domain, 632
MULTSQFR, 632

==== **N** ====

NAGC02, 639
NAGC05, 639
NAGC06, 640
NAGD01, 635
NAGD02, 637
NAGD03, 638
NAGE01, 635
NAGE02, 633
NAGE04, 637
NagEigenPackage
Domain, 632
NAGF01, 636
NAGF02, 632
NAGF04, 634
NAGF07, 636
NagFittingPackage
Domain, 633
NagIntegrationPackage
Domain, 635
NagInterpolationPackage
Domain, 635
NagLapack
Domain, 636
NagLinearEquationSolvingPackage
Domain, 634
NAGLinkSupportPackage
Domain, 634
NagMatrixOperationsPackage
Domain, 636
NagOptimisationPackage
Domain, 637
NagOrdinaryDifferentialEquationsPackage
Domain, 637
NagPartialDifferentialEquationsPackage
Domain, 638
NagPolynomialRootsPackage
Domain, 639
NagRootFindingPackage
Domain, 639
NAGS, 640
NagSeriesSummationPackage
Domain, 640
NAGSP, 634
NagSpecialFunctionsPackage
Domain, 640
NCEP, 651
NCNTFRAC, 652
NCODIV, 642
NewSparseUnivariatePolynomialFunctions2
Domain, 641
NEWTON, 641
NewtonInterpolation
Domain, 641
NewtonPolygon
Domain, 642
NFINTBAS, 646
NLINSOL, 644
NODE1, 643
NonCommutativeOperatorDivision
Domain, 642
NONE1, 643
NoneFunctions1
Domain, 643
NonLinearFirstOrderODESolver
Domain, 643

NonLinearSolvePackage
  Domain, 644
NormalizationPackage
  Domain, 644
NormInMonogenicAlgebra
  Domain, 645
NORMMA, 645
NORMPK, 644
NORMRETR, 645
NormRetractPackage
  Domain, 645
NPCOEF, 645
NPCoef
  Domain, 645
NPOLYGON, 642
NREP, 652
NSUP2, 641
NTPOLFN, 647
NumberFieldIntegralBasis
  Domain, 646
NumberFormats
  Domain, 646
NumberTheoreticPolynomialFunctions
  Domain, 647
NUMERIC, 647
Numeric
  Domain, 647
NumericalOrdinaryDifferentialEquations
  Domain, 648
NumericalQuadrature
  Domain, 650
NumericComplexEigenPackage
  Domain, 651
NumericContinuedFraction
  Domain, 652
NumericRealEigenPackage
  Domain, 652
NumericTubePlot
  Domain, 653
NUMFMT, 646
NUMODE, 648
NUMQUAD, 650
NUMTUBE, 653

==== **O** ====

OCTCT2, 653
OctonionCategoryFunctions2
  Domain, 653
ODECONST, 517
ODEEF, 532
ODEINT, 654
ODEIntegration
  Domain, 654
ODEPACK, 501
ODEPAL, 693
ODEPRIM, 690
ODEPRRIC, 691
ODERAT, 705
ODERED, 711
ODERTRIC, 706
ODESYS, 730
ODETOOLS, 654
ODETools
  Domain, 654
OMEXPR, 543
OMPKG, 656
OMSERVER, 656
ONECOMP2, 655
OneDimensionalArrayFunctions2
  Domain, 655
OnePointCompletionFunctions2
  Domain, 655
OpenMathPackage
  Domain, 656
OpenMathServerPackage
  Domain, 656
OperationsQuery
  Domain, 657
OPQUERY, 657
OPTPACK, 500
ORDCOMP2, 657
OrderedCompletionFunctions2
  Domain, 657
OrderingFunctions
  Domain, 658
ORDFUNS, 658
OREPCTO, 750
OrthogonalPolynomialFunctions
  Domain, 658
ORTHPOL, 658
OUT, 659
OutputPackage
  Domain, 659

==== **P** ====

PackageForAlgebraicFunctionField
  Domain, 659
PackageForAlgebraicFunctionFieldOverFiniteField
  Domain, 660
PackageForPoly
  Domain, 660
PADE, 661
PadeApproximantPackage
  Domain, 660
PadeApproximants
  Domain, 661

PADEPAC, 660
PAdicWildFunctionFieldIntegralBasis
Domain, 661
PAFF, 659
PAFFFF, 660
PAN2EXPR, 680
ParadoxicalCombinatorsForStreams
Domain, 662
ParametricLinearEquations
Domain, 662
ParametricPlaneCurveFunctions2
Domain, 663
ParametricSpaceCurveFunctions2
Domain, 664
ParametricSurfaceFunctions2
Domain, 664
ParametrizationPackage
Domain, 664
PARAMP, 664
PARPC2, 663
PARSC2, 664
PARSU2, 664
PartialFractionPackage
Domain, 665
PartitionsAndPermutations
Domain, 665
PARTPERM, 665
PATMATCH, 667
PATRES2, 672
PATTERN1, 666
PATTERN2, 666
PatternFunctions1
Domain, 666
PatternFunctions2
Domain, 666
PatternMatch
Domain, 667
PatternMatchAssertions
Domain, 667
PatternMatchFunctionSpace
Domain, 668
PatternMatchIntegerNumberSystem
Domain, 668
PatternMatchIntegration
Domain, 669
PatternMatchKernel
Domain, 669
PatternMatchListAggregate
Domain, 670
PatternMatchPolynomialCategory
Domain, 670
PatternMatchPushDown
Domain, 671
PatternMatchQuotientFieldCategory
Domain, 671
PatternMatchResultFunctions2
Domain, 672
PatternMatchSymbol
Domain, 672
PatternMatchTools
Domain, 672
PCOMP, 681
PDECOMP, 682
PDEPACK, 501
PERMAN, 673
Permanent
Domain, 673
PermutationGroupExamples
Domain, 673
PFBR, 682
PFBRU, 683
PFO, 677
PFOQ, 677
PFORP, 660
PFOTOOLS, 678
PFRPAC, 665
PGCD, 684
PGE, 673
PGROEB, 679
PICOERCE, 674
PiCoercions
Domain, 674
PINTERP, 684
PINTERPA, 685
PLEQN, 662
PLOT1, 674
PlotFunctions1
Domain, 674
PLOTTOOL, 675
PlotTools
Domain, 675
PLPKCRV, 678
PMASS, 667
PMASSFS, 563
PMDOWN, 671
PMFS, 668
PMINS, 668
PMKERNEL, 669
PMLSAGG, 670
PMPLCAT, 670
PMPRED, 504
PMPREDFS, 563
PMQFCAT, 671
PMSYM, 672
PMTOOLS, 672
PNTHEORY, 685
PointFunctions2
Domain, 676

PointPackage
    Domain, 676
PointsOfFiniteOrder
    Domain, 677
PointsOfFiniteOrderRational
    Domain, 677
PointsOfFiniteOrderTools
    Domain, 678
POLTOPOL, 679
PolToPol
    Domain, 679
POLUTIL, 707
POLY2, 683
POLY2UP, 688
POLYCATQ, 681
PolyGroebner
    Domain, 679
POLYLIFT, 680
PolynomialAN2Expression
    Domain, 680
PolynomialCategoryLifting
    Domain, 680
PolynomialCategoryQuotientFunctions
    Domain, 681
PolynomialComposition
    Domain, 681
PolynomialDecomposition
    Domain, 682
PolynomialFactorizationByRecursion
    Domain, 682
PolynomialFactorizationByRecursionUnivariate
    Domain, 683
PolynomialFunctions2
    Domain, 683
PolynomialGcdPackage
    Domain, 684
PolynomialInterpolation
    Domain, 684
PolynomialInterpolationAlgorithms
    Domain, 685
PolynomialNumberTheoryFunctions
    Domain, 685
PolynomialPackageForCurve
    Domain, 678
PolynomialRoots
    Domain, 686
PolynomialSetUtilitiesPackage
    Domain, 686
PolynomialSolveByFormulas
    Domain, 687
PolynomialSquareFree
    Domain, 687
PolynomialToUnivariatePolynomial
    Domain, 688
POLYROOT, 686
POLYVEC, 753
PowerSeriesLimitPackage
    Domain, 688
PREASSOC, 689
PrecomputedAssociatedEquations
    Domain, 689
PRIMARR2, 689
PRIMELT, 690
PRIMES, 597
PrimitiveArrayFunctions2
    Domain, 689
PrimitiveElement
    Domain, 690
PrimitiveRatDE
    Domain, 690
PrimitiveRatRicDE
    Domain, 691
PRINT, 691
PrintPackage
    Domain, 691
PRJALGPK, 675
ProjectiveAlgebraicSetPackage
    Domain, 675
PRS, 692
PSETPK, 686
PSEUDLIN, 692
PseudoLinearNormalForm
    Domain, 692
PseudoRemainderSequence
    Domain, 692
PSQFR, 687
PTFUNC2, 676
PTPACK, 676
PureAlgebraicIntegration
    Domain, 693
PureAlgebraicLODE
    Domain, 693
PUSHVAR, 694
PushVariables
    Domain, 694
PWFFINTB, 661

==== **Q** ====

QALGSET2, 694
QCMPACK, 695
QFCAT2, 696
QuasiAlgebraicSet2
    Domain, 694
QuasiComponentPackage
    Domain, 695
QUATCT2, 696
QuaternionCategoryFunctions2

Domain, 696
QuotientFieldCategoryFunctions2
Domain, 696

==== **R** ====

RadicalEigenPackage
Domain, 697
RadicalSolvePackage
Domain, 697
RadixUtilities
Domain, 698
RADUTIL, 698
RandomDistributions
Domain, 698
RandomFloatDistributions
Domain, 699
RandomIntegerDistributions
Domain, 699
RandomNumberSource
Domain, 699
RANDSRC, 699
RATFACT, 700
RationalFactorize
Domain, 700
RationalFunction
Domain, 701
RationalFunctionDefiniteIntegration
Domain, 701
RationalFunctionFactor
Domain, 702
RationalFunctionFactorizer
Domain, 702
RationalFunctionIntegration
Domain, 703
RationalFunctionLimitPackage
Domain, 703
RationalFunctionSign
Domain, 703
RationalFunctionSum
Domain, 704
RationalIntegration
Domain, 704
RationalInterpolation
Domain, 705
RationalLODE
Domain, 705
RationalRetractions
Domain, 706
RationalRicDE
Domain, 706
RationalUnivariateRepresentationPackage
Domain, 707
RATRET, 706
RDEEF, 534
RDEEFS, 534
RDETR, 740
RDETRS, 740
RDIST, 698
RDIV, 710
REAL0, 708
REAL0Q, 709
RealPolynomialUtilitiesPackage
Domain, 707
REALSOLV, 708
RealSolvePackage
Domain, 708
RealZeroPackage
Domain, 708
RealZeroPackageQ
Domain, 709
RECOP, 710
RectangularMatrixCategoryFunctions2
Domain, 709
RecurrenceOperator
Domain, 710
REDORDER, 711
ReducedDivisor
Domain, 710
ReduceLODE
Domain, 711
ReductionOfOrder
Domain, 711
RegularSetDecompositionPackage
Domain, 712
RegularTriangularSetGcdPackage
Domain, 713
REP, 697
REP1, 714
REP2, 714
REPDB, 713
RepeatedDoubling
Domain, 713
RepeatedSquaring
Domain, 713
RepresentationPackage1
Domain, 714
RepresentationPackage2
Domain, 714
REPSQ, 713
RESLATC, 715
ResolveLatticeCompletion
Domain, 715
RetractSolvePackage
Domain, 716
RETSOL, 716
RF, 701
RFDIST, 699

RFFACT, 702
RFFACTOR, 702
RFP, 716
RIDIST, 699
RINTERP, 705
RMCAT2, 709
RootsFindingPackage
    Domain, 716
RSDCMPK, 712
RSETGCD, 713
RURPK, 707

==== **S** ====

SAEFACT, 719
SAERationalFunctionAlgFactor
    Domain, 717
SAERFFC, 717
SCACHE, 720
SCPKG, 728
ScriptFormulaFormat1
    Domain, 717
SEG2, 718
SEGBIND2, 718
SegmentBindingFunctions2
    Domain, 718
SegmentFunctions2
    Domain, 718
SFQCMPK, 722
SFRGCD, 723
SGCF, 731
SHP, 729
SIGNEF, 533
SIGNRF, 703
SIMPAN, 719
SimpleAlgebraicExtensionAlgFactor
    Domain, 719
SimplifyAlgebraicNumberConvertPackage
    Domain, 719
SMITH, 720
SmithNormalForm
    Domain, 720
SOLVEFOR, 687
SOLVERAD, 697
SOLVESER, 741
SOLVETRA, 741
SortedCache
    Domain, 720
SortPackage
    Domain, 721
SORTPAK, 721
SparseUnivariatePolynomialFunctions2
    Domain, 721
SpecialOutputPackage
    Domain, 722
SPECOUT, 722
SquareFreeQuasiComponentPackage
    Domain, 722
SquareFreeRegularSetDecompositionPackage
    Domain, 723
SquareFreeRegularTriangularSetGcdPackage
    Domain, 723
SRDCMPK, 723
STINPROD, 726
STNSR, 727
StorageEfficientMatrixOperations
    Domain, 724
STREAM1, 724
STREAM2, 725
STREAM3, 725
StreamFunctions1
    Domain, 724
StreamFunctions2
    Domain, 725
StreamFunctions3
    Domain, 725
StreamInfiniteProduct
    Domain, 726
StreamTaylorSeriesOperations
    Domain, 726
StreamTensor
    Domain, 727
StreamTranscendentalFunctions
    Domain, 727
StreamTranscendentalFunctionsNonCommutative
    Domain, 728
StructuralConstantsPackage
    Domain, 728
STTAYLOR, 726
STTF, 727
STTFNC, 728
SturmHabichtPackage
    Domain, 729
SUBRESP, 729
SubResultantPackage
    Domain, 729
SUMFS, 566
SUMRF, 704
SUP2, 721
SUPFRACF, 730
SupFractionFactorizer
    Domain, 730
SYMFUNC, 732
SymmetricFunctions
    Domain, 732
SymmetricGroupCombinatoricFunctions
    Domain, 731
SYSSOLP, 731

SystemODESolver
  Domain, 730
SystemSolvePackage
  Domain, 731

==== **T** ====

TABLBUMP, 732
TableauxBumpers
  Domain, 732
TabulatedComputationPackage
  Domain, 733
TANEXP, 733
TangentExpansions
  Domain, 733
TaylorSolve
  Domain, 734
TBCMPPK, 733
TemplateUtilities
  Domain, 734
TEMUTL, 734
TEX1, 735
TexFormat1
  Domain, 735
ToolsForSign
  Domain, 735
TOOLSIGN, 735
TopLevelDrawFunctions
  Domain, 736
TopLevelDrawFunctionsForAlgebraicCurves
  Domain, 736
TopLevelDrawFunctionsForCompiledFunctions
  Domain, 737
TopLevelDrawFunctionsForPoints
  Domain, 737
TopLevelThreeSpace
  Domain, 738
TOPSP, 738
TranscendentalHermiteIntegration
  Domain, 738
TranscendentalIntegration
  Domain, 739
TranscendentalManipulations
  Domain, 739
TranscendentalRischDE
  Domain, 740
TranscendentalRischDESystem
  Domain, 740
TransSolvePackage
  Domain, 741
TransSolvePackageService
  Domain, 741
TriangularMatrixOperations
  Domain, 742
TRIGMNIP, 742
TrigonometricManipulations
  Domain, 742
TRIMAT, 742
TRMANIP, 739
TubePlotTools
  Domain, 743
TUBETOOL, 743
TwoDimensionalPlotClipping
  Domain, 743
TWOFACT, 744
TwoFactorize
  Domain, 744

==== **U** ====

U32VectorPolynomialOperations
  Domain, 753
UDPO, 752
UDVO, 752
UFPS1, 745
ULS2, 745
UNIFACT, 744
UNISEG2, 752
UnivariateFactorize
  Domain, 744
UnivariateFormalPowerSeriesFunctions
  Domain, 745
UnivariateLaurentSeriesFunctions2
  Domain, 745
UnivariatePolynomialCategoryFunctions2
  Domain, 746
UnivariatePolynomialCommonDenominator
  Domain, 746
UnivariatePolynomialDecompositionPackage
  Domain, 747
UnivariatePolynomialDivisionPackage
  Domain, 747
UnivariatePolynomialFunctions2
  Domain, 748
UnivariatePolynomialMultiplicationPackage
  Domain, 748
UnivariatePolynomialSquareFree
  Domain, 749
UnivariatePuiseuxSeriesFunctions2
  Domain, 750
UnivariateSkewPolynomialCategoryOps
  Domain, 750
UnivariateTaylorSeriesFunctions2
  Domain, 751
UnivariateTaylorSeriesODESolver
  Domain, 751
UniversalSegmentFunctions2
  Domain, 752

UP2, 748
UPCDEN, 746
UPDECOMP, 747
UPDIVP, 747
UPMP, 748
UPOLYC2, 746
UPSQFREE, 749
UPXS2, 750
UserDefinedPartialOrdering
    Domain, 752
UserDefinedVariableOrdering
    Domain, 752
UTS2, 751
UTSODE, 751
UTSODETL, 753
UTSodetools
    Domain, 753
UTSSOL, 734

==== **V** ====

VECTOR2, 754
VectorFunctions2
    Domain, 754
VIEW, 755
VIEWDEF, 755
ViewDefaultsPackage
    Domain, 755
ViewportPackage
    Domain, 755

==== **W** ====

WEIER, 756
WeierstrassPreparation
    Domain, 756
WFFINTBS, 756
WildFunctionFieldIntegralBasis
    Domain, 756

==== **X** ====

XExponentialPackage
    Domain, 757
XIXPPKG, 757

==== **Y** ====

YSTREAM, 662

==== **Z** ====

ZDSOLVE, 757
ZeroDimensionalSolvePackage
    Domain, 757
ZLINDEP, 596