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

October 12, 2019

ba618b749b6abfcdd4172a46a582ab32b2b16da5

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
Christian Aistleitner
Jerry Archibald
Knut Bahr
Stephen Balzac
Thomas Baruchel
Michael Becker
David Bindel
Mark Botch
Karen Braman
Martin Brock
Stephen Buchwald
William Burge
Jacques Carette
Pablo Cayuela
Tzu-Yi Chen
David V. Chudnovsky
Roland Coeurjoly
Josh Cohen
Christophe Conil
Robert Corless
Meino Cramer
David Cyganski
Timothy Daly Jr.
James Demmel
Inderjit Dhillon
Gabriel Dos Reis
Nicolas James Doye
Iain Duff
Brian Dupee
Hans-Dieter Ehrich
Lars Erickson
Richard Fateman
John Fletcher
George Frances
Korrinn Fu
Van de Geijn
Gustavo Goertkin
Holger Gollan
Laureano Gonzalez-Vega
Matt Grayson
James Griesmer
Ming Gu
Gaetan Hache
Sven Hammarling
Richard Harke
Vilya Harvey
Dan Hatton
Ralf Hemmecke
Antoine Hersen
Alan Hoffman
Gernot Hueber
Alejandro Jakubi
William Kahan
Grant Keady
Tony Kennedy

Cyril Alberga
Richard Anderson
S.J. Atkins
Henry Baker
Yurij Baransky
Gerald Baumgartner
Nelson H. F. Beebe
Fred Blair
Raoul Bourquin
Wolfgang Brehm
Manuel Bronstein
Florian Bundschuh
Ralph Byers
Pierre Casteran
Bruce Char
Bobby Cheng
Gregory V. Chudnovsky
Emil Cohen
James Cloos
Don Coppersmith
Gary Cornell
Karl Crary
Nathaniel Daly
James H. Davenport
Didier Deshommes
Jack Dongarra
Claire DiCrescendo
Zlatko Drmac
Lee Duhem
Dominique Duval
Heow Eide-Goodman
Mark Fahey
Bertfried Fauser
Brian Ford
Constantine Frangos
Marc Gaetano
Kathy Gerber
Samantha Goldrich
Teresa Gomez-Diaz
Stephen Gortler
Martin Griss
Vladimir Grinberg
Fred Gustavson
Steve Hague
Mike Hansen
Joseph Harry
Martin Hassner
Waldek Hebisch
Tony Hearn
Nicholas J. Higham
Hoon Hong
Pietro Iglio
Richard Jenks
Kyriakos Kalorkoti
Tom Kelsey
David Kincaid

Roy Adler
George Andrews
Jeremy Avigad
Martin Baker
David R. Barton
Gilbert Baumslag
Jay Belanger
Vladimir Bondarenko
Alexandre Bouyer
Peter A. Broadbery
Christopher Brown
Luanne Burns
Quentin Carpent
Robert Cavines
Ondrej Certik
Cheekai Chin
Mark Clements
Hirsh Cohen
Jia Zhao Cong
George Corliss
Frank Costa
Jeremy Du Croz
Timothy Daly Sr.
David Day
Michael Dewar
Jean Della Dora
Sam Dooley
Lionel Ducos
Martin Dunstan
Robert Edwards
Carl Engelman
William Farmer
Stuart Feldman
Albrecht Fortenbacher
Timothy Freeman
Rudiger Gebauer
Patricia Gianni
Max Goldstein
Ralph Gomory
Johannes Grabmeier
Klaus Ebbe Grue
Oswald Gschnitzer
Jocelyn Guidry
Satoshi Hamaguchi
Richard Hanson
Bill Hart
Arthur S. Hathaway
Karl Hegbloom
Henderson
Lou Hodes
Roger House
Joan Jaffe
Bo Kagstrom
Kai Kaminski
Wilfrid Kendall
Keshav Kini

Knut Korsvold
Igor Kozachenko
Bernhard Kutzler
Kaj Laurson
Franz Lehner
Howard Levy
John Lipson
Michael Lucks
Dave Mainey
Ursula Martin
Bob McElrath
Edi Meier
Jonathan Millen
William Miranker
Michael Monagan
Joel Moses
Patrice Naudin
Godfrey Nolan
Michael O'Connor
Humberto Ortiz-Zuazaga
David Parnas
Michel Petitot
Frederick H. Pitts
E. Quintana-Orti
A. Petitet
Greg Puhak
Norman Ramsey
Huan Ren
Jason Riedy
Jean Rivlin
Raymond Rogers
Jeff Rutter
Alfred Scheerhorn
Martin Schoenert
Fritz Schwartz
Nick Simicich
Jacob Nyffeler Smith
Ken Stanley
Christine Sundaresan
Moss E. Sweedler
Max Tegmark
Laurent Thery
Carol Thompson
Steve Toleque
Barry Trager
Gregory Vanuxem
Bernhard Wall
Andreas Weber
Juergen Weiss
James Wen
R. Clint Whaley
Berhard Will
Shmuel Winograd
Waldemar Wiwianka
Liu Xiaojun
Qian Yun
Evelyn Zoernack

Ted Kosan
Fred Krogh
Tim Lahey
Charles Lawson
Frederic Lehobey
J. Lewis
Rudiger Loos
Richard Luczak
Francois Maltey
Osni Marques
Michael McGettrick
Ian Meikle
Victor S. Miller
Mohammed Mobarak
Marc Moreno-Maza
Mark Murray
C. Andrew Neff
Arthur Norman
Summat Oemrawsingh
Julian A. Padget
Norm Pass
Didier Pinchon
Frank Pfenning
Gregorio Quintana-Orti
Andre Platzer
Claude Quitte
Anatoly Raportirenko
Albert D. Rich
Renaud Rioboo
Nicolas Robidoux
Michael Rothstein
Philip Santas
William Schelter
Marshall Schor
Steven Segletes
William Sit
Matthieu Sozeau
Jonathan Steinbach
Klaus Sutner
Eugene Surowitz
T. Doug Telford
Balbir Thomas
Dylan Thurston
Dick Toupin
Hale Trotter
Kresimir Veselic
Paul Wang
Jaap Weel
M. Weller
Thorsten Werther
James T. Wheeler
Clifton J. Williamson
Robert Wisbauer
Knut Wolf
Clifford Yapp
Vadim Zhytnikov
Bruno Zuercher

Paul Kosinski
Klaus Kusche
Larry Lambe
George L. Legendre
Michel Levaud
Ren-Cang Li
Craig Lucas
Camm Maguire
William Martin
Alasdair McAndrew
Bob McNeill
David Mentre
Gerard Milmeister
H. Michael Moeller
Scott Morrison
William Naylor
John Nelder
Jinzhong Niu
Kostas Oikonomou
Bill Page
Susan Pelzel
Ayal Pinkus
Jose Alfredo Portes
Beresford Parlett
Peter Poromaas
Arthur C. Ralfs
Guilherme Reis
Michael Richardson
Robert Risch
Simon Robinson
Martin Rubey
David Saunders
Gerhard Schneider
Frithjof Schulze
V. Sima
Elena Smirnova
Srinivasan Seshan
Fabio Stumbo
Robert Sutor
Yong Kiam Tan
James Thatcher
Mike Thomas
Francoise Tisseur
Raymond Toy
Themos T. Tsikas
Christof Voemel
Stephen Watt
Al Weis
Mark Wegman
Michael Wester
John M. Wiley
Stephen Wilson
Sandra Wityak
Yanyang Xiao
David Yun
Richard Zippel
Dan Zwillinger

Contents

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))

Motivation

> **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

> **Beware of bugs in the above code. I have only proved it correct, not tried it.**
> – Donald Knuth

> **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 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

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. This gives a solid theoretical framework which 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.

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].

Why this effort won't succeed

Ron Pressler [Pres19] 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.
```

The Problem

1.1 A Brief History

Axiom[Daly17] is a computer algebra system. It was originally developed at IBM Research based on a proposal by James Griesmer[**?**] 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.

1.2 Parallel Development

1.3 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.

1.4 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.

1.5 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.

1.6 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.

1.7 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.

1.8 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 1.11 section for more detailed information.

1.9 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**), which 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.

1.10 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 [**?**]

> **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

— **ignore** —

```
(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.

— **sane** —

```
\getchunk{splitchar}
```

————-

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.

— **sane** —

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

———-

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

— **sane** —

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

———-

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

— **sane** —

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

———-

— **sane** —

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

———-

1.11 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

— **sane** —

```
(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)))

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

(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))))
```

———-

— **sane** —

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

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

———-

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

— **sane** —

```
(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)
   (arglist      :initarg :arg          :accessor arglist  :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.

— **sane** —

```
(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) (arglist cat)))
     ((eq (marker cat) 'domain)
       (format stream formatDomainAbbrev (abbreviation cat) (name cat))
       (format stream formatDomainComment (comment cat))
       (format stream formatDomainSIG (name cat) (arglist cat)))
     ((eq (marker cat) 'package)
       (format stream formatPackageAbbrev (abbreviation cat) (name cat))
       (format stream formatPackageComment (comment cat))
       (format stream formatPackageSIG (name cat) (arglist 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)))))
```

———-

— **sane** —

```
(defclass typeParam ()
  ((variable   :initarg :variable  :reader typeParam-variable)
   (paramtype  :initarg :paramtype :reader typeParam-paramtype)))

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

———-

— **sane** —

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

———-

— **sane** —

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

———-

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

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

— **sane** —

```
(defgeneric showSig (name))

(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))))
```

———-

1.12 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)) ...)
```

— **sane** —

```
(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)))
```

————-

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.

— **sane** —

```
(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<))))
```

———-

We need to compute the most immediate ancestors (the leaves) of the type tree.

— **sane** —

```
(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.

— **sane** —

```
(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.

— **sane** —

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

———-

1.12.1 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.

— **sane** —

```
(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)))

(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?))

(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))))

(defun signature-make (returns argumentlist)
  (make-instance 'signature
    :returns returns
    :arguments argumentlist))
```

———-

— **sane** —

```
(defclass amacro ()
  ((name
     :initarg :name
```

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

———-

— **sane** —

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

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

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

(defun operation-make (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")
```

— **sane** —

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

— **sane** —

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

(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

— **sane** —

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

———-

The Parser

Note that this function is inserted by the primitive support tools section above.

— **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))
```

————-

— **splitchar** —

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

————-

— **sane** —

```
(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)))
```

————-

— **sane** —

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

(defmethod print-object ((p ParserClass) stream)
  (format stream formatabbrev (parser-abbrev p))
  (format stream formattopcomment (parser-topcomment p))
  (loop for acons in (parser-macros p) do
     (format stream formatmacros (car acons) (cdr acons))))
```

1.13 The Language

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

UCHAR       ::= uppercase character
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
CONSTRUCT   ::= constructorname
ABBREV      ::= ')abbrev' ['category' | 'domain' | 'package'] UABBREV CONSTRUCT

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]
```

1.14 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.

— **sane** —

```
(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.

— **sane** —

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

———-

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

— **sane** —

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

———-

— **sane** —

```
(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 which 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 which 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)")
```

— **sane** —

```
(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** which 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"
```

— **sane** —

```
(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.

— **sane** —

```
(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.

— **sane** —

```
(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.

— **sane** —

```
(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.

— **sane** —

```
(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.

— **sane** —

```
(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.

— **sane** —

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

———-

— **sane** —

```
(defun FSM-gatherList (str)
  (let (result (depth 0) (more t) instring)
```

```
  (loop for c across str while more do
    (cond
      ; end of string?
      ((and instring (char= c #\"))
        (setq instring nil)
        (push c result))
      ; still in string?
      (instring
        (push c result))
      ; start string?
      ((and (not instring) (char= c #\"))
        (setq instring t)
        (push c result))
      ; start list?
      ((char= c #\()
        (incf depth)
        (push #\( result))
      ; exit nested list?
      ((and (> depth 0) (char= c #\)))
        (decf depth)
        (when (= depth 0) (setq more nil))
        (push #\) result))
      ; end list?
      ((and (= depth 0) (char= c #\)))
        (push #\) result)
        (setq more nil))
      (t
        (push c result))))
  (unless more
    (coerce (nreverse result) 'string))))
```

———-

1.15 Reading the source file

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

— **sane** —

```
(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)))
```

———-

— **sane** —

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

———-

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

1.16 Sourcecode Tree Utilities

We process the original source code into a "clean form" which 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.

```
STRING -> DEPTH
```

— **sane** —

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

———-

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
```

— **sane** —

```
(defun 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
```

— **sane** —

```
(defun depthof (node) (car node))
```

————-

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

```
   BIRTHFORM -> ANY
```

— **sane** —

```
(defun 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
```

— **sane** —

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

————-

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 nexted nodes first.

```
   SPAWNFORM -> LIST DEPTH
```

— **sane** —

```
(defun deep (tree)
  (let (result)
    (loop for node in tree do
      (setq result (adjoin (depthof node) result)))
   (sort result #'>)))
```

————-

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 which groups the code into a tree form, called a **sourcecode tree** 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)"
 "    ))")
```

— sane —

```
;  PILEFORM -> TREEFORM
(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)))
    ; 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) (string-trim '(#\space) (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.

```
TREEFORM -> VOID
```

— **sane** —

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

———-

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 signature is

```
FILENAME -> SOURCECODE
```

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 which 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.

— **sane** —

```
; FILENAME -> SOURCECODE
(defun make-Sourcecode (filename)
  (labels (
    ; LINE -> BOOLEAN
    (blankline? (line)
      (string= "" (string-trim '(#\space) line)))
    ; 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)
      (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)
            (push (noComments line) listLine))
      (make-instance 'sourcecode :name filename
                                 :rawcode (nreverse rawcode)
                                 :pile pile
                                 :tree (pile2tree pile)))))))
```

———-

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]
```

ϵ s_1 s_2

s_1

ϵ

s_2

s_3

s_4

A signature may also have a
machine.
The **look** function just peeks
The **next** function updates the
The **here** function outputs deb

```
       (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)
))))
```

———-

1.16.1 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.

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

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

— **sane** —

```
(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)))

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

(defmethod print-object ((abb abbreviation) stream)
  (format stream ")abbrev ~a ~a ~a~%"
    (abbreviation-class abb)
    (abbreviation-abbrev abb)
    (abbreviation-string abb)))

(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))))))
```

———-

1.16.2 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.

— **sane** —

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

————-

1.16.3 Gather Macros

— **sane** —

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

————-

— **sane** —

```
(defun FSM-macros (linelist)
  (labels (
    (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)
       (setq alllines (gather-lines gather))
       (setq line (splitchar (string-trim '(#\space) (pop alllines)) #\space))
       (setq name (intern (first line)))
       (setq def (cons (join (cddr line)) alllines))
       (cons name def)))
  )
  (let (macro environ)
    (loop while (hasMacros? linelist) do
      (multiple-value-setq (linelist macro) (FSM-extract1Macro linelist))
      (push (makeAlist macro) environ))
```

```
    (values linelist environ))))
```

———-

— sane —

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

———-

1.16.4 parseSignature

— sane —

```
(defun parseSignature (lines)
  (let (funcname return arglist comment colon arrow line control plusplus)
    (labels (
      (substring (line start &optional end)
        (string-trim '(#\space) (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 lines))
    ; 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 arglist  (mangle (substring line (1+ colon) arrow)))
    (setq return   (mangle (substring line (+ 2 arrow))))
    ; collect the comment lines
    (dolist (line (cdr lines))
```

```
      (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= arglist "()")
         (setq control (concatenate 'string control " '~a ")))
       ((string= arglist "%")
         (setq control (concatenate 'string control " '(~a) ")))
       (t
         (setq control (concatenate 'string control " '(|~a|)"))))
    (setq control (concatenate 'string control " '~s)~%"))
    (format t control funcname return arglist (nreverse comment))
    (values)))
```

———-

1.17 The Parse Function

— **sane** —

```
(defvar place "/research/20190531/src/algebra/")
```

———-

— **sane** —

```
(defvar t1 nil)
```

———-

— **sane** —

```
(defvar theparse nil)

(defun Parse (filename)
  (let (pile object alist)
    (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))
    (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)
    (values theparse pile)))

(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))
```

———-

The Categories

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.

1.18 Level 1

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

— **sane** —

```
(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|))
```

———-

1.18.1 AdditiveValuationAttribute

— **sane** —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.2 ApproximateAttribute

— **sane** —

```
(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|)))
```

———-

1.18.3 ArbitraryExponentAttribute

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.4 ArbitraryPrecisionAttribute

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.5 ArcHyperbolicFunctionCategory

— **sane** —

```
(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"))
   (arglist :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|)))
```

———-

1.18.6 ArcTrigonometricFunctionCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.18.7 AttributeRegistry

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.18.8 BasicType

— sane —

```
(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)."))
   (arglist :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 =
getdatabase call: BasicType&           PARENTS
PARENTS: NIL
Value = NIL
```

1.18.9 CanonicalAttribute

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.10 CanonicalClosedAttribute

— **sane** —

```
(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)}"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.11 CanonicalUnitNormalAttribute

— **sane** —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.12 CentralAttribute

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.13 CoercibleTo

— **sane** —

```
(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."))
   (arglist :initform (list (make-instance 'typeParam :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|)))
```

———-

1.18.14 CombinatorialFunctionCategory

— sane —

```
(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;"))
   (arglist :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|)))
```

———-

1.18.15 CommutativeStarAttribute

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.16 ConvertibleTo

— **sane** —

```
(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."))
   (arglist :initform (list (make-instance 'typeParam :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|)))
```

———-

1.18.17 ElementaryFunctionCategory

— **sane** —

```
(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;"))
   (arglist :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|)))
```

———-

1.18.18 Eltable

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.18.19 FiniteAggregateAttribute

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteAggregateAttribute|
  (progn
```

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

———-

1.18.20 HyperbolicFunctionCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.18.21 InnerEvalable

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.18.22 JacobiIdentityAttribute

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.23 LazyRepresentationAttribute

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.24 LeftUnitaryAttribute

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.25 ModularAlgebraicGcdOperations

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.18.26 MultiplicativeValuationAttribute

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.27 NoZeroDivisorsAttribute

— sane —

```
(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."))
   (arglist :initform nil)
```

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

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

———-

1.18.28 NotherianAttribute

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.29 NullSquareAttribute

— sane —

```
(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."))
   (arglist :initform nil)
```

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

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

———-

1.18.30 OpenMath

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.18.31 PartialTranscendentalFunctions

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.18.32 PartiallyOrderedSetAttribute

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartiallyOrderedSetAttribute|
  (progn
```

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

———-

1.18.33 PrimitiveFunctionCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.18.34 RadicalCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.18.35 RetractableTo

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.18.36 RightUnitaryAttribute

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.37 ShallowlyMutableAttribute

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ShallowlyMutableAttribute|
  (progn
```

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

———-

1.18.38 SpecialFunctionCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.18.39 TrigonometricFunctionCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.18.40 Type

— sane —

```
(defclass |TypeType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Type")
```

```
   (marker :initform 'category)
   (level :initform 1)
   (abbreviation :initform 'TYPE)
   (comment :initform (list
     "The fundamental Type."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.18.41 UnitsKnownAttribute

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.19 Level 2

— **sane** —

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

———-

1.19.1 Aggregate

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.19.2 CombinatorialOpsCategory

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.19.3 EltableAggregate

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.19.4 Evalable

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.19.5 FortranProgramCategory

— sane —

```
(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."))
  (arglist :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|)))
```

———-

1.19.6 FullyRetractableTo

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.19.7 Logic

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.19.8 Patternable

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Patternable|
  (progn
```

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

———-

1.19.9 PlottablePlaneCurveCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.19.10 PlottableSpaceCurveCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.19.11 RealConstant

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

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

———-

1.19.12 SegmentCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.19.13 SetCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.19.14 TranscendentalFunctionCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.20 Level 3

— **sane** —

```
(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|))
```

———-

1.20.1 AbelianSemiGroup

— **sane** —

```
(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 +."))
   (arglist :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|)))
```

———-

1.20.2 BlowUpMethodCategory

— **sane** —

```
(defclass |BlowUpMethodCategoryType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "BlowUpMethodCategory")
   (marker :initform 'category)
   (level :initform 3)
   (abbreviation :initform 'BLMETCT)
   (comment :initform nil)
   (arglist :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|)))
```

———-

1.20.3 Comparable

— **sane** —

```
(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)."))
   (arglist :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|)))
```

———-

1.20.4 FileCategory

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.20.5 FileNameCategory

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.20.6 Finite

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.7 FortranFunctionCategory

— **sane** —

```
(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."))
   (arglist :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 For
     (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|)))
```

———-

1.20.8 FortranMatrixCategory

— **sane** —

```
(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."))
  (arglist :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|)))
```

———-

1.20.9 FortranMatrixFunctionCategory

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.20.10 FortranVectorCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.11 FortranVectorFunctionCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.12 FullyEvalableOver

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.20.13 GradedModule

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.14 HomogeneousAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HomogeneousAggregate|
  (progn
    (push '|HomogeneousAggregate| *Categories*)
    (make-instance '|HomogeneousAggregateType|)))
```

———-

1.20.15 IndexedDirectProductCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.16 LiouvillianFunctionCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.20.17 Monad

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.20.18 NumericalIntegrationCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalIntegrationCategory|
  (progn
    (push '|NumericalIntegrationCategory| *Categories*)
    (make-instance '|NumericalIntegrationCategoryType|)))
```

———-

1.20.19 NumericalOptimizationCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalOptimizationCategory|
  (progn
    (push '|NumericalOptimizationCategory| *Categories*)
    (make-instance '|NumericalOptimizationCategoryType|)))
```

———-

1.20.20 OrderedSet

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.21 OrdinaryDifferentialEquationsSolverCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdinaryDifferentialEquationsSolverCategory|
  (progn
    (push '|OrdinaryDifferentialEquationsSolverCategory| *Categories*)
    (make-instance '|OrdinaryDifferentialEquationsSolverCategoryType|)))
```

———-

1.20.22 PartialDifferentialEquationsSolverCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialDifferentialEquationsSolverCategory|
  (progn
    (push '|PartialDifferentialEquationsSolverCategory| *Categories*)
    (make-instance '|PartialDifferentialEquationsSolverCategoryType|)))
```

———-

1.20.23 PatternMatchable

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.20.24 RealRootCharacterizationCategory

— sane —

```
(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"))
   (arglist :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|)))
```

———-

1.20.25 SExpressionCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.26 SegmentExpansionCategory

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.27 SemiGroup

— **sane** —

```
(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 *."))
   (arglist :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|)))
```

———-

1.20.28 SetCategoryWithDegree

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.29 StepThrough

— sane —

```
(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."))
   (arglist :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|)))
```

———-

1.20.30 ThreeSpaceCategory

— **sane** —

```
(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."))
   (arglist :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|)))
```

———-

1.21 Level 4

— **sane** —

```
(defvar level4
 '(|AbelianMonoid| |AffineSpaceCategory| |BagAggregate| |CachableSet|
   |Collection| |DifferentialVariableCategory| |ExpressionSpace|
   |FullyPatternMatchable| |GradedAlgebra| |IndexedAggregate|
   |InfinitlyClosePointCategory| |MonadWithUnit| |Monoid|
   |OrderedAbelianSemiGroup| |OrderedFinite| |PlacesCategory|
   |ProjectiveSpaceCategory| |RecursiveAggregate| |TwoDimensionalArrayCategory|))
```

———-

1.21.1 AbelianMonoid

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AbelianMonoid|
  (progn
    (push '|AbelianMonoid| *Categories*)
    (make-instance '|AbelianMonoidType|)))
```

———-

1.21.2 AffineSpaceCategory

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffineSpaceCategory|
  (progn
    (push '|AffineSpaceCategory| *Categories*)
    (make-instance '|AffineSpaceCategoryType|)))
```

———-

1.21.3 BagAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BagAggregate|
  (progn
    (push '|BagAggregate| *Categories*)
    (make-instance '|BagAggregateType|)))
```

———-

1.21.4 CachableSet

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CachableSet|
  (progn
    (push '|CachableSet| *Categories*)
    (make-instance '|CachableSetType|)))
```

———-

1.21.5 Collection

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Collection|
  (progn
    (push '|Collection| *Categories*)
    (make-instance '|CollectionType|)))
```

———-

1.21.6 DifferentialVariableCategory

— sane —

```
(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."))
  (arglist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |DifferentialVariableCategory|
  (progn
    (push '|DifferentialVariableCategory| *Categories*)
    (make-instance '|DifferentialVariableCategoryType|)))
```

———-

1.21.7 ExpressionSpace

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpace|
  (progn
    (push '|ExpressionSpace| *Categories*)
    (make-instance '|ExpressionSpaceType|)))
```

———-

1.21.8 FullyPatternMatchable

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullyPatternMatchable|
  (progn
    (push '|FullyPatternMatchable| *Categories*)
    (make-instance '|FullyPatternMatchableType|)))
```

———-

1.21.9 GradedAlgebra

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GradedAlgebra|
  (progn
    (push '|GradedAlgebra| *Categories*)
    (make-instance '|GradedAlgebraType|)))
```

———-

1.21.10 IndexedAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedAggregate|
  (progn
    (push '|IndexedAggregate| *Categories*)
    (make-instance '|IndexedAggregateType|)))
```

———-

1.21.11 InfinitlyClosePointCategory

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfinitlyClosePointCategory|
  (progn
    (push '|InfinitlyClosePointCategory| *Categories*)
    (make-instance '|InfinitlyClosePointCategoryType|)))
```

———-

1.21.12 MonadWithUnit

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonadWithUnit|
  (progn
    (push '|MonadWithUnit| *Categories*)
    (make-instance '|MonadWithUnitType|)))
```

———-

1.21.13 Monoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Monoid|
  (progn
    (push '|Monoid| *Categories*)
    (make-instance '|MonoidType|)))
```

———-

1.21.14 OrderedAbelianSemiGroup

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianSemiGroup|
  (progn
```

```
    (push '|OrderedAbelianSemiGroup| *Categories*)
    (make-instance '|OrderedAbelianSemiGroupType|)))
```

———-

1.21.15 OrderedFinite

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedFinite|
  (progn
    (push '|OrderedFinite| *Categories*)
    (make-instance '|OrderedFiniteType|)))
```

———-

1.21.16 PlacesCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlacesCategory|
  (progn
```

```
    (push '|PlacesCategory| *Categories*)
    (make-instance '|PlacesCategoryType|)))
```

———-

1.21.17 ProjectiveSpaceCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectiveSpaceCategory|
  (progn
    (push '|ProjectiveSpaceCategory| *Categories*)
    (make-instance '|ProjectiveSpaceCategoryType|)))
```

———-

1.21.18 RecursiveAggregate

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |RecursiveAggregate|
  (progn
    (push '|RecursiveAggregate| *Categories*)
    (make-instance '|RecursiveAggregateType|)))
```

———-

1.21.19 TwoDimensionalArrayCategory

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoDimensionalArrayCategory|
  (progn
    (push '|TwoDimensionalArrayCategory| *Categories*)
    (make-instance '|TwoDimensionalArrayCategoryType|)))
```

———-

1.22 Level 5

— **sane** —

```
(defvar level5
 '(|BinaryRecursiveAggregate| |CancellationAbelianMonoid| |DesingTreeCategory|
   |DoublyLinkedAggregate| |Group| |LinearAggregate| |MatrixCategory|
   |OrderedAbelianMonoid| |OrderedMonoid| |PolynomialSetCategory|
   |PriorityQueueAggregate| |QueueAggregate| |SetAggregate| |StackAggregate|
   |UnaryRecursiveAggregate|))
```

———-

1.22.1 BinaryRecursiveAggregate

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryRecursiveAggregate|
  (progn
    (push '|BinaryRecursiveAggregate| *Categories*)
    (make-instance '|BinaryRecursiveAggregateType|)))
```

———-

1.22.2 CancellationAbelianMonoid

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CancellationAbelianMonoid|
  (progn
    (push '|CancellationAbelianMonoid| *Categories*)
```

```
    (make-instance '|CancellationAbelianMonoidType|)))
```

———-

1.22.3 DesingTreeCategory

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DesingTreeCategory|
  (progn
    (push '|DesingTreeCategory| *Categories*)
    (make-instance '|DesingTreeCategoryType|)))
```

———-

1.22.4 DoublyLinkedAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoublyLinkedAggregate|
```

```
  (progn
    (push '|DoublyLinkedAggregate| *Categories*)
    (make-instance '|DoublyLinkedAggregateType|)))
```

———-

1.22.5 Group

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Group|
  (progn
    (push '|Group| *Categories*)
    (make-instance '|GroupType|)))
```

———-

1.22.6 LinearAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearAggregate|
  (progn
    (push '|LinearAggregate| *Categories*)
    (make-instance '|LinearAggregateType|)))
```

———-

1.22.7 MatrixCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixCategory|
  (progn
    (push '|MatrixCategory| *Categories*)
    (make-instance '|MatrixCategoryType|)))
```

———-

1.22.8 OrderedAbelianMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianMonoid|
  (progn
    (push '|OrderedAbelianMonoid| *Categories*)
    (make-instance '|OrderedAbelianMonoidType|)))
```

———-

1.22.9 OrderedMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedMonoid|
  (progn
    (push '|OrderedMonoid| *Categories*)
    (make-instance '|OrderedMonoidType|)))
```

———-

1.22.10 PolynomialSetCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialSetCategory|
  (progn
    (push '|PolynomialSetCategory| *Categories*)
    (make-instance '|PolynomialSetCategoryType|)))
```

———-

1.22.11 PriorityQueueAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PriorityQueueAggregate|
  (progn
    (push '|PriorityQueueAggregate| *Categories*)
    (make-instance '|PriorityQueueAggregateType|)))
```

———-

1.22.12 QueueAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QueueAggregate|
  (progn
    (push '|QueueAggregate| *Categories*)
    (make-instance '|QueueAggregateType|)))
```

———-

1.22.13 SetAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SetAggregate|
  (progn
    (push '|SetAggregate| *Categories*)
    (make-instance '|SetAggregateType|)))
```

———-

1.22.14 StackAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StackAggregate|
  (progn
    (push '|StackAggregate| *Categories*)
    (make-instance '|StackAggregateType|)))
```

———-

1.22.15 UnaryRecursiveAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnaryRecursiveAggregate|
  (progn
    (push '|UnaryRecursiveAggregate| *Categories*)
    (make-instance '|UnaryRecursiveAggregateType|)))
```

———-

1.23 Level 6

— **sane** —

```
(defvar level6
 '(|AbelianGroup| |BinaryTreeCategory| |DequeueAggregate| |DictionaryOperations|
   |ExtensibleLinearAggregate| |FiniteLinearAggregate|
   |FreeAbelianMonoidCategory|
   |OrderedCancellationAbelianMonoid| |PermutationCategory| |StreamAggregate|
   |TriangularSetCategory|))
```

———-

1.23.1 AbelianGroup

— **sane** —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |AbelianGroup|
  (progn
    (push '|AbelianGroup| *Categories*)
    (make-instance '|AbelianGroupType|)))
```

———-

1.23.2 BinaryTreeCategory

— sane —

```
(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. "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryTreeCategory|
  (progn
    (push '|BinaryTreeCategory| *Categories*)
    (make-instance '|BinaryTreeCategoryType|)))
```

———-

1.23.3 DequeueAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DequeueAggregate|
  (progn
    (push '|DequeueAggregate| *Categories*)
    (make-instance '|DequeueAggregateType|)))
```

———-

1.23.4 DictionaryOperations

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DictionaryOperations|
  (progn
    (push '|DictionaryOperations| *Categories*)
    (make-instance '|DictionaryOperationsType|)))
```

———-

1.23.5 ExtensibleLinearAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExtensibleLinearAggregate|
  (progn
    (push '|ExtensibleLinearAggregate| *Categories*)
    (make-instance '|ExtensibleLinearAggregateType|)))
```

———-

1.23.6 FiniteLinearAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteLinearAggregate|
  (progn
    (push '|FiniteLinearAggregate| *Categories*)
    (make-instance '|FiniteLinearAggregateType|)))
```

———-

1.23.7 FreeAbelianMonoidCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeAbelianMonoidCategory|
  (progn
    (push '|FreeAbelianMonoidCategory| *Categories*)
    (make-instance '|FreeAbelianMonoidCategoryType|)))
```

———-

1.23.8 OrderedCancellationAbelianMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedCancellationAbelianMonoid|
  (progn
    (push '|OrderedCancellationAbelianMonoid| *Categories*)
```

```
    (make-instance '|OrderedCancellationAbelianMonoidType|)))
```

———-

1.23.9 PermutationCategory

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PermutationCategory|
  (progn
    (push '|PermutationCategory| *Categories*)
    (make-instance '|PermutationCategoryType|)))
```

———-

1.23.10 StreamAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |StreamAggregate|
  (progn
    (push '|StreamAggregate| *Categories*)
    (make-instance '|StreamAggregateType|)))
```

———-

1.23.11 TriangularSetCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TriangularSetCategory|
  (progn
    (push '|TriangularSetCategory| *Categories*)
    (make-instance '|TriangularSetCategoryType|)))
```

———-

1.24 Level 7

— sane —

```
(defvar level7
 '(|Dictionary| |FiniteDivisorCategory| |LazyStreamAggregate| |LeftModule|
   |ListAggregate| |MultiDictionary| |MultisetAggregate| |NonAssociativeRng|
   |OneDimensionalArrayAggregate| |OrderedAbelianGroup| |OrderedAbelianMonoidSup|
   |RegularTriangularSetCategory| |RightModule| |Rng|))
```

———-

1.24.1 Dictionary

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Dictionary|
  (progn
    (push '|Dictionary| *Categories*)
    (make-instance '|DictionaryType|)))
```

———-

1.24.2 FiniteDivisorCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteDivisorCategory|
  (progn
    (push '|FiniteDivisorCategory| *Categories*)
    (make-instance '|FiniteDivisorCategoryType|)))
```

———-

1.24.3 LazyStreamAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LazyStreamAggregate|
  (progn
    (push '|LazyStreamAggregate| *Categories*)
    (make-instance '|LazyStreamAggregateType|)))
```

———-

1.24.4 LeftModule

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeftModule|
  (progn
    (push '|LeftModule| *Categories*)
    (make-instance '|LeftModuleType|)))
```

———-

1.24.5 ListAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |ListAggregate|
  (progn
    (push '|ListAggregate| *Categories*)
    (make-instance '|ListAggregateType|)))
```

———-

1.24.6 MultiDictionary

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultiDictionary|
  (progn
    (push '|MultiDictionary| *Categories*)
    (make-instance '|MultiDictionaryType|)))
```

———-

1.24.7 MultisetAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |MultisetAggregate|
  (progn
    (push '|MultisetAggregate| *Categories*)
    (make-instance '|MultisetAggregateType|)))
```

———-

1.24.8 NonAssociativeRng

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonAssociativeRng|
  (progn
    (push '|NonAssociativeRng| *Categories*)
    (make-instance '|NonAssociativeRngType|)))
```

———-

1.24.9 OneDimensionalArrayAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OneDimensionalArrayAggregate|
  (progn
    (push '|OneDimensionalArrayAggregate| *Categories*)
    (make-instance '|OneDimensionalArrayAggregateType|)))
```

———-

1.24.10 OrderedAbelianGroup

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianGroup|
  (progn
    (push '|OrderedAbelianGroup| *Categories*)
    (make-instance '|OrderedAbelianGroupType|)))
```

———-

1.24.11 OrderedAbelianMonoidSup

— sane —

```
(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')."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedAbelianMonoidSup|
  (progn
    (push '|OrderedAbelianMonoidSup| *Categories*)
    (make-instance '|OrderedAbelianMonoidSupType|)))
```

———-

1.24.12 RegularTriangularSetCategory

— sane —

```
(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."))
  (arglist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |RegularTriangularSetCategory|
  (progn
    (push '|RegularTriangularSetCategory| *Categories*)
    (make-instance '|RegularTriangularSetCategoryType|)))
```

———-

1.24.13 RightModule

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RightModule|
  (progn
    (push '|RightModule| *Categories*)
    (make-instance '|RightModuleType|)))
```

———-

1.24.14 Rng

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Rng|
  (progn
    (push '|Rng| *Categories*)
    (make-instance '|RngType|)))
```

———-

1.25 Level 8

— **sane** —

```
(defvar level8
 '(|BiModule| |BitAggregate| |FiniteSetAggregate| |KeyedDictionary|
   |NonAssociativeRing| |NormalizedTriangularSetCategory|
```

```
   |OrderedMultisetAggregate| |Ring| |SquareFreeRegularTriangularSetCategory|
   |StringAggregate| |VectorCategory|))
```

———-

1.25.1 BiModule

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BiModule|
  (progn
    (push '|BiModule| *Categories*)
    (make-instance '|BiModuleType|)))
```

———-

1.25.2 BitAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |BitAggregate|
  (progn
    (push '|BitAggregate| *Categories*)
    (make-instance '|BitAggregateType|)))
```

———-

1.25.3 FiniteSetAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteSetAggregate|
  (progn
    (push '|FiniteSetAggregate| *Categories*)
    (make-instance '|FiniteSetAggregateType|)))
```

———-

1.25.4 KeyedDictionary

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |KeyedDictionary|
  (progn
    (push '|KeyedDictionary| *Categories*)
    (make-instance '|KeyedDictionaryType|)))
```

———-

1.25.5 NonAssociativeRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonAssociativeRing|
  (progn
    (push '|NonAssociativeRing| *Categories*)
    (make-instance '|NonAssociativeRingType|)))
```

———-

1.25.6 NormalizedTriangularSetCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormalizedTriangularSetCategory|
  (progn
    (push '|NormalizedTriangularSetCategory| *Categories*)
    (make-instance '|NormalizedTriangularSetCategoryType|)))
```

———-

1.25.7 OrderedMultisetAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedMultisetAggregate|
  (progn
    (push '|OrderedMultisetAggregate| *Categories*)
    (make-instance '|OrderedMultisetAggregateType|)))
```

———-

1.25.8 Ring

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Ring|
  (progn
    (push '|Ring| *Categories*)
    (make-instance '|RingType|)))
```

———-

1.25.9 SquareFreeRegularTriangularSetCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularTriangularSetCategory|
  (progn
    (push '|SquareFreeRegularTriangularSetCategory| *Categories*)
    (make-instance '|SquareFreeRegularTriangularSetCategoryType|)))
```

———-

1.25.10 StringAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StringAggregate|
  (progn
    (push '|StringAggregate| *Categories*)
    (make-instance '|StringAggregateType|)))
```

———-

1.25.11 VectorCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |VectorCategory|
  (progn
```

```
    (push '|VectorCategory| *Categories*)
    (make-instance '|VectorCategoryType|)))
```

———-

1.26 Level 9

— sane —

```
(defvar level9
 '(|CharacteristicNonZero| |CharacteristicZero| |CommutativeRing|
   |DifferentialRing| |EntireRing| |LeftAlgebra| |LinearlyExplicitRingOver|
   |Module| |OrderedRing| |PartialDifferentialRing| |PointCategory|
   |SquareFreeNormalizedTriangularSetCategory| |StringCategory| |TableAggregate|))
```

———-

1.26.1 CharacteristicNonZero

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicNonZero|
  (progn
    (push '|CharacteristicNonZero| *Categories*)
    (make-instance '|CharacteristicNonZeroType|)))
```

———-

1.26.2 CharacteristicZero

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicZero|
  (progn
    (push '|CharacteristicZero| *Categories*)
    (make-instance '|CharacteristicZeroType|)))
```

———-

1.26.3 CommutativeRing

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommutativeRing|
  (progn
    (push '|CommutativeRing| *Categories*)
    (make-instance '|CommutativeRingType|)))
```

———-

1.26.4 DifferentialRing

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialRing|
  (progn
    (push '|DifferentialRing| *Categories*)
    (make-instance '|DifferentialRingType|)))
```

———-

1.26.5 EntireRing

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EntireRing|
  (progn
    (push '|EntireRing| *Categories*)
    (make-instance '|EntireRingType|)))
```

———-

1.26.6 LeftAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeftAlgebra|
  (progn
    (push '|LeftAlgebra| *Categories*)
    (make-instance '|LeftAlgebraType|)))
```

———-

1.26.7 LinearlyExplicitRingOver

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearlyExplicitRingOver|
  (progn
    (push '|LinearlyExplicitRingOver| *Categories*)
    (make-instance '|LinearlyExplicitRingOverType|)))
```

———-

1.26.8 Module

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Module|
  (progn
    (push '|Module| *Categories*)
    (make-instance '|ModuleType|)))
```

———-

1.26.9 OrderedRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedRing|
  (progn
    (push '|OrderedRing| *Categories*)
    (make-instance '|OrderedRingType|)))
```

———-

1.26.10 PartialDifferentialRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialDifferentialRing|
  (progn
    (push '|PartialDifferentialRing| *Categories*)
    (make-instance '|PartialDifferentialRingType|)))
```

———-

1.26.11 PointCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointCategory|
  (progn
    (push '|PointCategory| *Categories*)
    (make-instance '|PointCategoryType|)))
```

———-

1.26.12 SquareFreeNormalizedTriangularSetCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeNormalizedTriangularSetCategory|
  (progn
    (push '|SquareFreeNormalizedTriangularSetCategory| *Categories*)
    (make-instance '|SquareFreeNormalizedTriangularSetCategoryType|)))
```

———-

1.26.13 StringCategory

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StringCategory|
  (progn
    (push '|StringCategory| *Categories*)
```

```
    (make-instance '|StringCategoryType|)))
```

———-

1.26.14 TableAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TableAggregate|
  (progn
    (push '|TableAggregate| *Categories*)
    (make-instance '|TableAggregateType|)))
```

———-

1.27 Level 10

— sane —

```
(defvar level10
 '(|Algebra| |AssociationListAggregate| |DifferentialExtension| |DivisorCategory|
   |FreeModuleCat| |FullyLinearlyExplicitRingOver| |LeftOreRing| |LieAlgebra|
   |NonAssociativeAlgebra| |RectangularMatrixCategory| |VectorSpace|))
```

———-

1.27.1 Algebra

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Algebra|
  (progn
    (push '|Algebra| *Categories*)
    (make-instance '|AlgebraType|)))
```

———-

1.27.2 AssociationListAggregate

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociationListAggregate|
  (progn
    (push '|AssociationListAggregate| *Categories*)
    (make-instance '|AssociationListAggregateType|)))
```

———-

1.27.3 DifferentialExtension

— **sane** —

```
(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 %."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialExtension|
  (progn
    (push '|DifferentialExtension| *Categories*)
    (make-instance '|DifferentialExtensionType|)))
```

———-

1.27.4 DivisorCategory

— **sane** —

```
(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 "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DivisorCategory|
  (progn
    (push '|DivisorCategory| *Categories*)
    (make-instance '|DivisorCategoryType|)))
```

———-

1.27.5 FreeModuleCat

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeModuleCat|
  (progn
    (push '|FreeModuleCat| *Categories*)
    (make-instance '|FreeModuleCatType|)))
```

———-

1.27.6 FullyLinearlyExplicitRingOver

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullyLinearlyExplicitRingOver|
  (progn
```

```
    (push '|FullyLinearlyExplicitRingOver| *Categories*)
    (make-instance '|FullyLinearlyExplicitRingOverType|)))
```

———-

1.27.7 LeftOreRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeftOreRing|
  (progn
    (push '|LeftOreRing| *Categories*)
    (make-instance '|LeftOreRingType|)))
```

———-

1.27.8 LieAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |LieAlgebra|
  (progn
    (push '|LieAlgebra| *Categories*)
    (make-instance '|LieAlgebraType|)))
```

———-

1.27.9 NonAssociativeAlgebra

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonAssociativeAlgebra|
  (progn
    (push '|NonAssociativeAlgebra| *Categories*)
    (make-instance '|NonAssociativeAlgebraType|)))
```

———-

1.27.10 RectangularMatrixCategory

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |RectangularMatrixCategory|
  (progn
    (push '|RectangularMatrixCategory| *Categories*)
    (make-instance '|RectangularMatrixCategoryType|)))
```

———-

1.27.11 VectorSpace

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |VectorSpace|
  (progn
    (push '|VectorSpace| *Categories*)
    (make-instance '|VectorSpaceType|)))
```

———-

1.28 Level 11

— **sane** —

```
(defvar level11
 '(|DirectProductCategory| |DivisionRing| |FiniteRankAlgebra|
   |FiniteRankNonAssociativeAlgebra| |FreeLieAlgebra| |IntegralDomain|
   |MonogenicLinearOperator| |OctonionCategory| |SquareMatrixCategory|
   |UnivariateSkewPolynomialCategory| |XAlgebra|))
```

———-

1.28.1 DirectProductCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductCategory|
  (progn
    (push '|DirectProductCategory| *Categories*)
    (make-instance '|DirectProductCategoryType|)))
```

———-

1.28.2 DivisionRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DivisionRing|
  (progn
    (push '|DivisionRing| *Categories*)
    (make-instance '|DivisionRingType|)))
```

———-

1.28.3 FiniteRankAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteRankAlgebra|
  (progn
    (push '|FiniteRankAlgebra| *Categories*)
    (make-instance '|FiniteRankAlgebraType|)))
```

———-

1.28.4 FiniteRankNonAssociativeAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteRankNonAssociativeAlgebra|
  (progn
    (push '|FiniteRankNonAssociativeAlgebra| *Categories*)
    (make-instance '|FiniteRankNonAssociativeAlgebraType|)))
```

———-

1.28.5 FreeLieAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeLieAlgebra|
  (progn
    (push '|FreeLieAlgebra| *Categories*)
    (make-instance '|FreeLieAlgebraType|)))
```

———-

1.28.6 IntegralDomain

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegralDomain|
  (progn
    (push '|IntegralDomain| *Categories*)
    (make-instance '|IntegralDomainType|)))
```

———-

1.28.7 MonogenicLinearOperator

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonogenicLinearOperator|
  (progn
    (push '|MonogenicLinearOperator| *Categories*)
    (make-instance '|MonogenicLinearOperatorType|)))
```

———-

1.28.8 OctonionCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OctonionCategory|
  (progn
    (push '|OctonionCategory| *Categories*)
    (make-instance '|OctonionCategoryType|)))
```

———-

1.28.9 SquareMatrixCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareMatrixCategory|
  (progn
    (push '|SquareMatrixCategory| *Categories*)
    (make-instance '|SquareMatrixCategoryType|)))
```

———-

1.28.10 UnivariateSkewPolynomialCategory

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateSkewPolynomialCategory|
  (progn
    (push '|UnivariateSkewPolynomialCategory| *Categories*)
    (make-instance '|UnivariateSkewPolynomialCategoryType|)))
```

———-

1.28.11 XAlgebra

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XAlgebra|
  (progn
    (push '|XAlgebra| *Categories*)
    (make-instance '|XAlgebraType|)))
```

———-

1.29 Level 12

— **sane** —

```
(defvar level12
 '(|AbelianMonoidRing| |FortranMachineTypeCategory| |FramedAlgebra|
   |FramedNonAssociativeAlgebra| |GcdDomain|
```

```
   |LinearOrdinaryDifferentialOperatorCategory| |OrderedIntegralDomain|
   |QuaternionCategory| |XFreeAlgebra|))
```

———-

1.29.1 AbelianMonoidRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AbelianMonoidRing|
  (progn
    (push '|AbelianMonoidRing| *Categories*)
    (make-instance '|AbelianMonoidRingType|)))
```

———-

1.29.2 FortranMachineTypeCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranMachineTypeCategory|
  (progn
    (push '|FortranMachineTypeCategory| *Categories*)
    (make-instance '|FortranMachineTypeCategoryType|)))
```

———-

1.29.3 FramedAlgebra

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedAlgebra|
  (progn
    (push '|FramedAlgebra| *Categories*)
    (make-instance '|FramedAlgebraType|)))
```

———-

1.29.4 FramedNonAssociativeAlgebra

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedNonAssociativeAlgebra|
  (progn
    (push '|FramedNonAssociativeAlgebra| *Categories*)
    (make-instance '|FramedNonAssociativeAlgebraType|)))
```

———-

1.29.5 GcdDomain

— **sane** —

```
(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."))
   (arglist :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| $) (|SparseUnivariatePolynomi

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)))
```

1.29.6 LinearOrdinaryDifferentialOperatorCategory

— **sane** —

```
(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"))
   (arglist :initform nil)
```

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

(defvar |LinearOrdinaryDifferentialOperatorCategory|
  (progn
    (push '|LinearOrdinaryDifferentialOperatorCategory| *Categories*)
    (make-instance '|LinearOrdinaryDifferentialOperatorCategoryType|)))
```

———-

1.29.7 OrderedIntegralDomain

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedIntegralDomain|
  (progn
    (push '|OrderedIntegralDomain| *Categories*)
    (make-instance '|OrderedIntegralDomainType|)))
```

———-

1.29.8 QuaternionCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuaternionCategory|
  (progn
    (push '|QuaternionCategory| *Categories*)
    (make-instance '|QuaternionCategoryType|)))
```

———-

1.29.9 XFreeAlgebra

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XFreeAlgebra|
  (progn
    (push '|XFreeAlgebra| *Categories*)
```

```
    (make-instance '|XFreeAlgebraType|)))
```

———-

1.30 Level 13

— sane —

```
(defvar level13
 '(|FiniteAbelianMonoidRing| |IntervalCategory| |PowerSeriesCategory|
   |PrincipalIdealDomain| |UniqueFactorizationDomain| |XPolynomialsCat|))
```

———-

1.30.1 FiniteAbelianMonoidRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteAbelianMonoidRing|
  (progn
    (push '|FiniteAbelianMonoidRing| *Categories*)
    (make-instance '|FiniteAbelianMonoidRingType|)))
```

———-

1.30.2 IntervalCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntervalCategory|
  (progn
    (push '|IntervalCategory| *Categories*)
    (make-instance '|IntervalCategoryType|)))
```

———-

1.30.3 PowerSeriesCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PowerSeriesCategory|
  (progn
    (push '|PowerSeriesCategory| *Categories*)
    (make-instance '|PowerSeriesCategoryType|)))
```

———-

1.30.4 PrincipalIdealDomain

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrincipalIdealDomain|
  (progn
    (push '|PrincipalIdealDomain| *Categories*)
    (make-instance '|PrincipalIdealDomainType|)))
```

———-

1.30.5 UniqueFactorizationDomain

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UniqueFactorizationDomain|
```

```
  (progn
    (push '|UniqueFactorizationDomain| *Categories*)
    (make-instance '|UniqueFactorizationDomainType|)))
```

———-

1.30.6 XPolynomialsCat

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPolynomialsCat|
  (progn
    (push '|XPolynomialsCat| *Categories*)
    (make-instance '|XPolynomialsCatType|)))
```

———-

1.31 Level 14

— sane —

```
(defvar level14
  '(|EuclideanDomain| |MultivariateTaylorSeriesCategory|
    |PolynomialFactorizationExplicit| |UnivariatePowerSeriesCategory|))
```

———-

1.31.1 EuclideanDomain

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EuclideanDomain|
  (progn
    (push '|EuclideanDomain| *Categories*)
    (make-instance '|EuclideanDomainType|)))
```

———-

1.31.2 MultivariateTaylorSeriesCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariateTaylorSeriesCategory|
  (progn
```

```
    (push '|MultivariateTaylorSeriesCategory| *Categories*)
    (make-instance '|MultivariateTaylorSeriesCategoryType|)))
```

———-

1.31.3 PolynomialFactorizationExplicit

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFactorizationExplicit|
  (progn
    (push '|PolynomialFactorizationExplicit| *Categories*)
    (make-instance '|PolynomialFactorizationExplicitType|)))
```

———-

1.31.4 UnivariatePowerSeriesCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePowerSeriesCategory|
  (progn
    (push '|UnivariatePowerSeriesCategory| *Categories*)
    (make-instance '|UnivariatePowerSeriesCategoryType|)))
```

———-

1.32 Level 15

— **sane** —

```
(defvar level15
  '(|Field| |IntegerNumberSystem| |PAdicIntegerCategory|
    |PolynomialCategory| |UnivariateTaylorSeriesCategory|))
```

———-

1.32.1 Field

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |Field|
  (progn
    (push '|Field| *Categories*)
    (make-instance '|FieldType|)))
```

———-

1.32.2 IntegerNumberSystem

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerNumberSystem|
  (progn
    (push '|IntegerNumberSystem| *Categories*)
    (make-instance '|IntegerNumberSystemType|)))
```

———-

1.32.3 PAdicIntegerCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicIntegerCategory|
  (progn
    (push '|PAdicIntegerCategory| *Categories*)
    (make-instance '|PAdicIntegerCategoryType|)))
```

———-

1.32.4 PolynomialCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialCategory|
  (progn
    (push '|PolynomialCategory| *Categories*)
    (make-instance '|PolynomialCategoryType|)))
```

———-

1.32.5 UnivariateTaylorSeriesCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesCategory|
  (progn
    (push '|UnivariateTaylorSeriesCategory| *Categories*)
    (make-instance '|UnivariateTaylorSeriesCategoryType|)))
```

———-

1.33 Level 16

— sane —

```
(defvar level16
 '(|AlgebraicallyClosedField| |DifferentialPolynomialCategory|
   |FieldOfPrimeCharacteristic| |FunctionSpace| |LocalPowerSeriesCategory|
   |PseudoAlgebraicClosureOfPerfectFieldCategory| |QuotientFieldCategory|
   |RealClosedField| |RealNumberSystem| |RecursivePolynomialCategory|
   |UnivariateLaurentSeriesCategory| |UnivariatePolynomialCategory|
   |UnivariatePuiseuxSeriesCategory|))
```

———-

1.33.1 AlgebraicallyClosedField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicallyClosedField|
  (progn
    (push '|AlgebraicallyClosedField| *Categories*)
    (make-instance '|AlgebraicallyClosedFieldType|)))
```

———-

1.33.2 DifferentialPolynomialCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialPolynomialCategory|
  (progn
    (push '|DifferentialPolynomialCategory| *Categories*)
    (make-instance '|DifferentialPolynomialCategoryType|)))
```

———-

1.33.3 FieldOfPrimeCharacteristic

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FieldOfPrimeCharacteristic|
  (progn
    (push '|FieldOfPrimeCharacteristic| *Categories*)
    (make-instance '|FieldOfPrimeCharacteristicType|)))
```

———-

1.33.4 FunctionSpace

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpace|
```

```
  (progn
    (push '|FunctionSpace| *Categories*)
    (make-instance '|FunctionSpaceType|)))
```

———-

1.33.5 LocalPowerSeriesCategory

— sane —

```
(defclass |LocalPowerSeriesCategoryType| (|UnivariatePowerSeriesCategoryType| |FieldType|)
  ((parents :initform '(|UnivariatePowerSeriesCategory| |Field|))
   (name :initform "LocalPowerSeriesCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'LOCPOWC)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LocalPowerSeriesCategory|
  (progn
    (push '|LocalPowerSeriesCategory| *Categories*)
    (make-instance '|LocalPowerSeriesCategoryType|)))
```

———-

1.33.6 PseudoAlgebraicClosureOfPerfectFieldCategory

— sane —

```
(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."))
  (arglist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfPerfectFieldCategory|
  (progn
    (push '|PseudoAlgebraicClosureOfPerfectFieldCategory| *Categories*)
    (make-instance '|PseudoAlgebraicClosureOfPerfectFieldCategoryType|)))
```

———-

1.33.7 QuotientFieldCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuotientFieldCategory|
  (progn
    (push '|QuotientFieldCategory| *Categories*)
    (make-instance '|QuotientFieldCategoryType|)))
```

———-

1.33.8 RealClosedField

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealClosedField|
  (progn
    (push '|RealClosedField| *Categories*)
    (make-instance '|RealClosedFieldType|)))
```

———-

1.33.9 RealNumberSystem

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealNumberSystem|
  (progn
    (push '|RealNumberSystem| *Categories*)
    (make-instance '|RealNumberSystemType|)))
```

———-

1.33.10 RecursivePolynomialCategory

— **sane** —

```
(defclass |RecursivePolynomialCategoryType| (|PolynomialCategoryType|)
  ((parents :initform '(|PolynomialCategory|))
   (name :initform "RecursivePolynomialCategory")
   (marker :initform 'category)
   (level :initform 16)
   (abbreviation :initform 'RPOLCAT)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RecursivePolynomialCategory|
  (progn
    (push '|RecursivePolynomialCategory| *Categories*)
    (make-instance '|RecursivePolynomialCategoryType|)))
```

———-

1.33.11 UnivariateLaurentSeriesCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeriesCategory|
  (progn
    (push '|UnivariateLaurentSeriesCategory| *Categories*)
    (make-instance '|UnivariateLaurentSeriesCategoryType|)))
```

———-

1.33.12 UnivariatePolynomialCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialCategory|
  (progn
    (push '|UnivariatePolynomialCategory| *Categories*)
    (make-instance '|UnivariatePolynomialCategoryType|)))
```

———-

1.33.13 UnivariatePuiseuxSeriesCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesCategory|
  (progn
    (push '|UnivariatePuiseuxSeriesCategory| *Categories*)
    (make-instance '|UnivariatePuiseuxSeriesCategoryType|)))
```

———-

1.34 Level 17

— sane —

```
(defvar level17
  '(|AlgebraicallyClosedFunctionSpace| |ExtensionField| |FiniteFieldCategory|
    |FloatingPointSystem| |UnivariateLaurentSeriesConstructorCategory|
    |UnivariatePuiseuxSeriesConstructorCategory|))
```

———-

1.34.1 AlgebraicallyClosedFunctionSpace

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicallyClosedFunctionSpace|
  (progn
    (push '|AlgebraicallyClosedFunctionSpace| *Categories*)
    (make-instance '|AlgebraicallyClosedFunctionSpaceType|)))
```

———-

1.34.2 ExtensionField

— sane —

```
(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"))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExtensionField|
  (progn
    (push '|ExtensionField| *Categories*)
    (make-instance '|ExtensionFieldType|)))
```

———-

1.34.3 FiniteFieldCategory

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCategory|
  (progn
    (push '|FiniteFieldCategory| *Categories*)
    (make-instance '|FiniteFieldCategoryType|)))
```

———-

1.34.4 FloatingPointSystem

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatingPointSystem|
  (progn
    (push '|FloatingPointSystem| *Categories*)
    (make-instance '|FloatingPointSystemType|)))
```

———-

1.34.5 UnivariateLaurentSeriesConstructorCategory

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |UnivariateLaurentSeriesConstructorCategory|
  (progn
    (push '|UnivariateLaurentSeriesConstructorCategory| *Categories*)
    (make-instance '|UnivariateLaurentSeriesConstructorCategoryType|)))
```

———-

1.34.6 UnivariatePuiseuxSeriesConstructorCategory

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesConstructorCategory|
  (progn
    (push '|UnivariatePuiseuxSeriesConstructorCategory| *Categories*)
    (make-instance '|UnivariatePuiseuxSeriesConstructorCategoryType|)))
```

———-

1.35 Level 18

— **sane** —

```
(defvar level18
  '(|FiniteAlgebraicExtensionField| |MonogenicAlgebra|
    |PseudoAlgebraicClosureOfFiniteFieldCategory|
    |PseudoAlgebraicClosureOfRationalNumberCategory|))
```

———-

1.35.1 FiniteAlgebraicExtensionField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteAlgebraicExtensionField|
  (progn
    (push '|FiniteAlgebraicExtensionField| *Categories*)
    (make-instance '|FiniteAlgebraicExtensionFieldType|)))
```

———-

1.35.2 MonogenicAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonogenicAlgebra|
  (progn
    (push '|MonogenicAlgebra| *Categories*)
    (make-instance '|MonogenicAlgebraType|)))
```

———-

1.35.3 PseudoAlgebraicClosureOfFiniteFieldCategory

— sane —

```
(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."))
  (arglist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfFiniteFieldCategory|
  (progn
    (push '|PseudoAlgebraicClosureOfFiniteFieldCategory| *Categories*)
    (make-instance '|PseudoAlgebraicClosureOfFiniteFieldCategoryType|)))
```

———-

1.35.4 PseudoAlgebraicClosureOfRationalNumberCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |PseudoAlgebraicClosureOfRationalNumberCategory|
  (progn
    (push '|PseudoAlgebraicClosureOfRationalNumberCategory| *Categories*)
    (make-instance '|PseudoAlgebraicClosureOfRationalNumberCategoryType|)))
```

———-

1.36 Level 19

— **sane** —

```
(defvar level19
  '(|ComplexCategory| |FunctionFieldCategory|
    |PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory|))
```

———-

1.36.1 ComplexCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexCategory|
```

```
  (progn
    (push '|ComplexCategory| *Categories*)
    (make-instance '|ComplexCategoryType|)))
```

———-

1.36.2 FunctionFieldCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionFieldCategory|
  (progn
    (push '|FunctionFieldCategory| *Categories*)
    (make-instance '|FunctionFieldCategoryType|)))
```

———-

1.36.3 PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory

— sane —

```
(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."))
   (arglist :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")
```

1.36.4 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.

1.36.5 defstruct database

— 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
```

———-

1.36.6 defvar *defaultdomain-list*

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|)))
```

———-

1.36.7 defvar *operation-hash*

— **initvars** —

```
(defvar *operation-hash* nil "given an operation name, what are its modemaps?")
```

———-

1.36.8 defvar *hasCategory-hash*

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")
```

———-

1.36.9 defvar *miss*

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.))
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

1.37 A

1.37.1 AffinePlane

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffinePlane|
  (progn
    (push '|AffinePlane| *Domains*)
    (make-instance '|AffinePlaneType|)))
```

———-

1.37.2 AffinePlaneOverPseudoAlgebraicClosureOfFiniteField

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffinePlaneOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|AffinePlaneOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|AffinePlaneOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.37.3 AffineSpace

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffineSpace|
  (progn
    (push '|AffineSpace| *Domains*)
    (make-instance '|AffineSpaceType|)))
```

———-

1.37.4 AlgebraGivenByStructuralConstants

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraGivenByStructuralConstants|
  (progn
    (push '|AlgebraGivenByStructuralConstants| *Domains*)
    (make-instance '|AlgebraGivenByStructuralConstantsType|)))
```

———-

1.37.5 AlgebraicFunctionField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicFunctionField|
  (progn
    (push '|AlgebraicFunctionField| *Domains*)
    (make-instance '|AlgebraicFunctionFieldType|)))
```

———-

1.37.6 AlgebraicNumber

— sane —

```
(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 ="))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicNumber|
  (progn
    (push '|AlgebraicNumber| *Domains*)
    (make-instance '|AlgebraicNumberType|)))
```

———-

1.37.7 AnonymousFunction

— **sane** —

```
(defclass |AnonymousFunctionType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "AnonymousFunction")
   (marker :initform 'domain)
   (abbreviation :initform 'ANON)
   (comment :initform (list
     "This domain implements anonymous functions"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnonymousFunction|
  (progn
    (push '|AnonymousFunction| *Domains*)
    (make-instance '|AnonymousFunctionType|)))
```

———-

1.37.8 AntiSymm

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AntiSymm|
  (progn
    (push '|AntiSymm| *Domains*)
    (make-instance '|AntiSymmType|)))
```

———-

1.37.9 Any

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Any|
  (progn
    (push '|Any| *Domains*)
    (make-instance '|AnyType|)))
```

———-

1.37.10 ArrayStack

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ArrayStack|
  (progn
    (push '|ArrayStack| *Domains*)
    (make-instance '|ArrayStackType|)))
```

———-

1.37.11 Asp1

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp1|
  (progn
```

```
    (push '|Asp1| *Domains*)
    (make-instance '|Asp1Type|)))
```

———-

1.37.12 Asp10

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp10|
  (progn
    (push '|Asp10| *Domains*)
    (make-instance '|Asp10Type|)))
```

———-

1.37.13 Asp12

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp12|
  (progn
    (push '|Asp12| *Domains*)
    (make-instance '|Asp12Type|)))
```

———-

1.37.14 Asp19

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp19|
  (progn
    (push '|Asp19| *Domains*)
    (make-instance '|Asp19Type|)))
```

———-

1.37.15 Asp20

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp20|
  (progn
    (push '|Asp20| *Domains*)
    (make-instance '|Asp20Type|)))
```

———-

1.37.16 Asp24

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp24|
  (progn
    (push '|Asp24| *Domains*)
    (make-instance '|Asp24Type|)))
```

———-

1.37.17 Asp27

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp27|
```

```
  (progn
    (push '|Asp27| *Domains*)
    (make-instance '|Asp27Type|)))
```

———-

1.37.18 Asp28

— sane —

```
(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)+0.00174739587495405 1D0*Z(15)+0.00"
     "    &2106973900813502D0*Z(14)+0.00295743499176908 7D0*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)+0.00523916596077929 9D0*Z(15)+(-0."
     "    &01623427735779699D0*Z(14))+(-0.01965809746040371D0*Z(13))+0.054688"
     "    &97337339577D0*Z(12)+(-0.014224695935687D0*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"
"    &4507963208617 7D0*Z(12))+(-0.05034242196614937D0*Z(11))+(-0.0376966"
"    &3291725935D0*Z(10))+0.2171103102175198D0*Z(9)+(-0.0824949256021352"
"    &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"
"    &26082005465965D0*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))+(-0.32800191089037 7D0*Z(7))+0.952576555482747D0*Z(6)+(-0.31583"
"    &09975786731D0*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"
"    &1173384583 4199D0*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))+(-0.01422469593568 7D0*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)+0.022734501110773 7D0*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"))
  (arglist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |Asp28|
  (progn
    (push '|Asp28| *Domains*)
    (make-instance '|Asp28Type|)))
```

———-

1.37.19 Asp29

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp29|
  (progn
    (push '|Asp29| *Domains*)
    (make-instance '|Asp29Type|)))
```

———-

1.37.20 Asp30

— sane —

```
(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"))
  (arglist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |Asp30|
  (progn
    (push '|Asp30| *Domains*)
    (make-instance '|Asp30Type|)))
```

———-

1.37.21 Asp31

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp31|
  (progn
    (push '|Asp31| *Domains*)
    (make-instance '|Asp31Type|)))
```

———-

1.37.22 Asp33

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp33|
  (progn
    (push '|Asp33| *Domains*)
    (make-instance '|Asp33Type|)))
```

———-

1.37.23 Asp34

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Asp34|
  (progn
    (push '|Asp34| *Domains*)
    (make-instance '|Asp34Type|)))
```

———-

1.37.24 Asp35

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp35|
  (progn
    (push '|Asp35| *Domains*)
    (make-instance '|Asp35Type|)))
```

———-

1.37.25 Asp4

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp4|
  (progn
    (push '|Asp4| *Domains*)
    (make-instance '|Asp4Type|)))
```

———-

1.37.26 Asp41

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp41|
  (progn
    (push '|Asp41| *Domains*)
    (make-instance '|Asp41Type|)))
```

———-

1.37.27 Asp42

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp42|
  (progn
    (push '|Asp42| *Domains*)
    (make-instance '|Asp42Type|)))
```

———-

1.37.28 Asp49

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp49|
  (progn
    (push '|Asp49| *Domains*)
    (make-instance '|Asp49Type|)))
```

———-

1.37.29 Asp50

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp50|
  (progn
    (push '|Asp50| *Domains*)
    (make-instance '|Asp50Type|)))
```

———-

1.37.30 Asp55

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp55|
```

```
  (progn
    (push '|Asp55| *Domains*)
    (make-instance '|Asp55Type|)))
```

———-

1.37.31 Asp6

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp6|
  (progn
    (push '|Asp6| *Domains*)
    (make-instance '|Asp6Type|)))
```

———-

1.37.32 Asp7

— sane —

```
(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)=((-0.03199999999999999D0*DCOS(Z(3))*DTAN(Z(3)))+(-0.02D0*Z(2)"
     "    &**2))/(Z(2)*DCOS(Z(3)))"
     "     F(3)=-0.03199999999999999D0/(X*Z(2)**2)"
     "     RETURN"
     "     END"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp7|
  (progn
    (push '|Asp7| *Domains*)
    (make-instance '|Asp7Type|)))
```

———-

1.37.33 Asp73

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp73|
  (progn
    (push '|Asp73| *Domains*)
    (make-instance '|Asp73Type|)))
```

———-

1.37.34 Asp74

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp74|
  (progn
    (push '|Asp74| *Domains*)
    (make-instance '|Asp74Type|)))
```

———-

1.37.35 Asp77

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp77|
  (progn
    (push '|Asp77| *Domains*)
    (make-instance '|Asp77Type|)))
```

———-

1.37.36 Asp78

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp78|
  (progn
    (push '|Asp78| *Domains*)
    (make-instance '|Asp78Type|)))
```

———-

1.37.37 Asp8

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp8|
  (progn
    (push '|Asp8| *Domains*)
    (make-instance '|Asp8Type|)))
```

———-

1.37.38 Asp80

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp80|
  (progn
    (push '|Asp80| *Domains*)
    (make-instance '|Asp80Type|)))
```

———-

1.37.39 Asp9

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Asp9|
  (progn
    (push '|Asp9| *Domains*)
    (make-instance '|Asp9Type|)))
```

———-

1.37.40 AssociatedJordanAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociatedJordanAlgebra|
  (progn
    (push '|AssociatedJordanAlgebra| *Domains*)
    (make-instance '|AssociatedJordanAlgebraType|)))
```

———-

1.37.41 AssociatedLieAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociatedLieAlgebra|
  (progn
    (push '|AssociatedLieAlgebra| *Domains*)
    (make-instance '|AssociatedLieAlgebraType|)))
```

———-

1.37.42 AssociationList

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociationList|
  (progn
    (push '|AssociationList| *Domains*)
    (make-instance '|AssociationListType|)))
```

———-

1.37.43 AttributeButtons

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AttributeButtons|
  (progn
    (push '|AttributeButtons| *Domains*)
    (make-instance '|AttributeButtonsType|)))
```

———-

1.37.44 Automorphism

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Automorphism|
  (progn
    (push '|Automorphism| *Domains*)
    (make-instance '|AutomorphismType|)))
```

———-

1.38 B

1.38.1 BalancedBinaryTree

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |BalancedBinaryTree|
  (progn
    (push '|BalancedBinaryTree| *Domains*)
    (make-instance '|BalancedBinaryTreeType|)))
```

———-

1.38.2 BalancedPAdicInteger

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BalancedPAdicInteger|
  (progn
    (push '|BalancedPAdicInteger| *Domains*)
    (make-instance '|BalancedPAdicIntegerType|)))
```

———-

1.38.3 BalancedPAdicRational

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |BalancedPAdicRational|
  (progn
    (push '|BalancedPAdicRational| *Domains*)
    (make-instance '|BalancedPAdicRationalType|)))
```

———-

1.38.4 BasicFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicFunctions|
  (progn
    (push '|BasicFunctions| *Domains*)
    (make-instance '|BasicFunctionsType|)))
```

———-

1.38.5 BasicOperator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |BasicOperator|
  (progn
    (push '|BasicOperator| *Domains*)
    (make-instance '|BasicOperatorType|)))
```

———-

1.38.6 BasicStochasticDifferential

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicStochasticDifferential|
  (progn
    (push '|BasicStochasticDifferential| *Domains*)
    (make-instance '|BasicStochasticDifferentialType|)))
```

———-

1.38.7 BinaryExpansion

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryExpansion|
  (progn
    (push '|BinaryExpansion| *Domains*)
    (make-instance '|BinaryExpansionType|)))
```

———-

1.38.8 BinaryFile

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryFile|
  (progn
    (push '|BinaryFile| *Domains*)
    (make-instance '|BinaryFileType|)))
```

———-

1.38.9 BinarySearchTree

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinarySearchTree|
  (progn
    (push '|BinarySearchTree| *Domains*)
    (make-instance '|BinarySearchTreeType|)))
```

———-

1.38.10 BinaryTournament

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryTournament|
  (progn
    (push '|BinaryTournament| *Domains*)
    (make-instance '|BinaryTournamentType|)))
```

———-

1.38.11 BinaryTree

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BinaryTree|
  (progn
    (push '|BinaryTree| *Domains*)
    (make-instance '|BinaryTreeType|)))
```

———-

1.38.12 Bits

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

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

———-

1.38.13 BlowUpWithHamburgerNoether

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BlowUpWithHamburgerNoether|
  (progn
    (push '|BlowUpWithHamburgerNoether| *Domains*)
    (make-instance '|BlowUpWithHamburgerNoetherType|)))
```

———-

1.38.14 BlowUpWithQuadTrans

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BlowUpWithQuadTrans|
  (progn
    (push '|BlowUpWithQuadTrans| *Domains*)
    (make-instance '|BlowUpWithQuadTransType|)))
```

———-

1.38.15 Boolean

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Boolean|
  (progn
    (push '|Boolean| *Domains*)
    (make-instance '|BooleanType|)))
```

———-

1.39 C

1.39.1 CardinalNumber

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CardinalNumber|
  (progn
    (push '|CardinalNumber| *Domains*)
    (make-instance '|CardinalNumberType|)))
```

———-

1.39.2 CartesianTensor

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CartesianTensor|
  (progn
    (push '|CartesianTensor| *Domains*)
    (make-instance '|CartesianTensorType|)))
```

———-

1.39.3 Cell

— sane —

```
(defclass |CellType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "Cell")
   (marker :initform 'domain)
   (abbreviation :initform 'CELL)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Cell|
  (progn
    (push '|Cell| *Domains*)
    (make-instance '|CellType|)))
```

———-

1.39.4 Character

— sane —

```
(defclass |CharacterType| (|OrderedFiniteType|)
  ((parents :initform '(|OrderedFinite|))
   (name :initform "Character")
   (marker :initform 'domain)
   (abbreviation :initform 'CHAR)))

(defvar |Character|
  (progn
    (push '|Character| *Domains*)
    (make-instance '|CharacterType|)))
```

———-

1.39.5 CharacterClass

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |CharacterClass|
  (progn
    (push '|CharacterClass| *Domains*)
    (make-instance '|CharacterClassType|)))
```

———-

1.39.6 CliffordAlgebra

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CliffordAlgebra|
  (progn
    (push '|CliffordAlgebra| *Domains*)
    (make-instance '|CliffordAlgebraType|)))
```

———-

1.39.7 Color

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Color|
  (progn
    (push '|Color| *Domains*)
    (make-instance '|ColorType|)))
```

———-

1.39.8 Commutator

— sane —

```
(defclass |CommutatorType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Commutator")
   (marker :initform 'domain)
   (abbreviation :initform 'COMM)
   (comment :initform (list
     "A type for basic commutators"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Commutator|
  (progn
    (push '|Commutator| *Domains*)
    (make-instance '|CommutatorType|)))
```

———-

1.39.9 Complex

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Complex|
  (progn
    (push '|Complex| *Domains*)
    (make-instance '|ComplexType|)))
```

———-

1.39.10 ComplexDoubleFloatMatrix

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexDoubleFloatMatrix|
  (progn
    (push '|ComplexDoubleFloatMatrix| *Domains*)
    (make-instance '|ComplexDoubleFloatMatrixType|)))
```

———-

1.39.11 ComplexDoubleFloatVector

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexDoubleFloatVector|
  (progn
    (push '|ComplexDoubleFloatVector| *Domains*)
    (make-instance '|ComplexDoubleFloatVectorType|)))
```

———-

1.39.12 ContinuedFraction

— sane —

```
(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))."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |ContinuedFraction|
  (progn
    (push '|ContinuedFraction| *Domains*)
    (make-instance '|ContinuedFractionType|)))
```

———-

1.40 D

1.40.1 Database

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Database|
  (progn
    (push '|Database| *Domains*)
    (make-instance '|DatabaseType|)))
```

———-

1.40.2 DataList

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |DataList|
  (progn
    (push '|DataList| *Domains*)
    (make-instance '|DataListType|)))
```

———-

1.40.3 DecimalExpansion

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DecimalExpansion|
  (progn
    (push '|DecimalExpansion| *Domains*)
    (make-instance '|DecimalExpansionType|)))
```

———-

1.40.4 DenavitHartenbergMatrix

— **sane** —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DenavitHartenbergMatrix|
  (progn
    (push '|DenavitHartenbergMatrix| *Domains*)
    (make-instance '|DenavitHartenbergMatrixType|)))
```

———-

1.40.5 Dequeue

— **sane** —

```
(defclass |DequeueType| (|DequeueAggregateType|)
  ((parents :initform '(|DequeueAggregate|))
   (name :initform "Dequeue")
   (marker :initform 'domain)
   (abbreviation :initform 'DEQUEUE)
   (comment :initform (list
     "Linked list implementation of a Dequeue"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Dequeue|
  (progn
    (push '|Dequeue| *Domains*)
    (make-instance '|DequeueType|)))
```

———-

1.40.6 DeRhamComplex

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DeRhamComplex|
  (progn
    (push '|DeRhamComplex| *Domains*)
    (make-instance '|DeRhamComplexType|)))
```

———-

1.40.7 DesingTree

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DesingTree|
  (progn
    (push '|DesingTree| *Domains*)
    (make-instance '|DesingTreeType|)))
```

———-

1.40.8 DifferentialSparseMultivariatePolynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DifferentialSparseMultivariatePolynomial|
  (progn
    (push '|DifferentialSparseMultivariatePolynomial| *Domains*)
    (make-instance '|DifferentialSparseMultivariatePolynomialType|)))
```

———-

1.40.9 DirectProduct

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProduct|
  (progn
    (push '|DirectProduct| *Domains*)
    (make-instance '|DirectProductType|)))
```

———-

1.40.10 DirectProductMatrixModule

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductMatrixModule|
  (progn
    (push '|DirectProductMatrixModule| *Domains*)
    (make-instance '|DirectProductMatrixModuleType|)))
```

———-

1.40.11 DirectProductModule

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductModule|
  (progn
    (push '|DirectProductModule| *Domains*)
    (make-instance '|DirectProductModuleType|)))
```

———-

1.40.12 DirichletRing

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirichletRing|
  (progn
    (push '|DirichletRing| *Domains*)
    (make-instance '|DirichletRingType|)))
```

———-

1.40.13 DistributedMultivariatePolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |DistributedMultivariatePolynomial|
  (progn
    (push '|DistributedMultivariatePolynomial| *Domains*)
    (make-instance '|DistributedMultivariatePolynomialType|)))
```

———-

1.40.14 Divisor

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Divisor|
  (progn
    (push '|Divisor| *Domains*)
    (make-instance '|DivisorType|)))
```

———-

1.40.15 DoubleFloat

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleFloat|
  (progn
    (push '|DoubleFloat| *Domains*)
    (make-instance '|DoubleFloatType|)))
```

———-

1.40.16 DoubleFloatMatrix

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |DoubleFloatMatrix|
  (progn
    (push '|DoubleFloatMatrix| *Domains*)
    (make-instance '|DoubleFloatMatrixType|)))
```

———-

1.40.17 DoubleFloatVector

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleFloatVector|
  (progn
    (push '|DoubleFloatVector| *Domains*)
    (make-instance '|DoubleFloatVectorType|)))
```

———-

1.40.18 DrawOption

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |DrawOption|
  (progn
    (push '|DrawOption| *Domains*)
    (make-instance '|DrawOptionType|)))
```

———-

1.40.19 d01ajfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01ajfAnnaType|
  (progn
    (push '|d01ajfAnnaType| *Domains*)
    (make-instance '|d01ajfAnnaTypeType|)))
```

———-

1.40.20 d01akfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01akfAnnaType|
  (progn
    (push '|d01akfAnnaType| *Domains*)
    (make-instance '|d01akfAnnaTypeType|)))
```

———-

1.40.21 d01alfAnnaType

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01alfAnnaType|
  (progn
    (push '|d01alfAnnaType| *Domains*)
    (make-instance '|d01alfAnnaTypeType|)))
```

———-

1.40.22 d01amfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01amfAnnaType|
  (progn
    (push '|d01amfAnnaType| *Domains*)
    (make-instance '|d01amfAnnaTypeType|)))
```

———-

1.40.23 d01anfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |d01anfAnnaType|
  (progn
    (push '|d01anfAnnaType| *Domains*)
    (make-instance '|d01anfAnnaTypeType|)))
```

———-

1.40.24 d01apfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01apfAnnaType|
  (progn
    (push '|d01apfAnnaType| *Domains*)
    (make-instance '|d01apfAnnaTypeType|)))
```

———-

1.40.25 d01aqfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01aqfAnnaType|
  (progn
    (push '|d01aqfAnnaType| *Domains*)
    (make-instance '|d01aqfAnnaTypeType|)))
```

———-

1.40.26 d01asfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01asfAnnaType|
  (progn
    (push '|d01asfAnnaType| *Domains*)
    (make-instance '|d01asfAnnaTypeType|)))
```

———-

1.40.27 d01fcfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01fcfAnnaType|
  (progn
    (push '|d01fcfAnnaType| *Domains*)
    (make-instance '|d01fcfAnnaTypeType|)))
```

———-

1.40.28 d01gbfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |d01gbfAnnaType|
  (progn
    (push '|d01gbfAnnaType| *Domains*)
    (make-instance '|d01gbfAnnaTypeType|)))
```

———-

1.40.29 d01TransformFunctionType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01TransformFunctionType|
  (progn
    (push '|d01TransformFunctionType| *Domains*)
    (make-instance '|d01TransformFunctionTypeType|)))
```

———-

1.40.30 d02bbfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02bbfAnnaType|
  (progn
    (push '|d02bbfAnnaType| *Domains*)
    (make-instance '|d02bbfAnnaTypeType|)))
```

———-

1.40.31 d02bhfAnnaType

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02bhfAnnaType|
  (progn
    (push '|d02bhfAnnaType| *Domains*)
    (make-instance '|d02bhfAnnaTypeType|)))
```

———-

1.40.32 d02cjfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02cjfAnnaType|
  (progn
    (push '|d02cjfAnnaType| *Domains*)
    (make-instance '|d02cjfAnnaTypeType|)))
```

———-

1.40.33 d02ejfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |d02ejfAnnaType|
  (progn
    (push '|d02ejfAnnaType| *Domains*)
    (make-instance '|d02ejfAnnaTypeType|)))
```

———-

1.40.34 d03eefAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d03eefAnnaType|
  (progn
    (push '|d03eefAnnaType| *Domains*)
    (make-instance '|d03eefAnnaTypeType|)))
```

———-

1.40.35 d03fafAnnaType

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d03fafAnnaType|
  (progn
    (push '|d03fafAnnaType| *Domains*)
    (make-instance '|d03fafAnnaTypeType|)))
```

———-

1.41 E

1.41.1 ElementaryFunctionsUnivariateLaurentSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionsUnivariateLaurentSeries|
  (progn
    (push '|ElementaryFunctionsUnivariateLaurentSeries| *Domains*)
    (make-instance '|ElementaryFunctionsUnivariateLaurentSeriesType|)))
```

———-

1.41.2 ElementaryFunctionsUnivariatePuiseuxSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionsUnivariatePuiseuxSeries|
  (progn
    (push '|ElementaryFunctionsUnivariatePuiseuxSeries| *Domains*)
    (make-instance '|ElementaryFunctionsUnivariatePuiseuxSeriesType|)))
```

———-

1.41.3 Equation

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |Equation|
  (progn
    (push '|Equation| *Domains*)
    (make-instance '|EquationType|)))
```

———-

1.41.4 EqTable

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EqTable|
  (progn
    (push '|EqTable| *Domains*)
    (make-instance '|EqTableType|)))
```

———-

1.41.5 EuclideanModularRing

— sane —

```
(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"))
   (arglist :initform nil)
```

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

(defvar |EuclideanModularRing|
  (progn
    (push '|EuclideanModularRing| *Domains*)
    (make-instance '|EuclideanModularRingType|)))
```

———-

1.41.6 Exit

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Exit|
  (progn
    (push '|Exit| *Domains*)
    (make-instance '|ExitType|)))
```

———-

1.41.7 ExponentialExpansion

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExponentialExpansion|
  (progn
    (push '|ExponentialExpansion| *Domains*)
    (make-instance '|ExponentialExpansionType|)))
```

———-

1.41.8 Expression

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Expression|
  (progn
    (push '|Expression| *Domains*)
    (make-instance '|ExpressionType|)))
```

———-

1.41.9 ExponentialOfUnivariatePuiseuxSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExponentialOfUnivariatePuiseuxSeries|
  (progn
    (push '|ExponentialOfUnivariatePuiseuxSeries| *Domains*)
    (make-instance '|ExponentialOfUnivariatePuiseuxSeriesType|)))
```

———-

1.41.10 ExtAlgBasis

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExtAlgBasis|
  (progn
    (push '|ExtAlgBasis| *Domains*)
    (make-instance '|ExtAlgBasisType|)))
```

———-

1.41.11 e04dgfAnnaType

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04dgfAnnaType|
  (progn
    (push '|e04dgfAnnaType| *Domains*)
    (make-instance '|e04dgfAnnaTypeType|)))
```

———-

1.41.12 e04fdfAnnaType

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04fdfAnnaType|
  (progn
    (push '|e04fdfAnnaType| *Domains*)
    (make-instance '|e04fdfAnnaTypeType|)))
```

———-

1.41.13 e04gcfAnnaType

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04gcfAnnaType|
  (progn
    (push '|e04gcfAnnaType| *Domains*)
    (make-instance '|e04gcfAnnaTypeType|)))
```

———-

1.41.14 e04jafAnnaType

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04jafAnnaType|
  (progn
    (push '|e04jafAnnaType| *Domains*)
    (make-instance '|e04jafAnnaTypeType|)))
```

————-

1.41.15 e04mbfAnnaType

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |e04mbfAnnaType|
  (progn
    (push '|e04mbfAnnaType| *Domains*)
    (make-instance '|e04mbfAnnaTypeType|)))
```

———-

1.41.16 e04nafAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04nafAnnaType|
  (progn
    (push '|e04nafAnnaType| *Domains*)
    (make-instance '|e04nafAnnaTypeType|)))
```

———-

1.41.17 e04ucfAnnaType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04ucfAnnaType|
  (progn
    (push '|e04ucfAnnaType| *Domains*)
    (make-instance '|e04ucfAnnaTypeType|)))
```

———-

1.42 F

1.42.1 Factored

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |Factored|
  (progn
    (push '|Factored| *Domains*)
    (make-instance '|FactoredType|)))
```

———-

1.42.2 File

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |File|
  (progn
    (push '|File| *Domains*)
    (make-instance '|FileType|)))
```

———-

1.42.3 FileName

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |FileName|
  (progn
    (push '|FileName| *Domains*)
    (make-instance '|FileNameType|)))
```

———-

1.42.4 FiniteDivisor

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteDivisor|
  (progn
    (push '|FiniteDivisor| *Domains*)
    (make-instance '|FiniteDivisorType|)))
```

———-

1.42.5 FiniteField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |FiniteField|
  (progn
    (push '|FiniteField| *Domains*)
    (make-instance '|FiniteFieldType|)))
```

———-

1.42.6 FiniteFieldCyclicGroup

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCyclicGroup|
  (progn
    (push '|FiniteFieldCyclicGroup| *Domains*)
    (make-instance '|FiniteFieldCyclicGroupType|)))
```

———-

1.42.7 FiniteFieldCyclicGroupExtension

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCyclicGroupExtension|
  (progn
    (push '|FiniteFieldCyclicGroupExtension| *Domains*)
    (make-instance '|FiniteFieldCyclicGroupExtensionType|)))
```

———-

1.42.8 FiniteFieldCyclicGroupExtensionByPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldCyclicGroupExtensionByPolynomial|
  (progn
```

```
    (push '|FiniteFieldCyclicGroupExtensionByPolynomial| *Domains*)
    (make-instance '|FiniteFieldCyclicGroupExtensionByPolynomialType|)))
```

———-

1.42.9 FiniteFieldExtension

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldExtension|
  (progn
    (push '|FiniteFieldExtension| *Domains*)
    (make-instance '|FiniteFieldExtensionType|)))
```

———-

1.42.10 FiniteFieldExtensionByPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |FiniteFieldExtensionByPolynomial|
  (progn
    (push '|FiniteFieldExtensionByPolynomial| *Domains*)
    (make-instance '|FiniteFieldExtensionByPolynomialType|)))
```

———-

1.42.11 FiniteFieldNormalBasis

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldNormalBasis|
  (progn
    (push '|FiniteFieldNormalBasis| *Domains*)
    (make-instance '|FiniteFieldNormalBasisType|)))
```

———-

1.42.12 FiniteFieldNormalBasisExtension

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldNormalBasisExtension|
  (progn
    (push '|FiniteFieldNormalBasisExtension| *Domains*)
    (make-instance '|FiniteFieldNormalBasisExtensionType|)))
```

———-

1.42.13 FiniteFieldNormalBasisExtensionByPolynomial

— **sane** —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldNormalBasisExtensionByPolynomial|
  (progn
    (push '|FiniteFieldNormalBasisExtensionByPolynomial| *Domains*)
    (make-instance '|FiniteFieldNormalBasisExtensionByPolynomialType|)))
```

———-

1.42.14 FlexibleArray

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FlexibleArray|
  (progn
    (push '|FlexibleArray| *Domains*)
    (make-instance '|FlexibleArrayType|)))
```

———-

1.42.15 Float

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Float|
  (progn
    (push '|Float| *Domains*)
    (make-instance '|FloatType|)))
```

———-

1.42.16 FortranCode

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranCode|
  (progn
    (push '|FortranCode| *Domains*)
    (make-instance '|FortranCodeType|)))
```

———-

1.42.17 FortranExpression

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranExpression|
  (progn
    (push '|FortranExpression| *Domains*)
    (make-instance '|FortranExpressionType|)))
```

———-

1.42.18 FortranProgram

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranProgram|
  (progn
    (push '|FortranProgram| *Domains*)
    (make-instance '|FortranProgramType|)))
```

———-

1.42.19 FortranScalarType

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranScalarType|
  (progn
    (push '|FortranScalarType| *Domains*)
    (make-instance '|FortranScalarTypeType|)))
```

———-

1.42.20 FortranTemplate

— **sane** —

```
(defclass |FortranTemplateType| (|FileCategoryType|)
  ((parents :initform '(|FileCategory|))
   (name :initform "FortranTemplate")
   (marker :initform 'domain)
   (abbreviation :initform 'FTEM)
   (comment :initform (list
     "Code to manipulate Fortran templates"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranTemplate|
  (progn
    (push '|FortranTemplate| *Domains*)
    (make-instance '|FortranTemplateType|)))
```

———-

1.42.21 FortranType

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranType|
  (progn
    (push '|FortranType| *Domains*)
    (make-instance '|FortranTypeType|)))
```

———-

1.42.22 FourierComponent

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FourierComponent|
  (progn
    (push '|FourierComponent| *Domains*)
    (make-instance '|FourierComponentType|)))
```

———-

1.42.23 FourierSeries

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FourierSeries|
  (progn
    (push '|FourierSeries| *Domains*)
    (make-instance '|FourierSeriesType|)))
```

———-

1.42.24 Fraction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Fraction|
  (progn
    (push '|Fraction| *Domains*)
    (make-instance '|FractionType|)))
```

———-

1.42.25 FractionalIdeal

— sane —

```
(defclass |FractionalIdealType| (|GroupType|)
  ((parents :initform '(|Group|))
   (name :initform "FractionalIdeal")
   (marker :initform 'domain)
   (abbreviation :initform 'FRIDEAL)
   (comment :initform (list
     "Fractional ideals in a framed algebra."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionalIdeal|
  (progn
    (push '|FractionalIdeal| *Domains*)
    (make-instance '|FractionalIdealType|)))
```

———-

1.42.26 FramedModule

— sane —

```
(defclass |FramedModuleType| (|MonoidType|)
  ((parents :initform '(|Monoid|))
   (name :initform "FramedModule")
   (marker :initform 'domain)
   (abbreviation :initform 'FRMOD)
   (comment :initform (list
     "Module representation of fractional ideals."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedModule|
  (progn
    (push '|FramedModule| *Domains*)
    (make-instance '|FramedModuleType|)))
```

———-

1.42.27 FreeAbelianGroup

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeAbelianGroup|
  (progn
    (push '|FreeAbelianGroup| *Domains*)
    (make-instance '|FreeAbelianGroupType|)))
```

————-

1.42.28 FreeAbelianMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeAbelianMonoid|
  (progn
```

```
    (push '|FreeAbelianMonoid| *Domains*)
    (make-instance '|FreeAbelianMonoidType|)))
```

———-

1.42.29 FreeGroup

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeGroup|
  (progn
    (push '|FreeGroup| *Domains*)
    (make-instance '|FreeGroupType|)))
```

———-

1.42.30 FreeModule

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |FreeModule|
  (progn
    (push '|FreeModule| *Domains*)
    (make-instance '|FreeModuleType|)))
```

———-

1.42.31 FreeModule1

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeModule1|
  (progn
    (push '|FreeModule1| *Domains*)
    (make-instance '|FreeModule1Type|)))
```

———-

1.42.32 FreeMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeMonoid|
  (progn
    (push '|FreeMonoid| *Domains*)
    (make-instance '|FreeMonoidType|)))
```

———-

1.42.33 FreeNilpotentLie

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FreeNilpotentLie|
  (progn
    (push '|FreeNilpotentLie| *Domains*)
    (make-instance '|FreeNilpotentLieType|)))
```

———-

1.42.34 FullPartialFractionExpansion

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FullPartialFractionExpansion|
  (progn
    (push '|FullPartialFractionExpansion| *Domains*)
    (make-instance '|FullPartialFractionExpansionType|)))
```

———-

1.42.35 FunctionCalled

— **sane** —

```
(defclass |FunctionCalledType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "FunctionCalled")
   (marker :initform 'domain)
   (abbreviation :initform 'FUNCTION)
   (comment :initform (list
     "This domain implements named functions"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionCalled|
  (progn
    (push '|FunctionCalled| *Domains*)
    (make-instance '|FunctionCalledType|)))
```

———-

1.43 G

1.43.1 GeneralDistributedMultivariatePolynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralDistributedMultivariatePolynomial|
  (progn
    (push '|GeneralDistributedMultivariatePolynomial| *Domains*)
    (make-instance '|GeneralDistributedMultivariatePolynomialType|)))
```

———-

1.43.2 GeneralModulePolynomial

— sane —

```
(defclass |GeneralModulePolynomialType| (|ModuleType|)
  ((parents :initform '(|Module|))
   (name :initform "GeneralModulePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'GMODPOL)
   (comment :initform (list
     "This package is undocumented"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralModulePolynomial|
  (progn
    (push '|GeneralModulePolynomial| *Domains*)
    (make-instance '|GeneralModulePolynomialType|)))
```

———-

1.43.3 GenericNonAssociativeAlgebra

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenericNonAssociativeAlgebra|
  (progn
    (push '|GenericNonAssociativeAlgebra| *Domains*)
    (make-instance '|GenericNonAssociativeAlgebraType|)))
```

———-

1.43.4 GeneralPolynomialSet

— sane —

```
(defclass |GeneralPolynomialSetType| (|PolynomialSetCategoryType|)
  ((parents :initform '(|PolynomialSetCategory|))
   (name :initform "GeneralPolynomialSet")
   (marker :initform 'domain)
   (abbreviation :initform 'GPOLSET)
   (comment :initform (list
     "A domain for polynomial sets."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralPolynomialSet|
  (progn
    (push '|GeneralPolynomialSet| *Domains*)
    (make-instance '|GeneralPolynomialSetType|)))
```

———-

1.43.5 GeneralSparseTable

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralSparseTable|
  (progn
    (push '|GeneralSparseTable| *Domains*)
    (make-instance '|GeneralSparseTableType|)))
```

———-

1.43.6 GeneralTriangularSet

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralTriangularSet|
  (progn
    (push '|GeneralTriangularSet| *Domains*)
```

```
    (make-instance '|GeneralTriangularSetType|)))
```

———-

1.43.7 GeneralUnivariatePowerSeries

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralUnivariatePowerSeries|
  (progn
    (push '|GeneralUnivariatePowerSeries| *Domains*)
    (make-instance '|GeneralUnivariatePowerSeriesType|)))
```

———-

1.43.8 GraphImage

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |GraphImage|
  (progn
    (push '|GraphImage| *Domains*)
    (make-instance '|GraphImageType|)))
```

———-

1.43.9 GuessOption

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessOption|
  (progn
    (push '|GuessOption| *Domains*)
    (make-instance '|GuessOptionType|)))
```

———-

1.43.10 GuessOptionFunctions0

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |GuessOptionFunctions0|
  (progn
    (push '|GuessOptionFunctions0| *Domains*)
    (make-instance '|GuessOptionFunctions0Type|)))
```

———-

1.44 H

1.44.1 HashTable

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HashTable|
   (progn
     (push '|HashTable| *Domains*)
     (make-instance '|HashTableType|)))
```

———-

1.44.2 Heap

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |Heap|
  (progn
    (push '|Heap| *Domains*)
    (make-instance '|HeapType|)))
```

———-

1.44.3 HexadecimalExpansion

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HexadecimalExpansion|
  (progn
    (push '|HexadecimalExpansion| *Domains*)
    (make-instance '|HexadecimalExpansionType|)))
```

———-

1.44.4 HTMLFormat

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |HTMLFormat|
  (progn
    (push '|HTMLFormat| *Domains*)
    (make-instance '|HTMLFormatType|)))
```

———-

1.44.5 HomogeneousDirectProduct

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HomogeneousDirectProduct|
  (progn
    (push '|HomogeneousDirectProduct| *Domains*)
    (make-instance '|HomogeneousDirectProductType|)))
```

———-

1.44.6 HomogeneousDistributedMultivariatePolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HomogeneousDistributedMultivariatePolynomial|
  (progn
    (push '|HomogeneousDistributedMultivariatePolynomial| *Domains*)
    (make-instance '|HomogeneousDistributedMultivariatePolynomialType|)))
```

———-

1.44.7 HyperellipticFiniteDivisor

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HyperellipticFiniteDivisor|
  (progn
    (push '|HyperellipticFiniteDivisor| *Domains*)
    (make-instance '|HyperellipticFiniteDivisorType|)))
```

———-

1.45 I

1.45.1 InfClsPt

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfClsPt|
  (progn
    (push '|InfClsPt| *Domains*)
    (make-instance '|InfClsPtType|)))
```

———-

1.45.2 IndexCard

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexCard|
  (progn
    (push '|IndexCard| *Domains*)
    (make-instance '|IndexCardType|)))
```

———-

1.45.3 IndexedBits

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedBits|
  (progn
    (push '|IndexedBits| *Domains*)
    (make-instance '|IndexedBitsType|)))
```

———-

1.45.4 IndexedDirectProductAbelianGroup

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductAbelianGroup|
  (progn
    (push '|IndexedDirectProductAbelianGroup| *Domains*)
    (make-instance '|IndexedDirectProductAbelianGroupType|)))
```

———-

1.45.5 IndexedDirectProductAbelianMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductAbelianMonoid|
  (progn
    (push '|IndexedDirectProductAbelianMonoid| *Domains*)
    (make-instance '|IndexedDirectProductAbelianMonoidType|)))
```

———-

1.45.6 IndexedDirectProductObject

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductObject|
  (progn
    (push '|IndexedDirectProductObject| *Domains*)
    (make-instance '|IndexedDirectProductObjectType|)))
```

———-

1.45.7 IndexedDirectProductOrderedAbelianMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductOrderedAbelianMonoid|
  (progn
    (push '|IndexedDirectProductOrderedAbelianMonoid| *Domains*)
    (make-instance '|IndexedDirectProductOrderedAbelianMonoidType|)))
```

———-

1.45.8 IndexedDirectProductOrderedAbelianMonoidSup

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedDirectProductOrderedAbelianMonoidSup|
  (progn
```

```
    (push '|IndexedDirectProductOrderedAbelianMonoidSup| *Domains*)
    (make-instance '|IndexedDirectProductOrderedAbelianMonoidSupType|)))
```

———-

1.45.9 IndexedExponents

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedExponents|
  (progn
    (push '|IndexedExponents| *Domains*)
    (make-instance '|IndexedExponentsType|)))
```

———-

1.45.10 IndexedFlexibleArray

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedFlexibleArray|
  (progn
    (push '|IndexedFlexibleArray| *Domains*)
    (make-instance '|IndexedFlexibleArrayType|)))
```

———-

1.45.11 IndexedList

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedList|
  (progn
    (push '|IndexedList| *Domains*)
    (make-instance '|IndexedListType|)))
```

———-

1.45.12 IndexedMatrix

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedMatrix|
  (progn
    (push '|IndexedMatrix| *Domains*)
    (make-instance '|IndexedMatrixType|)))
```

———-

1.45.13 IndexedOneDimensionalArray

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedOneDimensionalArray|
  (progn
    (push '|IndexedOneDimensionalArray| *Domains*)
```

```
    (make-instance '|IndexedOneDimensionalArrayType|)))
```

———-

1.45.14 IndexedString

— sane —

```
(defclass |IndexedStringType| (|StringAggregateType|)
  ((parents :initform '(|StringAggregate|))
   (name :initform "IndexedString")
   (marker :initform 'domain)
   (abbreviation :initform 'ISTRING)
   (comment :initform (list
     "This domain implements low-level strings"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedString|
  (progn
    (push '|IndexedString| *Domains*)
    (make-instance '|IndexedStringType|)))
```

———-

1.45.15 IndexedTwoDimensionalArray

— sane —

```
(defclass |IndexedTwoDimensionalArrayType| (|TwoDimensionalArrayCategoryType|)
  ((parents :initform '(|TwoDimensionalArrayCategory|))
   (name :initform "IndexedTwoDimensionalArray")
   (marker :initform 'domain)
   (abbreviation :initform 'IARRAY2)
   (comment :initform (list
     "This domain implements two dimensional arrays"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedTwoDimensionalArray|
  (progn
    (push '|IndexedTwoDimensionalArray| *Domains*)
    (make-instance '|IndexedTwoDimensionalArrayType|)))
```

———-

1.45.16 IndexedVector

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IndexedVector|
  (progn
    (push '|IndexedVector| *Domains*)
    (make-instance '|IndexedVectorType|)))
```

———-

1.45.17 InfiniteTuple

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteTuple|
  (progn
    (push '|InfiniteTuple| *Domains*)
    (make-instance '|InfiniteTupleType|)))
```

———-

1.45.18 InfinitlyClosePoint

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfinitlyClosePoint|
  (progn
    (push '|InfinitlyClosePoint| *Domains*)
    (make-instance '|InfinitlyClosePointType|)))
```

———-

1.45.19 InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.45.20 InnerAlgebraicNumber

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerAlgebraicNumber|
  (progn
    (push '|InnerAlgebraicNumber| *Domains*)
    (make-instance '|InnerAlgebraicNumberType|)))
```

———-

1.45.21 InnerFiniteField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |InnerFiniteField|
  (progn
    (push '|InnerFiniteField| *Domains*)
    (make-instance '|InnerFiniteFieldType|)))
```

———-

1.45.22 InnerFreeAbelianMonoid

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerFreeAbelianMonoid|
  (progn
    (push '|InnerFreeAbelianMonoid| *Domains*)
    (make-instance '|InnerFreeAbelianMonoidType|)))
```

———-

1.45.23 InnerIndexedTwoDimensionalArray

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |InnerIndexedTwoDimensionalArray|
  (progn
    (push '|InnerIndexedTwoDimensionalArray| *Domains*)
    (make-instance '|InnerIndexedTwoDimensionalArrayType|)))
```

———-

1.45.24 InnerPAdicInteger

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerPAdicInteger|
  (progn
    (push '|InnerPAdicInteger| *Domains*)
    (make-instance '|InnerPAdicIntegerType|)))
```

———-

1.45.25 InnerPrimeField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |InnerPrimeField|
  (progn
    (push '|InnerPrimeField| *Domains*)
    (make-instance '|InnerPrimeFieldType|)))
```

———-

1.45.26 InnerSparseUnivariatePowerSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerSparseUnivariatePowerSeries|
  (progn
    (push '|InnerSparseUnivariatePowerSeries| *Domains*)
    (make-instance '|InnerSparseUnivariatePowerSeriesType|)))
```

———-

1.45.27 InnerTable

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |InnerTable|
  (progn
    (push '|InnerTable| *Domains*)
    (make-instance '|InnerTableType|)))
```

———-

1.45.28 InnerTaylorSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerTaylorSeries|
  (progn
    (push '|InnerTaylorSeries| *Domains*)
    (make-instance '|InnerTaylorSeriesType|)))
```

———-

1.45.29 InputForm

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InputForm|
  (progn
    (push '|InputForm| *Domains*)
    (make-instance '|InputFormType|)))
```

———-

1.45.30 Integer

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Integer|
  (progn
    (push '|Integer| *Domains*)
    (make-instance '|IntegerType|)))
```

———-

1.45.31 IntegerMod

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerMod|
  (progn
    (push '|IntegerMod| *Domains*)
    (make-instance '|IntegerModType|)))
```

———-

1.45.32 IntegrationFunctionsTable

— **sane** —

```
(defclass |IntegrationFunctionsTableType| (|AxiomClass|)
  ((parents :initform '())
   (name :initform "IntegrationFunctionsTable")
   (marker :initform 'domain)
   (abbreviation :initform 'INTFTBL)
   (comment :initform (list
     "There is no description for this domain"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationFunctionsTable|
  (progn
    (push '|IntegrationFunctionsTable| *Domains*)
    (make-instance '|IntegrationFunctionsTableType|)))
```

———-

1.45.33 IntegrationResult

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResult|
  (progn
    (push '|IntegrationResult| *Domains*)
    (make-instance '|IntegrationResultType|)))
```

———-

1.45.34 Interval

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Interval|
  (progn
    (push '|Interval| *Domains*)
    (make-instance '|IntervalType|)))
```

———-

1.46 K

1.46.1 Kernel

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Kernel|
  (progn
    (push '|Kernel| *Domains*)
    (make-instance '|KernelType|)))
```

———-

1.46.2 KeyedAccessFile

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |KeyedAccessFile|
  (progn
    (push '|KeyedAccessFile| *Domains*)
    (make-instance '|KeyedAccessFileType|)))
```

———-

1.47 L

1.47.1 LaurentPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LaurentPolynomial|
  (progn
    (push '|LaurentPolynomial| *Domains*)
    (make-instance '|LaurentPolynomialType|)))
```

———-

1.47.2 Library

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Library|
  (progn
    (push '|Library| *Domains*)
    (make-instance '|LibraryType|)))
```

———-

1.47.3 LieExponentials

— **sane** —

```
(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. "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LieExponentials|
  (progn
    (push '|LieExponentials| *Domains*)
    (make-instance '|LieExponentialsType|)))
```

———-

1.47.4 LiePolynomial

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LiePolynomial|
  (progn
    (push '|LiePolynomial| *Domains*)
    (make-instance '|LiePolynomialType|)))
```

———-

1.47.5 LieSquareMatrix

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LieSquareMatrix|
  (progn
    (push '|LieSquareMatrix| *Domains*)
    (make-instance '|LieSquareMatrixType|)))
```

———-

1.47.6 LinearOrdinaryDifferentialOperator

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperator|
  (progn
    (push '|LinearOrdinaryDifferentialOperator| *Domains*)
    (make-instance '|LinearOrdinaryDifferentialOperatorType|)))
```

———-

1.47.7 LinearOrdinaryDifferentialOperator1

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperator1|
  (progn
    (push '|LinearOrdinaryDifferentialOperator1| *Domains*)
```

```
    (make-instance '|LinearOrdinaryDifferentialOperator1Type|)))
```

———-

1.47.8 LinearOrdinaryDifferentialOperator2

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperator2|
  (progn
    (push '|LinearOrdinaryDifferentialOperator2| *Domains*)
    (make-instance '|LinearOrdinaryDifferentialOperator2Type|)))
```

———-

1.47.9 List

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |List|
  (progn
    (push '|List| *Domains*)
    (make-instance '|ListType|)))
```

———-

1.47.10 ListMonoidOps

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListMonoidOps|
  (progn
    (push '|ListMonoidOps| *Domains*)
    (make-instance '|ListMonoidOpsType|)))
```

———-

1.47.11 ListMultiDictionary

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListMultiDictionary|
  (progn
    (push '|ListMultiDictionary| *Domains*)
    (make-instance '|ListMultiDictionaryType|)))
```

———-

1.47.12 LocalAlgebra

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LocalAlgebra|
  (progn
    (push '|LocalAlgebra| *Domains*)
    (make-instance '|LocalAlgebraType|)))
```

———-

1.47.13 Localize

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Localize|
  (progn
    (push '|Localize| *Domains*)
    (make-instance '|LocalizeType|)))
```

———-

1.47.14 LyndonWord

— **sane** —

```
(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. "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LyndonWord|
  (progn
    (push '|LyndonWord| *Domains*)
    (make-instance '|LyndonWordType|)))
```

———-

1.48 M

1.48.1 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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MachineComplex|
  (progn
    (push '|MachineComplex| *Domains*)
    (make-instance '|MachineComplexType|)))
```

————-

1.48.2 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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MachineFloat|
  (progn
    (push '|MachineFloat| *Domains*)
    (make-instance '|MachineFloatType|)))
```

———-

1.48.3 MachineInteger

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MachineInteger|
  (progn
    (push '|MachineInteger| *Domains*)
    (make-instance '|MachineIntegerType|)))
```

———-

1.48.4 Magma

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Magma|
  (progn
    (push '|Magma| *Domains*)
    (make-instance '|MagmaType|)))
```

———-

1.48.5 MakeCachableSet

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeCachableSet|
  (progn
    (push '|MakeCachableSet| *Domains*)
    (make-instance '|MakeCachableSetType|)))
```

———-

1.48.6 MathMLFormat

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MathMLFormat|
  (progn
    (push '|MathMLFormat| *Domains*)
    (make-instance '|MathMLFormatType|)))
```

———-

1.48.7 Matrix

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Matrix|
  (progn
    (push '|Matrix| *Domains*)
    (make-instance '|MatrixType|)))
```

———-

1.48.8 ModMonic

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModMonic|
  (progn
    (push '|ModMonic| *Domains*)
    (make-instance '|ModMonicType|)))
```

———-

1.48.9 ModularField

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularField|
  (progn
    (push '|ModularField| *Domains*)
    (make-instance '|ModularFieldType|)))
```

———-

1.48.10 ModularRing

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularRing|
  (progn
    (push '|ModularRing| *Domains*)
    (make-instance '|ModularRingType|)))
```

———-

1.48.11 ModuleMonomial

— sane —

```
(defclass |ModuleMonomialType| (|OrderedSetType|)
  ((parents :initform '(|OrderedSet|))
   (name :initform "ModuleMonomial")
   (marker :initform 'domain)
   (abbreviation :initform 'MODMONOM)
   (comment :initform (list
     "This package has no documentation"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModuleMonomial|
  (progn
    (push '|ModuleMonomial| *Domains*)
    (make-instance '|ModuleMonomialType|)))
```

———-

1.48.12 ModuleOperator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |ModuleOperator|
  (progn
    (push '|ModuleOperator| *Domains*)
    (make-instance '|ModuleOperatorType|)))
```

———-

1.48.13 MoebiusTransform

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MoebiusTransform|
  (progn
    (push '|MoebiusTransform| *Domains*)
    (make-instance '|MoebiusTransformType|)))
```

———-

1.48.14 MonoidRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonoidRing|
  (progn
    (push '|MonoidRing| *Domains*)
    (make-instance '|MonoidRingType|)))
```

———-

1.48.15 Multiset

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Multiset|
  (progn
    (push '|Multiset| *Domains*)
    (make-instance '|MultisetType|)))
```

———-

1.48.16 MultivariatePolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariatePolynomial|
  (progn
    (push '|MultivariatePolynomial| *Domains*)
    (make-instance '|MultivariatePolynomialType|)))
```

———-

1.48.17 MyExpression

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MyExpression|
  (progn
    (push '|MyExpression| *Domains*)
    (make-instance '|MyExpressionType|)))
```

———-

1.48.18 MyUnivariatePolynomial

— sane —

```
(defclass |MyUnivariatePolynomialType| (|UnivariatePolynomialCategoryType|)
  ((parents :initform '(|UnivariatePolynomialCategory|))
   (name :initform "MyUnivariatePolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'MYUP)
   (comment :initform (list
     "This domain has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MyUnivariatePolynomial|
  (progn
    (push '|MyUnivariatePolynomial| *Domains*)
    (make-instance '|MyUnivariatePolynomialType|)))
```

———-

1.49 N

1.49.1 NeitherSparseOrDensePowerSeries

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NeitherSparseOrDensePowerSeries|
  (progn
```

```
    (push '|NeitherSparseOrDensePowerSeries| *Domains*)
    (make-instance '|NeitherSparseOrDensePowerSeriesType|)))
```

———-

1.49.2 NewSparseMultivariatePolynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewSparseMultivariatePolynomial|
  (progn
    (push '|NewSparseMultivariatePolynomial| *Domains*)
    (make-instance '|NewSparseMultivariatePolynomialType|)))
```

———-

1.49.3 NewSparseUnivariatePolynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |NewSparseUnivariatePolynomial|
  (progn
    (push '|NewSparseUnivariatePolynomial| *Domains*)
    (make-instance '|NewSparseUnivariatePolynomialType|)))
```

———-

1.49.4 None

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |None|
  (progn
    (push '|None| *Domains*)
    (make-instance '|NoneType|)))
```

———-

1.49.5 NonNegativeInteger

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |NonNegativeInteger|
  (progn
    (push '|NonNegativeInteger| *Domains*)
    (make-instance '|NonNegativeIntegerType|)))
```

———-

1.49.6 NottinghamGroup

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NottinghamGroup|
  (progn
    (push '|NottinghamGroup| *Domains*)
    (make-instance '|NottinghamGroupType|)))
```

———-

1.49.7 NumericalIntegrationProblem

— sane —

```
(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,)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalIntegrationProblem|
  (progn
    (push '|NumericalIntegrationProblem| *Domains*)
    (make-instance '|NumericalIntegrationProblemType|)))
```

———-

1.49.8 NumericalODEProblem

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |NumericalODEProblem|
  (progn
    (push '|NumericalODEProblem| *Domains*)
    (make-instance '|NumericalODEProblemType|)))
```

———-

1.49.9 NumericalOptimizationProblem

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalOptimizationProblem|
  (progn
    (push '|NumericalOptimizationProblem| *Domains*)
    (make-instance '|NumericalOptimizationProblemType|)))
```

———-

1.49.10 NumericalPDEProblem

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalPDEProblem|
  (progn
    (push '|NumericalPDEProblem| *Domains*)
    (make-instance '|NumericalPDEProblemType|)))
```

———-

1.50 O

1.50.1 Octonion

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Octonion|
  (progn
    (push '|Octonion| *Domains*)
    (make-instance '|OctonionType|)))
```

———-

1.50.2 ODEIntensityFunctionsTable

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ODEIntensityFunctionsTable|
  (progn
    (push '|ODEIntensityFunctionsTable| *Domains*)
    (make-instance '|ODEIntensityFunctionsTableType|)))
```

———-

1.50.3 OneDimensionalArray

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OneDimensionalArray|
  (progn
    (push '|OneDimensionalArray| *Domains*)
    (make-instance '|OneDimensionalArrayType|)))
```

———-

1.50.4 OnePointCompletion

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OnePointCompletion|
  (progn
    (push '|OnePointCompletion| *Domains*)
    (make-instance '|OnePointCompletionType|)))
```

———-

1.50.5 OpenMathConnection

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathConnection|
  (progn
    (push '|OpenMathConnection| *Domains*)
    (make-instance '|OpenMathConnectionType|)))
```

———-

1.50.6 OpenMathDevice

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathDevice|
  (progn
    (push '|OpenMathDevice| *Domains*)
    (make-instance '|OpenMathDeviceType|)))
```

———-

1.50.7 OpenMathEncoding

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathEncoding|
  (progn
    (push '|OpenMathEncoding| *Domains*)
    (make-instance '|OpenMathEncodingType|)))
```

———-

1.50.8 OpenMathError

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathError|
  (progn
    (push '|OpenMathError| *Domains*)
    (make-instance '|OpenMathErrorType|)))
```

———-

1.50.9 OpenMathErrorKind

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathErrorKind|
  (progn
    (push '|OpenMathErrorKind| *Domains*)
    (make-instance '|OpenMathErrorKindType|)))
```

———-

1.50.10 Operator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Operator|
  (progn
    (push '|Operator| *Domains*)
    (make-instance '|OperatorType|)))
```

———-

1.50.11 OppositeMonogenicLinearOperator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OppositeMonogenicLinearOperator|
  (progn
    (push '|OppositeMonogenicLinearOperator| *Domains*)
    (make-instance '|OppositeMonogenicLinearOperatorType|)))
```

———-

1.50.12 OrderedCompletion

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |OrderedCompletion|
  (progn
    (push '|OrderedCompletion| *Domains*)
    (make-instance '|OrderedCompletionType|)))
```

———-

1.50.13 OrderedDirectProduct

— sane —

```
(defclass |OrderedDirectProductType| (|DirectProductCategoryType|)
  ((parents :initform '(|DirectProductCategory|))
   (name :initform "OrderedDirectProduct")
   (marker :initform 'domain)
   (abbreviation :initform 'ODP)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedDirectProduct|
  (progn
    (push '|OrderedDirectProduct| *Domains*)
    (make-instance '|OrderedDirectProductType|)))
```

———-

1.50.14 OrderedFreeMonoid

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedFreeMonoid|
  (progn
    (push '|OrderedFreeMonoid| *Domains*)
    (make-instance '|OrderedFreeMonoidType|)))
```

———-

1.50.15 OrderedVariableList

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedVariableList|
  (progn
    (push '|OrderedVariableList| *Domains*)
    (make-instance '|OrderedVariableListType|)))
```

———-

1.50.16 OrderlyDifferentialPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderlyDifferentialPolynomial|
  (progn
    (push '|OrderlyDifferentialPolynomial| *Domains*)
    (make-instance '|OrderlyDifferentialPolynomialType|)))
```

———-

1.50.17 OrderlyDifferentialVariable

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderlyDifferentialVariable|
  (progn
    (push '|OrderlyDifferentialVariable| *Domains*)
    (make-instance '|OrderlyDifferentialVariableType|)))
```

———-

1.50.18 OrdinaryDifferentialRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdinaryDifferentialRing|
  (progn
    (push '|OrdinaryDifferentialRing| *Domains*)
    (make-instance '|OrdinaryDifferentialRingType|)))
```

———-

1.50.19 OrdinaryWeightedPolynomials

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdinaryWeightedPolynomials|
  (progn
    (push '|OrdinaryWeightedPolynomials| *Domains*)
    (make-instance '|OrdinaryWeightedPolynomialsType|)))
```

———-

1.50.20 OrdSetInts

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrdSetInts|
  (progn
    (push '|OrdSetInts| *Domains*)
    (make-instance '|OrdSetIntsType|)))
```

———-

1.50.21 OutputForm

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OutputForm|
  (progn
    (push '|OutputForm| *Domains*)
```

```
    (make-instance '|OutputFormType|)))
```

———-

1.51 P

1.51.1 PAdicInteger

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicInteger|
  (progn
    (push '|PAdicInteger| *Domains*)
    (make-instance '|PAdicIntegerType|)))
```

———-

1.51.2 PAdicRational

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PAdicRational|
  (progn
    (push '|PAdicRational| *Domains*)
    (make-instance '|PAdicRationalType|)))
```

———-

1.51.3 PAdicRationalConstructor

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicRationalConstructor|
  (progn
    (push '|PAdicRationalConstructor| *Domains*)
    (make-instance '|PAdicRationalConstructorType|)))
```

———-

1.51.4 Palette

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Palette|
  (progn
```

```
    (push '|Palette| *Domains*)
    (make-instance '|PaletteType|)))
```

———-

1.51.5 ParametricPlaneCurve

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricPlaneCurve|
  (progn
    (push '|ParametricPlaneCurve| *Domains*)
    (make-instance '|ParametricPlaneCurveType|)))
```

———-

1.51.6 ParametricSpaceCurve

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSpaceCurve|
  (progn
```

```
    (push '|ParametricSpaceCurve| *Domains*)
    (make-instance '|ParametricSpaceCurveType|)))
```

———-

1.51.7 ParametricSurface

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSurface|
  (progn
    (push '|ParametricSurface| *Domains*)
    (make-instance '|ParametricSurfaceType|)))
```

———-

1.51.8 PartialFraction

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialFraction|
  (progn
    (push '|PartialFraction| *Domains*)
    (make-instance '|PartialFractionType|)))
```

———-

1.51.9 Partition

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Partition|
  (progn
    (push '|Partition| *Domains*)
    (make-instance '|PartitionType|)))
```

———-

1.51.10 Pattern

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Pattern|
  (progn
    (push '|Pattern| *Domains*)
    (make-instance '|PatternType|)))
```

———-

1.51.11 PatternMatchListResult

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchListResult|
  (progn
    (push '|PatternMatchListResult| *Domains*)
    (make-instance '|PatternMatchListResultType|)))
```

———-

1.51.12 PatternMatchResult

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchResult|
  (progn
    (push '|PatternMatchResult| *Domains*)
    (make-instance '|PatternMatchResultType|)))
```

———-

1.51.13 PendantTree

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PendantTree|
  (progn
    (push '|PendantTree| *Domains*)
    (make-instance '|PendantTreeType|)))
```

———-

1.51.14 Permutation

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Permutation|
  (progn
    (push '|Permutation| *Domains*)
    (make-instance '|PermutationType|)))
```

———-

1.51.15 PermutationGroup

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PermutationGroup|
  (progn
```

```
    (push '|PermutationGroup| *Domains*)
    (make-instance '|PermutationGroupType|)))
```

———-

1.51.16 Pi

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Pi|
  (progn
    (push '|Pi| *Domains*)
    (make-instance '|PiType|)))
```

———-

1.51.17 PlaneAlgebraicCurvePlot

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |PlaneAlgebraicCurvePlot|
  (progn
    (push '|PlaneAlgebraicCurvePlot| *Domains*)
    (make-instance '|PlaneAlgebraicCurvePlotType|)))
```

———-

1.51.18 Places

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Places|
  (progn
    (push '|Places| *Domains*)
    (make-instance '|PlacesType|)))
```

———-

1.51.19 PlacesOverPseudoAlgebraicClosureOfFiniteField

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |PlacesOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|PlacesOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|PlacesOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.51.20 Plcs

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Plcs|
  (progn
    (push '|Plcs| *Domains*)
    (make-instance '|PlcsType|)))
```

———-

1.51.21 Plot

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |Plot|
  (progn
    (push '|Plot| *Domains*)
    (make-instance '|PlotType|)))
```

———-

1.51.22 Plot3D

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Plot3D|
  (progn
    (push '|Plot3D| *Domains*)
    (make-instance '|Plot3DType|)))
```

———-

1.51.23 PoincareBirkhoffWittLyndonBasis

— sane —

```
(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 "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PoincareBirkhoffWittLyndonBasis|
  (progn
    (push '|PoincareBirkhoffWittLyndonBasis| *Domains*)
    (make-instance '|PoincareBirkhoffWittLyndonBasisType|)))
```

———-

1.51.24 Point

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Point|
  (progn
    (push '|Point| *Domains*)
    (make-instance '|PointType|)))
```

———-

1.51.25 Polynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Polynomial|
  (progn
    (push '|Polynomial| *Domains*)
    (make-instance '|PolynomialType|)))
```

———-

1.51.26 PolynomialIdeals

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialIdeals|
  (progn
    (push '|PolynomialIdeals| *Domains*)
    (make-instance '|PolynomialIdealsType|)))
```

———-

1.51.27 PolynomialRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialRing|
  (progn
    (push '|PolynomialRing| *Domains*)
    (make-instance '|PolynomialRingType|)))
```

———-

1.51.28 PositiveInteger

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PositiveInteger|
  (progn
    (push '|PositiveInteger| *Domains*)
    (make-instance '|PositiveIntegerType|)))
```

———-

1.51.29 PrimeField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimeField|
  (progn
    (push '|PrimeField| *Domains*)
    (make-instance '|PrimeFieldType|)))
```

———-

1.51.30 PrimitiveArray

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveArray|
  (progn
    (push '|PrimitiveArray| *Domains*)
```

```
    (make-instance '|PrimitiveArrayType|)))
```

———-

1.51.31 Product

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Product|
  (progn
    (push '|Product| *Domains*)
    (make-instance '|ProductType|)))
```

———-

1.51.32 ProjectivePlane

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectivePlane|
  (progn
```

```
    (push '|ProjectivePlane| *Domains*)
    (make-instance '|ProjectivePlaneType|)))
```

———-

1.51.33 ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.51.34 ProjectiveSpace

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectiveSpace|
  (progn
    (push '|ProjectiveSpace| *Domains*)
    (make-instance '|ProjectiveSpaceType|)))
```

———-

1.51.35 PseudoAlgebraicClosureOfAlgExtOfRationalNumber

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfAlgExtOfRationalNumber|
  (progn
    (push '|PseudoAlgebraicClosureOfAlgExtOfRationalNumber| *Domains*)
    (make-instance '|PseudoAlgebraicClosureOfAlgExtOfRationalNumberType|)))
```

———-

1.51.36 PseudoAlgebraicClosureOfFiniteField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoAlgebraicClosureOfFiniteField|
  (progn
    (push '|PseudoAlgebraicClosureOfFiniteField| *Domains*)
    (make-instance '|PseudoAlgebraicClosureOfFiniteFieldType|)))
```

———-

1.51.37 PseudoAlgebraicClosureOfRationalNumber

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PseudoAlgebraicClosureOfRationalNumber|
  (progn
    (push '|PseudoAlgebraicClosureOfRationalNumber| *Domains*)
    (make-instance '|PseudoAlgebraicClosureOfRationalNumberType|)))
```

———-

1.52 Q

1.52.1 QuadraticForm

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuadraticForm|
  (progn
    (push '|QuadraticForm| *Domains*)
    (make-instance '|QuadraticFormType|)))
```

———-

1.52.2 QuasiAlgebraicSet

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuasiAlgebraicSet|
  (progn
    (push '|QuasiAlgebraicSet| *Domains*)
    (make-instance '|QuasiAlgebraicSetType|)))
```

———-

1.52.3 Quaternion

— **sane** —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Quaternion|
  (progn
    (push '|Quaternion| *Domains*)
    (make-instance '|QuaternionType|)))
```

———-

1.52.4 QueryEquation

— sane —

```
(defclass |QueryEquationType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "QueryEquation")
   (marker :initform 'domain)
   (abbreviation :initform 'QEQUAT)
   (comment :initform (list
     "This domain implements simple database queries"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QueryEquation|
  (progn
    (push '|QueryEquation| *Domains*)
    (make-instance '|QueryEquationType|)))
```

———-

1.52.5 Queue

— sane —

```
(defclass |QueueType| (|QueueAggregateType|)
  ((parents :initform '(|QueueAggregate|))
   (name :initform "Queue")
   (marker :initform 'domain)
   (abbreviation :initform 'QUEUE)
   (comment :initform (list
     "Linked List implementation of a Queue"))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |Queue|
  (progn
    (push '|Queue| *Domains*)
    (make-instance '|QueueType|)))
```

———-

1.53 R

1.53.1 RadicalFunctionField

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadicalFunctionField|
  (progn
    (push '|RadicalFunctionField| *Domains*)
    (make-instance '|RadicalFunctionFieldType|)))
```

———-

1.53.2 RadixExpansion

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadixExpansion|
  (progn
    (push '|RadixExpansion| *Domains*)
    (make-instance '|RadixExpansionType|)))
```

———-

1.53.3 RealClosure

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealClosure|
  (progn
    (push '|RealClosure| *Domains*)
    (make-instance '|RealClosureType|)))
```

———-

1.53.4 RectangularMatrix

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RectangularMatrix|
  (progn
    (push '|RectangularMatrix| *Domains*)
    (make-instance '|RectangularMatrixType|)))
```

———-

1.53.5 Reference

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Reference|
  (progn
    (push '|Reference| *Domains*)
    (make-instance '|ReferenceType|)))
```

———-

1.53.6 RegularChain

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularChain|
  (progn
    (push '|RegularChain| *Domains*)
    (make-instance '|RegularChainType|)))
```

———-

1.53.7 RegularTriangularSet

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularTriangularSet|
  (progn
    (push '|RegularTriangularSet| *Domains*)
    (make-instance '|RegularTriangularSetType|)))
```

———-

1.53.8 ResidueRing

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ResidueRing|
  (progn
    (push '|ResidueRing| *Domains*)
    (make-instance '|ResidueRingType|)))
```

———-

1.53.9 Result

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Result|
  (progn
    (push '|Result| *Domains*)
    (make-instance '|ResultType|)))
```

———-

1.53.10 RewriteRule

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RewriteRule|
  (progn
    (push '|RewriteRule| *Domains*)
    (make-instance '|RewriteRuleType|)))
```

———-

1.53.11 RightOpenIntervalRootCharacterization

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RightOpenIntervalRootCharacterization|
  (progn
    (push '|RightOpenIntervalRootCharacterization| *Domains*)
    (make-instance '|RightOpenIntervalRootCharacterizationType|)))
```

———-

1.53.12 RomanNumeral

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RomanNumeral|
  (progn
    (push '|RomanNumeral| *Domains*)
    (make-instance '|RomanNumeralType|)))
```

———-

1.53.13 RoutinesTable

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RoutinesTable|
  (progn
    (push '|RoutinesTable| *Domains*)
    (make-instance '|RoutinesTableType|)))
```

———-

1.53.14 RuleCalled

— sane —

```
(defclass |RuleCalledType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "RuleCalled")
   (marker :initform 'domain)
   (abbreviation :initform 'RULECOLD)
   (comment :initform (list
     "This domain implements named rules "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RuleCalled|
  (progn
    (push '|RuleCalled| *Domains*)
    (make-instance '|RuleCalledType|)))
```

———-

1.53.15 Ruleset

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Ruleset|
  (progn
    (push '|Ruleset| *Domains*)
    (make-instance '|RulesetType|)))
```

———-

1.54 S

1.54.1 ScriptFormulaFormat

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ScriptFormulaFormat|
  (progn
    (push '|ScriptFormulaFormat| *Domains*)
    (make-instance '|ScriptFormulaFormatType|)))
```

———-

1.54.2 Segment

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |Segment|
  (progn
    (push '|Segment| *Domains*)
    (make-instance '|SegmentType|)))
```

———-

1.54.3 SegmentBinding

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SegmentBinding|
  (progn
    (push '|SegmentBinding| *Domains*)
    (make-instance '|SegmentBindingType|)))
```

———-

1.54.4 Set

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Set|
  (progn
    (push '|Set| *Domains*)
    (make-instance '|SetType|)))
```

———-

1.54.5 SetOfMIntegersInOneToN

— sane —

```
(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]"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SetOfMIntegersInOneToN|
  (progn
    (push '|SetOfMIntegersInOneToN| *Domains*)
    (make-instance '|SetOfMIntegersInOneToNType|)))
```

———-

1.54.6 SequentialDifferentialPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SequentialDifferentialPolynomial|
  (progn
    (push '|SequentialDifferentialPolynomial| *Domains*)
    (make-instance '|SequentialDifferentialPolynomialType|)))
```

———-

1.54.7 SequentialDifferentialVariable

— sane —

```
(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))."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |SequentialDifferentialVariable|
  (progn
    (push '|SequentialDifferentialVariable| *Domains*)
    (make-instance '|SequentialDifferentialVariableType|)))
```

———-

1.54.8 SExpression

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SExpression|
  (progn
    (push '|SExpression| *Domains*)
    (make-instance '|SExpressionType|)))
```

———-

1.54.9 SExpressionOf

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |SExpressionOf|
  (progn
    (push '|SExpressionOf| *Domains*)
    (make-instance '|SExpressionOfType|)))
```

———-

1.54.10 SimpleAlgebraicExtension

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleAlgebraicExtension|
  (progn
    (push '|SimpleAlgebraicExtension| *Domains*)
    (make-instance '|SimpleAlgebraicExtensionType|)))
```

———-

1.54.11 SimpleCell

— **sane** —

```
(defclass |SimpleCellType| (|CoercibleToType|)
  ((parents :initform '(|CoercibleTo|))
   (name :initform "SimpleCell")
```

```
   (marker :initform 'domain)
   (abbreviation :initform 'SCELL)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleCell|
  (progn
    (push '|SimpleCell| *Domains*)
    (make-instance '|SimpleCellType|)))
```

———-

1.54.12 SimpleFortranProgram

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleFortranProgram|
  (progn
    (push '|SimpleFortranProgram| *Domains*)
    (make-instance '|SimpleFortranProgramType|)))
```

———-

1.54.13 SingleInteger

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SingleInteger|
  (progn
    (push '|SingleInteger| *Domains*)
    (make-instance '|SingleIntegerType|)))
```

———-

1.54.14 SingletonAsOrderedSet

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SingletonAsOrderedSet|
  (progn
    (push '|SingletonAsOrderedSet| *Domains*)
    (make-instance '|SingletonAsOrderedSetType|)))
```

———-

1.54.15 SparseEchelonMatrix

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseEchelonMatrix|
  (progn
    (push '|SparseEchelonMatrix| *Domains*)
    (make-instance '|SparseEchelonMatrixType|)))
```

———-

1.54.16 SparseMultivariatePolynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseMultivariatePolynomial|
  (progn
    (push '|SparseMultivariatePolynomial| *Domains*)
    (make-instance '|SparseMultivariatePolynomialType|)))
```

———-

1.54.17 SparseMultivariateTaylorSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseMultivariateTaylorSeries|
  (progn
    (push '|SparseMultivariateTaylorSeries| *Domains*)
    (make-instance '|SparseMultivariateTaylorSeriesType|)))
```

———-

1.54.18 SparseTable

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseTable|
  (progn
    (push '|SparseTable| *Domains*)
    (make-instance '|SparseTableType|)))
```

———-

1.54.19 SparseUnivariateLaurentSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariateLaurentSeries|
  (progn
    (push '|SparseUnivariateLaurentSeries| *Domains*)
    (make-instance '|SparseUnivariateLaurentSeriesType|)))
```

———-

1.54.20 SparseUnivariatePolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |SparseUnivariatePolynomial|
  (progn
    (push '|SparseUnivariatePolynomial| *Domains*)
    (make-instance '|SparseUnivariatePolynomialType|)))
```

———-

1.54.21 SparseUnivariatePolynomialExpressions

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariatePolynomialExpressions|
  (progn
    (push '|SparseUnivariatePolynomialExpressions| *Domains*)
    (make-instance '|SparseUnivariatePolynomialExpressionsType|)))
```

———-

1.54.22 SparseUnivariatePuiseuxSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariatePuiseuxSeries|
  (progn
    (push '|SparseUnivariatePuiseuxSeries| *Domains*)
    (make-instance '|SparseUnivariatePuiseuxSeriesType|)))
```

———-

1.54.23 SparseUnivariateSkewPolynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariateSkewPolynomial|
  (progn
    (push '|SparseUnivariateSkewPolynomial| *Domains*)
    (make-instance '|SparseUnivariateSkewPolynomialType|)))
```

———-

1.54.24 SparseUnivariateTaylorSeries

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariateTaylorSeries|
  (progn
    (push '|SparseUnivariateTaylorSeries| *Domains*)
    (make-instance '|SparseUnivariateTaylorSeriesType|)))
```

———-

1.54.25 SplitHomogeneousDirectProduct

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SplitHomogeneousDirectProduct|
  (progn
    (push '|SplitHomogeneousDirectProduct| *Domains*)
    (make-instance '|SplitHomogeneousDirectProductType|)))
```

———-

1.54.26 SplittingNode

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SplittingNode|
  (progn
    (push '|SplittingNode| *Domains*)
    (make-instance '|SplittingNodeType|)))
```

———-

1.54.27 SplittingTree

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SplittingTree|
  (progn
    (push '|SplittingTree| *Domains*)
    (make-instance '|SplittingTreeType|)))
```

———-

1.54.28 SquareFreeRegularTriangularSet

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularTriangularSet|
  (progn
    (push '|SquareFreeRegularTriangularSet| *Domains*)
    (make-instance '|SquareFreeRegularTriangularSetType|)))
```

———-

1.54.29 SquareMatrix

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareMatrix|
  (progn
    (push '|SquareMatrix| *Domains*)
    (make-instance '|SquareMatrixType|)))
```

———-

1.54.30 Stack

— sane —

```
(defclass |StackType| (|StackAggregateType|)
  ((parents :initform '(|StackAggregate|))
   (name :initform "Stack")
   (marker :initform 'domain)
   (abbreviation :initform 'STACK)
   (comment :initform (list
     "Linked List implementation of a Stack"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Stack|
  (progn
    (push '|Stack| *Domains*)
    (make-instance '|StackType|)))
```

———-

1.54.31 StochasticDifferential

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StochasticDifferential|
  (progn
    (push '|StochasticDifferential| *Domains*)
    (make-instance '|StochasticDifferentialType|)))
```

———-

1.54.32 Stream

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Stream|
  (progn
```

```
    (push '|Stream| *Domains*)
    (make-instance '|StreamType|)))
```

———-

1.54.33 String

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |String|
  (progn
    (push '|String| *Domains*)
    (make-instance '|StringType|)))
```

———-

1.54.34 StringTable

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StringTable|
  (progn
    (push '|StringTable| *Domains*)
```

```
    (make-instance '|StringTableType|)))
```

———-

1.54.35 SubSpace

— **sane** —

```
(defclass |SubSpaceType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SubSpace")
   (marker :initform 'domain)
   (abbreviation :initform 'SUBSPACE)
   (comment :initform (list
     "This domain is not documented"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SubSpace|
  (progn
    (push '|SubSpace| *Domains*)
    (make-instance '|SubSpaceType|)))
```

———-

1.54.36 SubSpaceComponentProperty

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SubSpaceComponentProperty|
  (progn
    (push '|SubSpaceComponentProperty| *Domains*)
    (make-instance '|SubSpaceComponentPropertyType|)))
```

———-

1.54.37 SuchThat

— sane —

```
(defclass |SuchThatType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "SuchThat")
   (marker :initform 'domain)
   (abbreviation :initform 'SUCH)
   (comment :initform (list
     "This domain implements 'such that' forms"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SuchThat|
  (progn
    (push '|SuchThat| *Domains*)
    (make-instance '|SuchThatType|)))
```

———-

1.54.38 Switch

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Switch|
  (progn
    (push '|Switch| *Domains*)
    (make-instance '|SwitchType|)))
```

———-

1.54.39 Symbol

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Symbol|
  (progn
    (push '|Symbol| *Domains*)
    (make-instance '|SymbolType|)))
```

———-

1.54.40 SymbolTable

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SymbolTable|
  (progn
    (push '|SymbolTable| *Domains*)
    (make-instance '|SymbolTableType|)))
```

———-

1.54.41 SymmetricPolynomial

— sane —

```
(defclass |SymmetricPolynomialType| (|FiniteAbelianMonoidRingType|)
  ((parents :initform '(|FiniteAbelianMonoidRing|))
   (name :initform "SymmetricPolynomial")
   (marker :initform 'domain)
   (abbreviation :initform 'SYMPOLY)
   (comment :initform (list
     "This domain implements symmetric polynomial"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SymmetricPolynomial|
  (progn
    (push '|SymmetricPolynomial| *Domains*)
    (make-instance '|SymmetricPolynomialType|)))
```

———-

1.55 T

1.55.1 Table

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |Table|
  (progn
    (push '|Table| *Domains*)
    (make-instance '|TableType|)))
```

———-

1.55.2 Tableau

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Tableau|
  (progn
    (push '|Tableau| *Domains*)
    (make-instance '|TableauType|)))
```

———-

1.55.3 TaylorSerieso

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |TaylorSerieso|
  (progn
    (push '|TaylorSerieso| *Domains*)
    (make-instance '|TaylorSeriesoType|)))
```

———-

1.55.4 TexFormat

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TexFormat|
  (progn
    (push '|TexFormat| *Domains*)
    (make-instance '|TexFormatType|)))
```

———-

1.55.5 TextFile

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TextFile|
  (progn
    (push '|TextFile| *Domains*)
    (make-instance '|TextFileType|)))
```

———-

1.55.6 TheSymbolTable

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TheSymbolTable|
  (progn
    (push '|TheSymbolTable| *Domains*)
    (make-instance '|TheSymbolTableType|)))
```

———-

1.55.7 ThreeDimensionalMatrix

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ThreeDimensionalMatrix|
  (progn
    (push '|ThreeDimensionalMatrix| *Domains*)
    (make-instance '|ThreeDimensionalMatrixType|)))
```

———-

1.55.8 ThreeDimensionalViewport

— **sane** —

```
(defclass |ThreeDimensionalViewportType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "ThreeDimensionalViewport")
   (marker :initform 'domain)
   (abbreviation :initform 'VIEW3D)
   (comment :initform (list
     "ThreeDimensionalViewport creates viewports to display graphs"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ThreeDimensionalViewport|
  (progn
    (push '|ThreeDimensionalViewport| *Domains*)
    (make-instance '|ThreeDimensionalViewportType|)))
```

———-

1.55.9 ThreeSpace

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ThreeSpace|
  (progn
    (push '|ThreeSpace| *Domains*)
    (make-instance '|ThreeSpaceType|)))
```

———-

1.55.10 Tree

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Tree|
  (progn
    (push '|Tree| *Domains*)
    (make-instance '|TreeType|)))
```

———-

1.55.11 TubePlot

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TubePlot|
  (progn
    (push '|TubePlot| *Domains*)
    (make-instance '|TubePlotType|)))
```

———-

1.55.12 Tuple

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Tuple|
  (progn
    (push '|Tuple| *Domains*)
    (make-instance '|TupleType|)))
```

———-

1.55.13 TwoDimensionalArray

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoDimensionalArray|
  (progn
    (push '|TwoDimensionalArray| *Domains*)
    (make-instance '|TwoDimensionalArrayType|)))
```

———-

1.55.14 TwoDimensionalViewport

— **sane** —

```
(defclass |TwoDimensionalViewportType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "TwoDimensionalViewport")
   (marker :initform 'domain)
   (abbreviation :initform 'VIEW2D)
   (comment :initform (list
     "TwoDimensionalViewport creates viewports to display graphs."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoDimensionalViewport|
  (progn
    (push '|TwoDimensionalViewport| *Domains*)
    (make-instance '|TwoDimensionalViewportType|)))
```

———-

1.56 U

1.56.1 UnivariateFormalPowerSeries

— **sane** —

```
(defclass |UnivariateFormalPowerSeriesType| (|UnivariateTaylorSeriesCategoryType|)
```

```
  ((parents :initform '(|UnivariateTaylorSeriesCategory|))
   (name :initform "UnivariateFormalPowerSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'UFPS)
   (comment :initform (list
     "This domain has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateFormalPowerSeries|
  (progn
    (push '|UnivariateFormalPowerSeries| *Domains*)
    (make-instance '|UnivariateFormalPowerSeriesType|)))
```

———-

1.56.2 UnivariateLaurentSeries

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeries|
  (progn
    (push '|UnivariateLaurentSeries| *Domains*)
    (make-instance '|UnivariateLaurentSeriesType|)))
```

———-

1.56.3 UnivariateLaurentSeriesConstructor

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeriesConstructor|
  (progn
    (push '|UnivariateLaurentSeriesConstructor| *Domains*)
    (make-instance '|UnivariateLaurentSeriesConstructorType|)))
```

———-

1.56.4 UnivariatePolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomial|
  (progn
    (push '|UnivariatePolynomial| *Domains*)
    (make-instance '|UnivariatePolynomialType|)))
```

———-

1.56.5 UnivariatePuiseuxSeries

— sane —

```
(defclass |UnivariatePuiseuxSeriesType| (|UnivariatePuiseuxSeriesConstructorCategoryType|)
  ((parents :initform '(|UnivariatePuiseuxSeriesConstructorCategory|))
   (name :initform "UnivariatePuiseuxSeries")
   (marker :initform 'domain)
   (abbreviation :initform 'UPXS)
   (comment :initform (list
     "Dense Puiseux series in one variable"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeries|
  (progn
    (push '|UnivariatePuiseuxSeries| *Domains*)
    (make-instance '|UnivariatePuiseuxSeriesType|)))
```

———-

1.56.6 UnivariatePuiseuxSeriesConstructor

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesConstructor|
  (progn
    (push '|UnivariatePuiseuxSeriesConstructor| *Domains*)
    (make-instance '|UnivariatePuiseuxSeriesConstructorType|)))
```

———-

1.56.7 UnivariatePuiseuxSeriesWithExponentialSingularity

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesWithExponentialSingularity|
  (progn
    (push '|UnivariatePuiseuxSeriesWithExponentialSingularity| *Domains*)
    (make-instance '|UnivariatePuiseuxSeriesWithExponentialSingularityType|)))
```

———-

1.56.8 UnivariateSkewPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |UnivariateSkewPolynomial|
  (progn
    (push '|UnivariateSkewPolynomial| *Domains*)
    (make-instance '|UnivariateSkewPolynomialType|)))
```

———-

1.56.9 UnivariateTaylorSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeries|
  (progn
    (push '|UnivariateTaylorSeries| *Domains*)
    (make-instance '|UnivariateTaylorSeriesType|)))
```

———-

1.56.10 UnivariateTaylorSeriesCZero

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesCZero|
  (progn
    (push '|UnivariateTaylorSeriesCZero| *Domains*)
    (make-instance '|UnivariateTaylorSeriesCZeroType|)))
```

———-

1.56.11 UniversalSegment

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UniversalSegment|
  (progn
    (push '|UniversalSegment| *Domains*)
    (make-instance '|UniversalSegmentType|)))
```

———-

1.56.12 U8Matrix

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U8Matrix|
  (progn
    (push '|U8Matrix| *Domains*)
    (make-instance '|U8MatrixType|)))
```

———-

1.56.13 U16Matrix

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U16Matrix|
  (progn
    (push '|U16Matrix| *Domains*)
    (make-instance '|U16MatrixType|)))
```

———-

1.56.14 U32Matrix

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U32Matrix|
  (progn
    (push '|U32Matrix| *Domains*)
    (make-instance '|U32MatrixType|)))
```

———-

1.56.15 U8Vector

— **sane** —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U8Vector|
  (progn
    (push '|U8Vector| *Domains*)
    (make-instance '|U8VectorType|)))
```

———-

1.56.16 U16Vector

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U16Vector|
  (progn
    (push '|U16Vector| *Domains*)
    (make-instance '|U16VectorType|)))
```

———-

1.56.17 U32Vector

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U32Vector|
  (progn
    (push '|U32Vector| *Domains*)
    (make-instance '|U32VectorType|)))
```

———-

1.57 V

1.57.1 Variable

— sane —

```
(defclass |VariableType| (|SetCategoryType|)
  ((parents :initform '(|SetCategory|))
   (name :initform "Variable")
   (marker :initform 'domain)
   (abbreviation :initform 'VARIABLE)
   (comment :initform (list
     "This domain implements variables"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Variable|
  (progn
    (push '|Variable| *Domains*)
    (make-instance '|VariableType|)))
```

———-

1.57.2 Vector

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Vector|
  (progn
    (push '|Vector| *Domains*)
    (make-instance '|VectorType|)))
```

———-

1.57.3 Void

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Void|
  (progn
    (push '|Void| *Domains*)
    (make-instance '|VoidType|)))
```

———-

1.58 W

1.58.1 WeightedPolynomials

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |WeightedPolynomials|
  (progn
    (push '|WeightedPolynomials| *Domains*)
    (make-instance '|WeightedPolynomialsType|)))
```

———-

1.58.2 WuWenTsunTriangularSet

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |WuWenTsunTriangularSet|
  (progn
    (push '|WuWenTsunTriangularSet| *Domains*)
    (make-instance '|WuWenTsunTriangularSetType|)))
```

———-

1.59 X

1.59.1 XDistributedPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XDistributedPolynomial|
  (progn
    (push '|XDistributedPolynomial| *Domains*)
    (make-instance '|XDistributedPolynomialType|)))
```

———-

1.59.2 XPBWPolynomial

— sane —

```
(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. "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPBWPolynomial|
  (progn
    (push '|XPBWPolynomial| *Domains*)
    (make-instance '|XPBWPolynomialType|)))
```

———-

1.59.3 XPolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPolynomial|
  (progn
    (push '|XPolynomial| *Domains*)
    (make-instance '|XPolynomialType|)))
```

———-

1.59.4 XPolynomialRing

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XPolynomialRing|
  (progn
    (push '|XPolynomialRing| *Domains*)
    (make-instance '|XPolynomialRingType|)))
```

———-

1.59.5 XRecursivePolynomial

— sane —

```
(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."))
   (arglist :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

1.60 A

1.60.1 AffineAlgebraicSetComputeWithGroebnerBasis

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AffineAlgebraicSetComputeWithGroebnerBasis|
  (progn
    (push '|AffineAlgebraicSetComputeWithGroebnerBasis| *Packages*)
    (make-instance '|AffineAlgebraicSetComputeWithGroebnerBasisType|)))
```

———-

1.60.2 AffineAlgebraicSetComputeWithResultant

— sane —

```
(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"))
   (arglist :initform nil)
```

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

 (defvar |AffineAlgebraicSetComputeWithResultant|
   (progn
     (push '|AffineAlgebraicSetComputeWithResultant| *Packages*)
     (make-instance '|AffineAlgebraicSetComputeWithResultantType|)))
```

———-

1.60.3 AlgebraicFunction

— sane —

```
 (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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicFunction|
  (progn
    (push '|AlgebraicFunction| *Packages*)
    (make-instance '|AlgebraicFunctionType|)))
```

———-

1.60.4 AlgebraicHermiteIntegration

— sane —

```
(defclass |AlgebraicHermiteIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "AlgebraicHermiteIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTHERAL)
   (comment :initform (list
     "Algebraic Hermite reduction."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |AlgebraicHermiteIntegration|
  (progn
    (push '|AlgebraicHermiteIntegration| *Packages*)
    (make-instance '|AlgebraicHermiteIntegrationType|)))
```

———-

1.60.5 AlgebraicIntegrate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicIntegrate|
  (progn
    (push '|AlgebraicIntegrate| *Packages*)
    (make-instance '|AlgebraicIntegrateType|)))
```

———-

1.60.6 AlgebraicIntegration

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |AlgebraicIntegration|
  (progn
    (push '|AlgebraicIntegration| *Packages*)
    (make-instance '|AlgebraicIntegrationType|)))
```

———-

1.60.7 AlgebraicManipulations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraicManipulations|
  (progn
    (push '|AlgebraicManipulations| *Packages*)
    (make-instance '|AlgebraicManipulationsType|)))
```

———-

1.60.8 AlgebraicMultFact

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |AlgebraicMultFact|
  (progn
    (push '|AlgebraicMultFact| *Packages*)
    (make-instance '|AlgebraicMultFactType|)))
```

———-

1.60.9 AlgebraPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AlgebraPackage|
  (progn
    (push '|AlgebraPackage| *Packages*)
    (make-instance '|AlgebraPackageType|)))
```

———-

1.60.10 AlgFactor

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |AlgFactor|
  (progn
    (push '|AlgFactor| *Packages*)
    (make-instance '|AlgFactorType|)))
```

———-

1.60.11 AnnaNumericalIntegrationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaNumericalIntegrationPackage|
  (progn
    (push '|AnnaNumericalIntegrationPackage| *Packages*)
    (make-instance '|AnnaNumericalIntegrationPackageType|)))
```

———-

1.60.12 AnnaNumericalOptimizationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaNumericalOptimizationPackage|
  (progn
    (push '|AnnaNumericalOptimizationPackage| *Packages*)
    (make-instance '|AnnaNumericalOptimizationPackageType|)))
```

———-

1.60.13 AnnaOrdinaryDifferentialEquationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaOrdinaryDifferentialEquationPackage|
  (progn
    (push '|AnnaOrdinaryDifferentialEquationPackage| *Packages*)
    (make-instance '|AnnaOrdinaryDifferentialEquationPackageType|)))
```

———-

1.60.14 AnnaPartialDifferentialEquationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnnaPartialDifferentialEquationPackage|
  (progn
    (push '|AnnaPartialDifferentialEquationPackage| *Packages*)
    (make-instance '|AnnaPartialDifferentialEquationPackageType|)))
```

———-

1.60.15 AnyFunctions1

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AnyFunctions1|
  (progn
    (push '|AnyFunctions1| *Packages*)
    (make-instance '|AnyFunctions1Type|)))
```

———-

1.60.16 ApplicationProgramInterface

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ApplicationProgramInterface|
  (progn
    (push '|ApplicationProgramInterface| *Packages*)
    (make-instance '|ApplicationProgramInterfaceType|)))
```

———-

1.60.17 ApplyRules

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ApplyRules|
  (progn
    (push '|ApplyRules| *Packages*)
    (make-instance '|ApplyRulesType|)))
```

———-

1.60.18 ApplyUnivariateSkewPolynomial

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ApplyUnivariateSkewPolynomial|
  (progn
    (push '|ApplyUnivariateSkewPolynomial| *Packages*)
    (make-instance '|ApplyUnivariateSkewPolynomialType|)))
```

———-

1.60.19 AssociatedEquations

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AssociatedEquations|
  (progn
    (push '|AssociatedEquations| *Packages*)
    (make-instance '|AssociatedEquationsType|)))
```

———-

1.60.20 AttachPredicates

— **sane** —

```
(defclass |AttachPredicatesType| (|AxiomClass|)
  ((parents :initform ())
```

```
   (name :initform "AttachPredicates")
   (marker :initform 'package)
   (abbreviation :initform 'PMPRED)
   (comment :initform (list
     "Attaching predicates to symbols for pattern matching."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AttachPredicates|
  (progn
    (push '|AttachPredicates| *Packages*)
    (make-instance '|AttachPredicatesType|)))
```

———-

1.60.21 AxiomServer

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |AxiomServer|
  (progn
    (push '|AxiomServer| *Packages*)
    (make-instance '|AxiomServerType|)))
```

———-

1.61 B

1.61.1 BalancedFactorisation

— sane —

```
(defclass |BalancedFactorisationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BalancedFactorisation")
   (marker :initform 'package)
   (abbreviation :initform 'BALFACT)
   (comment :initform (list
     "This package provides balanced factorisations of polynomials."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BalancedFactorisation|
  (progn
    (push '|BalancedFactorisation| *Packages*)
    (make-instance '|BalancedFactorisationType|)))
```

———-

1.61.2 BasicOperatorFunctions1

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BasicOperatorFunctions1|
  (progn
    (push '|BasicOperatorFunctions1| *Packages*)
    (make-instance '|BasicOperatorFunctions1Type|)))
```

———-

1.61.3 Bezier

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Bezier|
  (progn
    (push '|Bezier| *Packages*)
    (make-instance '|BezierType|)))
```

———-

1.61.4 BezoutMatrix

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BezoutMatrix|
  (progn
    (push '|BezoutMatrix| *Packages*)
    (make-instance '|BezoutMatrixType|)))
```

———-

1.61.5 BlowUpPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BlowUpPackage|
  (progn
    (push '|BlowUpPackage| *Packages*)
    (make-instance '|BlowUpPackageType|)))
```

———-

1.61.6 BoundIntegerRoots

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BoundIntegerRoots|
  (progn
    (push '|BoundIntegerRoots| *Packages*)
    (make-instance '|BoundIntegerRootsType|)))
```

———-

1.61.7 BrillhartTests

— **sane** —

```
(defclass |BrillhartTestsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "BrillhartTests")
```

```
   (marker :initform 'package)
   (abbreviation :initform 'BRILL)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |BrillhartTests|
  (progn
    (push '|BrillhartTests| *Packages*)
    (make-instance '|BrillhartTestsType|)))
```

———-

1.62 C

1.62.1 CartesianTensorFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CartesianTensorFunctions2|
  (progn
    (push '|CartesianTensorFunctions2| *Packages*)
    (make-instance '|CartesianTensorFunctions2Type|)))
```

———-

1.62.2 ChangeOfVariable

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ChangeOfVariable|
  (progn
    (push '|ChangeOfVariable| *Packages*)
    (make-instance '|ChangeOfVariableType|)))
```

———-

1.62.3 CharacteristicPolynomialInMonogenicalAlgebra

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicPolynomialInMonogenicalAlgebra|
  (progn
    (push '|CharacteristicPolynomialInMonogenicalAlgebra| *Packages*)
    (make-instance '|CharacteristicPolynomialInMonogenicalAlgebraType|)))
```

———-

1.62.4 CharacteristicPolynomialPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CharacteristicPolynomialPackage|
  (progn
    (push '|CharacteristicPolynomialPackage| *Packages*)
    (make-instance '|CharacteristicPolynomialPackageType|)))
```

———-

1.62.5 ChineseRemainderToolsForIntegralBases

— **sane** —

```
(defclass |ChineseRemainderToolsForIntegralBasesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ChineseRemainderToolsForIntegralBases")
   (marker :initform 'package)
   (abbreviation :initform 'IBACHIN)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ChineseRemainderToolsForIntegralBases|
  (progn
    (push '|ChineseRemainderToolsForIntegralBases| *Packages*)
    (make-instance '|ChineseRemainderToolsForIntegralBasesType|)))
```

———-

1.62.6 CoerceVectorMatrixPackage

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CoerceVectorMatrixPackage|
  (progn
    (push '|CoerceVectorMatrixPackage| *Packages*)
    (make-instance '|CoerceVectorMatrixPackageType|)))
```

———-

1.62.7 CombinatorialFunction

— **sane** —

```
(defclass |CombinatorialFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CombinatorialFunction")
   (marker :initform 'package)
   (abbreviation :initform 'COMBF)
   (comment :initform (list
     "Provides combinatorial functions over an integral domain."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CombinatorialFunction|
  (progn
    (push '|CombinatorialFunction| *Packages*)
    (make-instance '|CombinatorialFunctionType|)))
```

———-

1.62.8 CommonDenominator

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommonDenominator|
  (progn
    (push '|CommonDenominator| *Packages*)
    (make-instance '|CommonDenominatorType|)))
```

———-

1.62.9 CommonOperators

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommonOperators|
  (progn
    (push '|CommonOperators| *Packages*)
    (make-instance '|CommonOperatorsType|)))
```

———-

1.62.10 CommuteUnivariatePolynomialCategory

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CommuteUnivariatePolynomialCategory|
  (progn
    (push '|CommuteUnivariatePolynomialCategory| *Packages*)
    (make-instance '|CommuteUnivariatePolynomialCategoryType|)))
```

———-

1.62.11 ComplexFactorization

— **sane** —

```
(defclass |ComplexFactorizationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ComplexFactorization")
   (marker :initform 'package)
   (abbreviation :initform 'COMPFACT)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexFactorization|
  (progn
    (push '|ComplexFactorization| *Packages*)
    (make-instance '|ComplexFactorizationType|)))
```

———-

1.62.12 ComplexFunctions2

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexFunctions2|
  (progn
    (push '|ComplexFunctions2| *Packages*)
    (make-instance '|ComplexFunctions2Type|)))
```

———-

1.62.13 ComplexIntegerSolveLinearPolynomialEquation

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexIntegerSolveLinearPolynomialEquation|
  (progn
    (push '|ComplexIntegerSolveLinearPolynomialEquation| *Packages*)
    (make-instance '|ComplexIntegerSolveLinearPolynomialEquationType|)))
```

———-

1.62.14 ComplexPattern

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexPattern|
  (progn
    (push '|ComplexPattern| *Packages*)
    (make-instance '|ComplexPatternType|)))
```

———-

1.62.15 ComplexPatternMatch

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexPatternMatch|
  (progn
    (push '|ComplexPatternMatch| *Packages*)
    (make-instance '|ComplexPatternMatchType|)))
```

———-

1.62.16 ComplexRootFindingPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexRootFindingPackage|
  (progn
    (push '|ComplexRootFindingPackage| *Packages*)
    (make-instance '|ComplexRootFindingPackageType|)))
```

———-

1.62.17 ComplexRootPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexRootPackage|
  (progn
```

```
    (push '|ComplexRootPackage| *Packages*)
    (make-instance '|ComplexRootPackageType|)))
```

———-

1.62.18 ComplexTrigonometricManipulations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ComplexTrigonometricManipulations|
  (progn
    (push '|ComplexTrigonometricManipulations| *Packages*)
    (make-instance '|ComplexTrigonometricManipulationsType|)))
```

———-

1.62.19 ConstantLODE

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ConstantLODE|
  (progn
```

```
    (push '|ConstantLODE| *Packages*)
    (make-instance '|ConstantLODEType|)))
```

———-

1.62.20 CoordinateSystems

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CoordinateSystems|
  (progn
    (push '|CoordinateSystems| *Packages*)
    (make-instance '|CoordinateSystemsType|)))
```

———-

1.62.21 CRApackage

— sane —

```
(defclass |CRApackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CRApackage")
   (marker :initform 'package)
   (abbreviation :initform 'CRAPACK)
   (comment :initform (list
     "This package has no documentation"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CRApackage|
```

```
  (progn
    (push '|CRApackage| *Packages*)
    (make-instance '|CRApackageType|)))
```

———-

1.62.22 CycleIndicators

— sane —

```
(defclass |CycleIndicatorsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CycleIndicators")
   (marker :initform 'package)
   (abbreviation :initform 'CYCLES)
   (comment :initform (list
     "Polya-Redfield enumeration by cycle indices."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CycleIndicators|
  (progn
    (push '|CycleIndicators| *Packages*)
    (make-instance '|CycleIndicatorsType|)))
```

———-

1.62.23 CyclicStreamTools

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CyclicStreamTools|
  (progn
    (push '|CyclicStreamTools| *Packages*)
```

```
    (make-instance '|CyclicStreamToolsType|)))
```

———-

1.62.24 CyclotomicPolynomialPackage

— sane —

```
(defclass |CyclotomicPolynomialPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CyclotomicPolynomialPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CYCLOTOM)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CyclotomicPolynomialPackage|
  (progn
    (push '|CyclotomicPolynomialPackage| *Packages*)
    (make-instance '|CyclotomicPolynomialPackageType|)))
```

———-

1.62.25 CylindricalAlgebraicDecompositionPackage

— sane —

```
(defclass |CylindricalAlgebraicDecompositionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CylindricalAlgebraicDecompositionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'CAD)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CylindricalAlgebraicDecompositionPackage|
  (progn
    (push '|CylindricalAlgebraicDecompositionPackage| *Packages*)
    (make-instance '|CylindricalAlgebraicDecompositionPackageType|)))
```

———-

1.62.26 CylindricalAlgebraicDecompositionUtilities

— sane —

```
(defclass |CylindricalAlgebraicDecompositionUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "CylindricalAlgebraicDecompositionUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'CADU)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |CylindricalAlgebraicDecompositionUtilities|
  (progn
    (push '|CylindricalAlgebraicDecompositionUtilities| *Packages*)
    (make-instance '|CylindricalAlgebraicDecompositionUtilitiesType|)))
```

———-

1.63 D

1.63.1 DefiniteIntegrationTools

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DefiniteIntegrationTools|
  (progn
    (push '|DefiniteIntegrationTools| *Packages*)
    (make-instance '|DefiniteIntegrationToolsType|)))
```

———-

1.63.2 DegreeReductionPackage

— sane —

```
(defclass |DegreeReductionPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DegreeReductionPackage")
   (marker :initform 'package)
   (abbreviation :initform 'DEGRED)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DegreeReductionPackage|
  (progn
    (push '|DegreeReductionPackage| *Packages*)
    (make-instance '|DegreeReductionPackageType|)))
```

———-

1.63.3 DesingTreePackage

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DesingTreePackage|
  (progn
    (push '|DesingTreePackage| *Packages*)
    (make-instance '|DesingTreePackageType|)))
```

———-

1.63.4 DiophantineSolutionPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DiophantineSolutionPackage|
  (progn
    (push '|DiophantineSolutionPackage| *Packages*)
    (make-instance '|DiophantineSolutionPackageType|)))
```

———-

1.63.5 DirectProductFunctions2

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DirectProductFunctions2|
  (progn
    (push '|DirectProductFunctions2| *Packages*)
    (make-instance '|DirectProductFunctions2Type|)))
```

———-

1.63.6 DiscreteLogarithmPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DiscreteLogarithmPackage|
  (progn
    (push '|DiscreteLogarithmPackage| *Packages*)
    (make-instance '|DiscreteLogarithmPackageType|)))
```

———-

1.63.7 DisplayPackage

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DisplayPackage|
  (progn
    (push '|DisplayPackage| *Packages*)
    (make-instance '|DisplayPackageType|)))
```

———-

1.63.8 DistinctDegreeFactorize

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DistinctDegreeFactorize|
  (progn
    (push '|DistinctDegreeFactorize| *Packages*)
    (make-instance '|DistinctDegreeFactorizeType|)))
```

———-

1.63.9 DoubleFloatSpecialFunctions

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleFloatSpecialFunctions|
  (progn
    (push '|DoubleFloatSpecialFunctions| *Packages*)
    (make-instance '|DoubleFloatSpecialFunctionsType|)))
```

———-

1.63.10 DoubleResultantPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DoubleResultantPackage|
  (progn
    (push '|DoubleResultantPackage| *Packages*)
    (make-instance '|DoubleResultantPackageType|)))
```

———-

1.63.11 DrawComplex

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawComplex|
  (progn
    (push '|DrawComplex| *Packages*)
    (make-instance '|DrawComplexType|)))
```

———-

1.63.12 DrawNumericHack

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawNumericHack|
  (progn
    (push '|DrawNumericHack| *Packages*)
    (make-instance '|DrawNumericHackType|)))
```

———-

1.63.13 DrawOptionFunctions0

— **sane** —

```
(defclass |DrawOptionFunctions0Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DrawOptionFunctions0")
   (marker :initform 'package)
   (abbreviation :initform 'DROPT0)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawOptionFunctions0|
  (progn
    (push '|DrawOptionFunctions0| *Packages*)
    (make-instance '|DrawOptionFunctions0Type|)))
```

———-

1.63.14 DrawOptionFunctions1

— sane —

```
(defclass |DrawOptionFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "DrawOptionFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'DROPT1)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |DrawOptionFunctions1|
  (progn
    (push '|DrawOptionFunctions1| *Packages*)
    (make-instance '|DrawOptionFunctions1Type|)))
```

———-

1.63.15 d01AgentsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01AgentsPackage|
  (progn
    (push '|d01AgentsPackage| *Packages*)
    (make-instance '|d01AgentsPackageType|)))
```

———-

1.63.16 d01WeightsPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d01WeightsPackage|
  (progn
    (push '|d01WeightsPackage| *Packages*)
    (make-instance '|d01WeightsPackageType|)))
```

———-

1.63.17 d02AgentsPackage

— sane —

```
(defclass |d02AgentsPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "d02AgentsPackage")
   (marker :initform 'package)
   (abbreviation :initform 'D02AGNT)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d02AgentsPackage|
  (progn
    (push '|d02AgentsPackage| *Packages*)
    (make-instance '|d02AgentsPackageType|)))
```

———-

1.63.18 d03AgentsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |d03AgentsPackage|
  (progn
    (push '|d03AgentsPackage| *Packages*)
    (make-instance '|d03AgentsPackageType|)))
```

———-

1.64 E

1.64.1 EigenPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EigenPackage|
  (progn
    (push '|EigenPackage| *Packages*)
    (make-instance '|EigenPackageType|)))
```

———-

1.64.2 ElementaryFunction

— sane —

```
(defclass |ElementaryFunctionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryFunction")
   (marker :initform 'package)
   (abbreviation :initform 'EF)
   (comment :initform (list
     "Provides elementary functions over an integral domain."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunction|
  (progn
    (push '|ElementaryFunction| *Packages*)
    (make-instance '|ElementaryFunctionType|)))
```

———-

1.64.3 ElementaryFunctionDefiniteIntegration

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionDefiniteIntegration|
  (progn
    (push '|ElementaryFunctionDefiniteIntegration| *Packages*)
    (make-instance '|ElementaryFunctionDefiniteIntegrationType|)))
```

———-

1.64.4 ElementaryFunctionLODESolver

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionLODESolver|
  (progn
    (push '|ElementaryFunctionLODESolver| *Packages*)
```

```
    (make-instance '|ElementaryFunctionLODESolverType|)))
```

———-

1.64.5 ElementaryFunctionODESolver

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionODESolver|
  (progn
    (push '|ElementaryFunctionODESolver| *Packages*)
    (make-instance '|ElementaryFunctionODESolverType|)))
```

———-

1.64.6 ElementaryFunctionSign

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionSign|
  (progn
```

```
    (push '|ElementaryFunctionSign| *Packages*)
    (make-instance '|ElementaryFunctionSignType|)))
```

———-

1.64.7 ElementaryFunctionStructurePackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryFunctionStructurePackage|
  (progn
    (push '|ElementaryFunctionStructurePackage| *Packages*)
    (make-instance '|ElementaryFunctionStructurePackageType|)))
```

———-

1.64.8 ElementaryIntegration

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |ElementaryIntegration|
  (progn
    (push '|ElementaryIntegration| *Packages*)
    (make-instance '|ElementaryIntegrationType|)))
```

———-

1.64.9 ElementaryRischDE

— sane —

```
(defclass |ElementaryRischDEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryRischDE")
   (marker :initform 'package)
   (abbreviation :initform 'RDEEF)
   (comment :initform (list
     "Risch differential equation, elementary case."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ElementaryRischDE|
  (progn
    (push '|ElementaryRischDE| *Packages*)
    (make-instance '|ElementaryRischDEType|)))
```

———-

1.64.10 ElementaryRischDESystem

— sane —

```
(defclass |ElementaryRischDESystemType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ElementaryRischDESystem")
   (marker :initform 'package)
   (abbreviation :initform 'RDEEFS)
   (comment :initform (list
     "Risch differential equation, elementary case."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |ElementaryRischDESystem|
  (progn
    (push '|ElementaryRischDESystem| *Packages*)
    (make-instance '|ElementaryRischDESystemType|)))
```

———-

1.64.11 EllipticFunctionsUnivariateTaylorSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EllipticFunctionsUnivariateTaylorSeries|
  (progn
    (push '|EllipticFunctionsUnivariateTaylorSeries| *Packages*)
    (make-instance '|EllipticFunctionsUnivariateTaylorSeriesType|)))
```

———-

1.64.12 EquationFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EquationFunctions2|
  (progn
```

```
    (push '|EquationFunctions2| *Packages*)
    (make-instance '|EquationFunctions2Type|)))
```

———-

1.64.13 ErrorFunctions

— sane —

```
(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!']"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |ErrorFunctions|
  (progn
    (push '|ErrorFunctions| *Packages*)
    (make-instance '|ErrorFunctionsType|)))
```

———-

1.64.14 EuclideanGroebnerBasisPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EuclideanGroebnerBasisPackage|
  (progn
    (push '|EuclideanGroebnerBasisPackage| *Packages*)
    (make-instance '|EuclideanGroebnerBasisPackageType|)))
```

———-

1.64.15 EvaluateCycleIndicators

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |EvaluateCycleIndicators|
  (progn
    (push '|EvaluateCycleIndicators| *Packages*)
    (make-instance '|EvaluateCycleIndicatorsType|)))
```

———-

1.64.16 ExpertSystemContinuityPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemContinuityPackage|
  (progn
    (push '|ExpertSystemContinuityPackage| *Packages*)
    (make-instance '|ExpertSystemContinuityPackageType|)))
```

———-

1.64.17 ExpertSystemContinuityPackage1

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemContinuityPackage1|
  (progn
    (push '|ExpertSystemContinuityPackage1| *Packages*)
    (make-instance '|ExpertSystemContinuityPackage1Type|)))
```

———-

1.64.18 ExpertSystemToolsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemToolsPackage|
  (progn
    (push '|ExpertSystemToolsPackage| *Packages*)
    (make-instance '|ExpertSystemToolsPackageType|)))
```

———-

1.64.19 ExpertSystemToolsPackage1

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemToolsPackage1|
  (progn
    (push '|ExpertSystemToolsPackage1| *Packages*)
    (make-instance '|ExpertSystemToolsPackage1Type|)))
```

———-

1.64.20 ExpertSystemToolsPackage2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpertSystemToolsPackage2|
  (progn
    (push '|ExpertSystemToolsPackage2| *Packages*)
    (make-instance '|ExpertSystemToolsPackage2Type|)))
```

———-

1.64.21 ExpressionFunctions2

— sane —

```
(defclass |ExpressionFunctions2Type| (|AxiomClass|)
```

```
  ((parents :initform ())
   (name :initform "ExpressionFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'EXPR2)
   (comment :initform (list
     "Lifting of maps to Expressions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionFunctions2|
  (progn
    (push '|ExpressionFunctions2| *Packages*)
    (make-instance '|ExpressionFunctions2Type|)))
```

———-

1.64.22 ExpressionSolve

— **sane** —

```
(defclass |ExpressionSolveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionSolve")
   (marker :initform 'package)
   (abbreviation :initform 'EXPRSOL)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSolve|
  (progn
    (push '|ExpressionSolve| *Packages*)
    (make-instance '|ExpressionSolveType|)))
```

———-

1.64.23 ExpressionSpaceFunctions1

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpaceFunctions1|
  (progn
    (push '|ExpressionSpaceFunctions1| *Packages*)
    (make-instance '|ExpressionSpaceFunctions1Type|)))
```

———-

1.64.24 ExpressionSpaceFunctions2

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpaceFunctions2|
  (progn
    (push '|ExpressionSpaceFunctions2| *Packages*)
    (make-instance '|ExpressionSpaceFunctions2Type|)))
```

———-

1.64.25 ExpressionSpaceODESolver

— **sane** —

```
(defclass |ExpressionSpaceODESolverType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ExpressionSpaceODESolver")
   (marker :initform 'package)
   (abbreviation :initform 'EXPRODE)
   (comment :initform (list
     "Taylor series solutions of explicit ODE's"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionSpaceODESolver|
  (progn
    (push '|ExpressionSpaceODESolver| *Packages*)
    (make-instance '|ExpressionSpaceODESolverType|)))
```

———-

1.64.26 ExpressionToOpenMath

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionToOpenMath|
  (progn
    (push '|ExpressionToOpenMath| *Packages*)
    (make-instance '|ExpressionToOpenMathType|)))
```

———-

1.64.27 ExpressionToUnivariatePowerSeries

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionToUnivariatePowerSeries|
  (progn
    (push '|ExpressionToUnivariatePowerSeries| *Packages*)
    (make-instance '|ExpressionToUnivariatePowerSeriesType|)))
```

———-

1.64.28 ExpressionTubePlot

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ExpressionTubePlot|
  (progn
    (push '|ExpressionTubePlot| *Packages*)
    (make-instance '|ExpressionTubePlotType|)))
```

———-

1.64.29 Export3D

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Export3D|
  (progn
    (push '|Export3D| *Packages*)
    (make-instance '|Export3DType|)))
```

———-

1.64.30 e04AgentsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |e04AgentsPackage|
  (progn
    (push '|e04AgentsPackage| *Packages*)
    (make-instance '|e04AgentsPackageType|)))
```

———-

1.65 F

1.65.1 FactoredFunctions

— sane —

```
(defclass |FactoredFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FactoredFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'FACTFUNC)
   (comment :initform (list
     "Computes various functions on factored arguments."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoredFunctions|
  (progn
    (push '|FactoredFunctions| *Packages*)
    (make-instance '|FactoredFunctionsType|)))
```

———-

1.65.2 FactoredFunctions2

— sane —

```
(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))."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoredFunctions2|
  (progn
    (push '|FactoredFunctions2| *Packages*)
    (make-instance '|FactoredFunctions2Type|)))
```

———-

1.65.3 FactoredFunctionUtilities

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoredFunctionUtilities|
  (progn
    (push '|FactoredFunctionUtilities| *Packages*)
    (make-instance '|FactoredFunctionUtilitiesType|)))
```

———-

1.65.4 FactoringUtilities

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactoringUtilities|
  (progn
    (push '|FactoringUtilities| *Packages*)
    (make-instance '|FactoringUtilitiesType|)))
```

———-

1.65.5 FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber|
  (progn
    (push '|FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber| *Packages*)
    (make-instance '|FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumberType|)))
```

———-

1.65.6 FactorisationOverPseudoAlgebraicClosureOfRationalNumber

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FactorisationOverPseudoAlgebraicClosureOfRationalNumber|
  (progn
    (push '|FactorisationOverPseudoAlgebraicClosureOfRationalNumber| *Packages*)
    (make-instance '|FactorisationOverPseudoAlgebraicClosureOfRationalNumberType|)))
```

———-

1.65.7 FGLMIfCanPackage

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FGLMIfCanPackage|
  (progn
    (push '|FGLMIfCanPackage| *Packages*)
    (make-instance '|FGLMIfCanPackageType|)))
```

———-

1.65.8 FindOrderFinite

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FindOrderFinite|
  (progn
    (push '|FindOrderFinite| *Packages*)
    (make-instance '|FindOrderFiniteType|)))
```

———-

1.65.9 FiniteAbelianMonoidRingFunctions2

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteAbelianMonoidRingFunctions2|
  (progn
    (push '|FiniteAbelianMonoidRingFunctions2| *Packages*)
    (make-instance '|FiniteAbelianMonoidRingFunctions2Type|)))
```

———-

1.65.10 FiniteDivisorFunctions2

— sane —

```
(defclass |FiniteDivisorFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteDivisorFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FDIV2)
   (comment :initform (list
     "Lift a map to finite divisors."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteDivisorFunctions2|
  (progn
    (push '|FiniteDivisorFunctions2| *Packages*)
    (make-instance '|FiniteDivisorFunctions2Type|)))
```

———-

1.65.11 FiniteFieldFactorization

— sane —

```
(defclass |FiniteFieldFactorizationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FiniteFieldFactorization")
   (marker :initform 'package)
   (abbreviation :initform 'FFFACTOR)
   (comment :initform (list
     "Part of the PAFF package"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldFactorization|
  (progn
    (push '|FiniteFieldFactorization| *Packages*)
    (make-instance '|FiniteFieldFactorizationType|)))
```

———-

1.65.12 FiniteFieldFactorizationWithSizeParseBySideEffect

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldFactorizationWithSizeParseBySideEffect|
  (progn
    (push '|FiniteFieldFactorizationWithSizeParseBySideEffect| *Packages*)
```

```
    (make-instance '|FiniteFieldFactorizationWithSizeParseBySideEffectType|)))
```

———-

1.65.13 FiniteFieldFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldFunctions|
  (progn
    (push '|FiniteFieldFunctions| *Packages*)
    (make-instance '|FiniteFieldFunctionsType|)))
```

———-

1.65.14 FiniteFieldHomomorphisms

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldHomomorphisms|
```

```
  (progn
    (push '|FiniteFieldHomomorphisms| *Packages*)
    (make-instance '|FiniteFieldHomomorphismsType|)))
```

———-

1.65.15 FiniteFieldPolynomialPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldPolynomialPackage|
  (progn
    (push '|FiniteFieldPolynomialPackage| *Packages*)
    (make-instance '|FiniteFieldPolynomialPackageType|)))
```

———-

1.65.16 FiniteFieldPolynomialPackage2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |FiniteFieldPolynomialPackage2|
  (progn
    (push '|FiniteFieldPolynomialPackage2| *Packages*)
    (make-instance '|FiniteFieldPolynomialPackage2Type|)))
```

———-

1.65.17 FiniteFieldSolveLinearPolynomialEquation

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteFieldSolveLinearPolynomialEquation|
  (progn
    (push '|FiniteFieldSolveLinearPolynomialEquation| *Packages*)
    (make-instance '|FiniteFieldSolveLinearPolynomialEquationType|)))
```

———-

1.65.18 FiniteFieldSquareFreeDecomposition

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |FiniteFieldSquareFreeDecomposition|
  (progn
    (push '|FiniteFieldSquareFreeDecomposition| *Packages*)
    (make-instance '|FiniteFieldSquareFreeDecompositionType|)))
```

———-

1.65.19 FiniteLinearAggregateFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteLinearAggregateFunctions2|
  (progn
    (push '|FiniteLinearAggregateFunctions2| *Packages*)
    (make-instance '|FiniteLinearAggregateFunctions2Type|)))
```

———-

1.65.20 FiniteLinearAggregateSort

— sane —

```
(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"))
   (arglist :initform nil)
```

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

(defvar |FiniteLinearAggregateSort|
  (progn
    (push '|FiniteLinearAggregateSort| *Packages*)
    (make-instance '|FiniteLinearAggregateSortType|)))
```

———-

1.65.21 FiniteSetAggregateFunctions2

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FiniteSetAggregateFunctions2|
  (progn
    (push '|FiniteSetAggregateFunctions2| *Packages*)
    (make-instance '|FiniteSetAggregateFunctions2Type|)))
```

———-

1.65.22 FloatingComplexPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatingComplexPackage|
  (progn
    (push '|FloatingComplexPackage| *Packages*)
    (make-instance '|FloatingComplexPackageType|)))
```

———-

1.65.23 FloatingRealPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatingRealPackage|
  (progn
    (push '|FloatingRealPackage| *Packages*)
    (make-instance '|FloatingRealPackageType|)))
```

———-

1.65.24 FloatSpecialFunctions

— **sane** —

```
(defclass |FloatSpecialFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FloatSpecialFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'FSFUN)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FloatSpecialFunctions|
  (progn
    (push '|FloatSpecialFunctions| *Packages*)
    (make-instance '|FloatSpecialFunctionsType|)))
```

———-

1.65.25 FortranCodePackage1

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranCodePackage1|
  (progn
    (push '|FortranCodePackage1| *Packages*)
    (make-instance '|FortranCodePackage1Type|)))
```

———-

1.65.26 FortranOutputStackPackage

— sane —

```
(defclass |FortranOutputStackPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FortranOutputStackPackage")
   (marker :initform 'package)
   (abbreviation :initform 'FOP)
   (comment :initform (list
     "Code to manipulate Fortran Output Stack"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranOutputStackPackage|
  (progn
    (push '|FortranOutputStackPackage| *Packages*)
    (make-instance '|FortranOutputStackPackageType|)))
```

———-

1.65.27 FortranPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FortranPackage|
  (progn
    (push '|FortranPackage| *Packages*)
    (make-instance '|FortranPackageType|)))
```

———-

1.65.28 FractionalIdealFunctions2

— sane —

```
(defclass |FractionalIdealFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FractionalIdealFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'FRIDEAL2)
   (comment :initform (list
     "Lifting of morphisms to fractional ideals."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionalIdealFunctions2|
  (progn
    (push '|FractionalIdealFunctions2| *Packages*)
    (make-instance '|FractionalIdealFunctions2Type|)))
```

———-

1.65.29 FractionFreeFastGaussian

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionFreeFastGaussian|
  (progn
    (push '|FractionFreeFastGaussian| *Packages*)
    (make-instance '|FractionFreeFastGaussianType|)))
```

———-

1.65.30 FractionFreeFastGaussianFractions

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionFreeFastGaussianFractions|
  (progn
    (push '|FractionFreeFastGaussianFractions| *Packages*)
    (make-instance '|FractionFreeFastGaussianFractionsType|)))
```

———-

1.65.31 FractionFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FractionFunctions2|
  (progn
    (push '|FractionFunctions2| *Packages*)
```

```
    (make-instance '|FractionFunctions2Type|)))
```

———-

1.65.32 FramedNonAssociativeAlgebraFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FramedNonAssociativeAlgebraFunctions2|
  (progn
    (push '|FramedNonAssociativeAlgebraFunctions2| *Packages*)
    (make-instance '|FramedNonAssociativeAlgebraFunctions2Type|)))
```

———-

1.65.33 FunctionalSpecialFunction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionalSpecialFunction|
  (progn
```

```
    (push '|FunctionalSpecialFunction| *Packages*)
    (make-instance '|FunctionalSpecialFunctionType|)))
```

———-

1.65.34 FunctionFieldCategoryFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionFieldCategoryFunctions2|
  (progn
    (push '|FunctionFieldCategoryFunctions2| *Packages*)
    (make-instance '|FunctionFieldCategoryFunctions2Type|)))
```

———-

1.65.35 FunctionFieldIntegralBasis

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |FunctionFieldIntegralBasis|
  (progn
    (push '|FunctionFieldIntegralBasis| *Packages*)
    (make-instance '|FunctionFieldIntegralBasisType|)))
```

———-

1.65.36 FunctionSpaceAssertions

— sane —

```
(defclass |FunctionSpaceAssertionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceAssertions")
   (marker :initform 'package)
   (abbreviation :initform 'PMASSFS)
   (comment :initform (list
     "Attaching assertions to symbols for pattern matching"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceAssertions|
  (progn
    (push '|FunctionSpaceAssertions| *Packages*)
    (make-instance '|FunctionSpaceAssertionsType|)))
```

———-

1.65.37 FunctionSpaceAttachPredicates

— sane —

```
(defclass |FunctionSpaceAttachPredicatesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceAttachPredicates")
   (marker :initform 'package)
   (abbreviation :initform 'PMPREDFS)
   (comment :initform (list
     "Attaching predicates to symbols for pattern matching."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceAttachPredicates|
```

```
  (progn
    (push '|FunctionSpaceAttachPredicates| *Packages*)
    (make-instance '|FunctionSpaceAttachPredicatesType|)))
```

———-

1.65.38 FunctionSpaceComplexIntegration

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceComplexIntegration|
  (progn
    (push '|FunctionSpaceComplexIntegration| *Packages*)
    (make-instance '|FunctionSpaceComplexIntegrationType|)))
```

———-

1.65.39 FunctionSpaceFunctions2

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceFunctions2|
```

```
  (progn
    (push '|FunctionSpaceFunctions2| *Packages*)
    (make-instance '|FunctionSpaceFunctions2Type|)))
```

———-

1.65.40 FunctionSpaceIntegration

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceIntegration|
  (progn
    (push '|FunctionSpaceIntegration| *Packages*)
    (make-instance '|FunctionSpaceIntegrationType|)))
```

———-

1.65.41 FunctionSpacePrimitiveElement

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |FunctionSpacePrimitiveElement|
  (progn
    (push '|FunctionSpacePrimitiveElement| *Packages*)
    (make-instance '|FunctionSpacePrimitiveElementType|)))
```

———-

1.65.42 FunctionSpaceReduce

— sane —

```
(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()."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceReduce|
  (progn
    (push '|FunctionSpaceReduce| *Packages*)
    (make-instance '|FunctionSpaceReduceType|)))
```

———-

1.65.43 FunctionSpaceSum

— sane —

```
(defclass |FunctionSpaceSumType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceSum")
   (marker :initform 'package)
   (abbreviation :initform 'SUMFS)
   (comment :initform (list
     "Computes sums of top-level expressions"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |FunctionSpaceSum|
  (progn
    (push '|FunctionSpaceSum| *Packages*)
    (make-instance '|FunctionSpaceSumType|)))
```

———-

1.65.44 FunctionSpaceToExponentialExpansion

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceToExponentialExpansion|
  (progn
    (push '|FunctionSpaceToExponentialExpansion| *Packages*)
    (make-instance '|FunctionSpaceToExponentialExpansionType|)))
```

———-

1.65.45 FunctionSpaceToUnivariatePowerSeries

— sane —

```
(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)."))
   (arglist :initform nil)
```

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

(defvar |FunctionSpaceToUnivariatePowerSeries|
  (progn
    (push '|FunctionSpaceToUnivariatePowerSeries| *Packages*)
    (make-instance '|FunctionSpaceToUnivariatePowerSeriesType|)))
```

———-

1.65.46 FunctionSpaceUnivariatePolynomialFactor

— sane —

```
(defclass |FunctionSpaceUnivariatePolynomialFactorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "FunctionSpaceUnivariatePolynomialFactor")
   (marker :initform 'package)
   (abbreviation :initform 'FSUPFACT)
   (comment :initform (list
     "This package is used internally by IR2F"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |FunctionSpaceUnivariatePolynomialFactor|
  (progn
    (push '|FunctionSpaceUnivariatePolynomialFactor| *Packages*)
    (make-instance '|FunctionSpaceUnivariatePolynomialFactorType|)))
```

———-

1.66 G

1.66.1 GaloisGroupFactorizationUtilities

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupFactorizationUtilities|
  (progn
    (push '|GaloisGroupFactorizationUtilities| *Packages*)
    (make-instance '|GaloisGroupFactorizationUtilitiesType|)))
```

———-

1.66.2 GaloisGroupFactorizer

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupFactorizer|
  (progn
    (push '|GaloisGroupFactorizer| *Packages*)
    (make-instance '|GaloisGroupFactorizerType|)))
```

———-

1.66.3 GaloisGroupPolynomialUtilities

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupPolynomialUtilities|
  (progn
    (push '|GaloisGroupPolynomialUtilities| *Packages*)
    (make-instance '|GaloisGroupPolynomialUtilitiesType|)))
```

———-

1.66.4 GaloisGroupUtilities

— **sane** —

```
(defclass |GaloisGroupUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GaloisGroupUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'GALUTIL)
   (comment :initform (list
     "GaloisGroupUtilities provides several useful functions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaloisGroupUtilities|
  (progn
    (push '|GaloisGroupUtilities| *Packages*)
    (make-instance '|GaloisGroupUtilitiesType|)))
```

———-

1.66.5 GaussianFactorizationPackage

— **sane** —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GaussianFactorizationPackage|
  (progn
    (push '|GaussianFactorizationPackage| *Packages*)
    (make-instance '|GaussianFactorizationPackageType|)))
```

———-

1.66.6 GeneralHenselPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralHenselPackage|
  (progn
    (push '|GeneralHenselPackage| *Packages*)
    (make-instance '|GeneralHenselPackageType|)))
```

———-

1.66.7 GeneralizedMultivariateFactorize

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |GeneralizedMultivariateFactorize|
  (progn
    (push '|GeneralizedMultivariateFactorize| *Packages*)
    (make-instance '|GeneralizedMultivariateFactorizeType|)))
```

———-

1.66.8 GeneralPackageForAlgebraicFunctionField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GeneralPackageForAlgebraicFunctionField|
  (progn
    (push '|GeneralPackageForAlgebraicFunctionField| *Packages*)
    (make-instance '|GeneralPackageForAlgebraicFunctionFieldType|)))
```

———-

1.66.9 GeneralPolynomialGcdPackage

— sane —

```
(defclass |GeneralPolynomialGcdPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GeneralPolynomialGcdPackage")
   (marker :initform 'package)
   (abbreviation :initform 'GENPGCD)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |GeneralPolynomialGcdPackage|
  (progn
    (push '|GeneralPolynomialGcdPackage| *Packages*)
    (make-instance '|GeneralPolynomialGcdPackageType|)))
```

———-

1.66.10 GenerateUnivariatePowerSeries

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenerateUnivariatePowerSeries|
  (progn
    (push '|GenerateUnivariatePowerSeries| *Packages*)
    (make-instance '|GenerateUnivariatePowerSeriesType|)))
```

———-

1.66.11 GenExEuclid

— sane —

```
(defclass |GenExEuclidType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GenExEuclid")
   (marker :initform 'package)
   (abbreviation :initform 'GENEEZ)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |GenExEuclid|
  (progn
    (push '|GenExEuclid| *Packages*)
    (make-instance '|GenExEuclidType|)))
```

———-

1.66.12 GenUFactorize

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenUFactorize|
  (progn
    (push '|GenUFactorize| *Packages*)
    (make-instance '|GenUFactorizeType|)))
```

———-

1.66.13 GenusZeroIntegration

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GenusZeroIntegration|
  (progn
    (push '|GenusZeroIntegration| *Packages*)
    (make-instance '|GenusZeroIntegrationType|)))
```

———-

1.66.14 GnuDraw

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GnuDraw|
  (progn
    (push '|GnuDraw| *Packages*)
    (make-instance '|GnuDrawType|)))
```

———-

1.66.15 GosperSummationMethod

— sane —

```
(defclass |GosperSummationMethodType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "GosperSummationMethod")
```

```
   (marker :initform 'package)
   (abbreviation :initform 'GOSPER)
   (comment :initform (list
     "Gosper's summation algorithm."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GosperSummationMethod|
  (progn
    (push '|GosperSummationMethod| *Packages*)
    (make-instance '|GosperSummationMethodType|)))
```

———-

1.66.16 GraphicsDefaults

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GraphicsDefaults|
  (progn
    (push '|GraphicsDefaults| *Packages*)
    (make-instance '|GraphicsDefaultsType|)))
```

———-

1.66.17 Graphviz

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Graphviz|
  (progn
    (push '|Graphviz| *Packages*)
    (make-instance '|GraphvizType|)))
```

———-

1.66.18 GrayCode

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GrayCode|
  (progn
    (push '|GrayCode| *Packages*)
    (make-instance '|GrayCodeType|)))
```

———-

1.66.19 GroebnerFactorizationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerFactorizationPackage|
  (progn
    (push '|GroebnerFactorizationPackage| *Packages*)
    (make-instance '|GroebnerFactorizationPackageType|)))
```

———-

1.66.20 GroebnerInternalPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerInternalPackage|
  (progn
    (push '|GroebnerInternalPackage| *Packages*)
```

```
    (make-instance '|GroebnerInternalPackageType|)))
```

———-

1.66.21 GroebnerPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerPackage|
  (progn
    (push '|GroebnerPackage| *Packages*)
    (make-instance '|GroebnerPackageType|)))
```

———-

1.66.22 GroebnerSolve

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GroebnerSolve|
  (progn
    (push '|GroebnerSolve| *Packages*)
    (make-instance '|GroebnerSolveType|)))
```

———-

1.66.23 Guess

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Guess|
  (progn
    (push '|Guess| *Packages*)
    (make-instance '|GuessType|)))
```

———-

1.66.24 GuessAlgebraicNumber

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessAlgebraicNumber|
  (progn
    (push '|GuessAlgebraicNumber| *Packages*)
    (make-instance '|GuessAlgebraicNumberType|)))
```

———-

1.66.25 GuessFinite

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessFinite|
  (progn
    (push '|GuessFinite| *Packages*)
    (make-instance '|GuessFiniteType|)))
```

———-

1.66.26 GuessFiniteFunctions

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessFiniteFunctions|
  (progn
    (push '|GuessFiniteFunctions| *Packages*)
    (make-instance '|GuessFiniteFunctionsType|)))
```

———-

1.66.27 GuessInteger

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessInteger|
  (progn
    (push '|GuessInteger| *Packages*)
    (make-instance '|GuessIntegerType|)))
```

———-

1.66.28 GuessPolynomial

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessPolynomial|
  (progn
    (push '|GuessPolynomial| *Packages*)
    (make-instance '|GuessPolynomialType|)))
```

———-

1.66.29 GuessUnivariatePolynomial

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |GuessUnivariatePolynomial|
  (progn
    (push '|GuessUnivariatePolynomial| *Packages*)
    (make-instance '|GuessUnivariatePolynomialType|)))
```

———-

1.67 H

1.67.1 HallBasis

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HallBasis|
  (progn
    (push '|HallBasis| *Packages*)
    (make-instance '|HallBasisType|)))
```

———-

1.67.2 HeuGcd

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |HeuGcd|
  (progn
    (push '|HeuGcd| *Packages*)
    (make-instance '|HeuGcdType|)))
```

———-

1.68 I

1.68.1 IdealDecompositionPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IdealDecompositionPackage|
  (progn
    (push '|IdealDecompositionPackage| *Packages*)
    (make-instance '|IdealDecompositionPackageType|)))
```

———-

1.68.2 IncrementingMaps

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IncrementingMaps|
  (progn
    (push '|IncrementingMaps| *Packages*)
    (make-instance '|IncrementingMapsType|)))
```

———-

1.68.3 InfiniteProductCharacteristicZero

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteProductCharacteristicZero|
  (progn
    (push '|InfiniteProductCharacteristicZero| *Packages*)
    (make-instance '|InfiniteProductCharacteristicZeroType|)))
```

———-

1.68.4 InfiniteProductFiniteField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteProductFiniteField|
  (progn
    (push '|InfiniteProductFiniteField| *Packages*)
    (make-instance '|InfiniteProductFiniteFieldType|)))
```

———-

1.68.5 InfiniteProductPrimeField

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteProductPrimeField|
  (progn
    (push '|InfiniteProductPrimeField| *Packages*)
    (make-instance '|InfiniteProductPrimeFieldType|)))
```

———-

1.68.6 InfiniteTupleFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteTupleFunctions2|
  (progn
    (push '|InfiniteTupleFunctions2| *Packages*)
    (make-instance '|InfiniteTupleFunctions2Type|)))
```

———-

1.68.7 InfiniteTupleFunctions3

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InfiniteTupleFunctions3|
  (progn
    (push '|InfiniteTupleFunctions3| *Packages*)
    (make-instance '|InfiniteTupleFunctions3Type|)))
```

———-

1.68.8 Infinity

— sane —

```
(defclass |InfinityType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "Infinity")
   (marker :initform 'package)
   (abbreviation :initform 'INFINITY)
   (comment :initform (list
     "Default infinity signatures for the interpreter"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Infinity|
  (progn
    (push '|Infinity| *Packages*)
    (make-instance '|InfinityType|)))
```

———-

1.68.9 InnerAlgFactor

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerAlgFactor|
  (progn
    (push '|InnerAlgFactor| *Packages*)
    (make-instance '|InnerAlgFactorType|)))
```

———-

1.68.10 InnerCommonDenominator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerCommonDenominator|
  (progn
    (push '|InnerCommonDenominator| *Packages*)
    (make-instance '|InnerCommonDenominatorType|)))
```

———-

1.68.11 InnerMatrixLinearAlgebraFunctions

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerMatrixLinearAlgebraFunctions|
  (progn
    (push '|InnerMatrixLinearAlgebraFunctions| *Packages*)
    (make-instance '|InnerMatrixLinearAlgebraFunctionsType|)))
```

———-

1.68.12 InnerMatrixQuotientFieldFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerMatrixQuotientFieldFunctions|
  (progn
    (push '|InnerMatrixQuotientFieldFunctions| *Packages*)
    (make-instance '|InnerMatrixQuotientFieldFunctionsType|)))
```

———-

1.68.13 InnerModularGcd

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerModularGcd|
  (progn
    (push '|InnerModularGcd| *Packages*)
    (make-instance '|InnerModularGcdType|)))
```

———-

1.68.14 InnerMultFact

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |InnerMultFact|
  (progn
    (push '|InnerMultFact| *Packages*)
    (make-instance '|InnerMultFactType|)))
```

———-

1.68.15 InnerNormalBasisFieldFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerNormalBasisFieldFunctions|
  (progn
    (push '|InnerNormalBasisFieldFunctions| *Packages*)
    (make-instance '|InnerNormalBasisFieldFunctionsType|)))
```

———-

1.68.16 InnerNumericEigenPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |InnerNumericEigenPackage|
  (progn
    (push '|InnerNumericEigenPackage| *Packages*)
    (make-instance '|InnerNumericEigenPackageType|)))
```

———-

1.68.17 InnerNumericFloatSolvePackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerNumericFloatSolvePackage|
  (progn
    (push '|InnerNumericFloatSolvePackage| *Packages*)
    (make-instance '|InnerNumericFloatSolvePackageType|)))
```

———-

1.68.18 InnerPolySign

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerPolySign|
  (progn
    (push '|InnerPolySign| *Packages*)
    (make-instance '|InnerPolySignType|)))
```

———-

1.68.19 InnerPolySum

— **sane** —

```
(defclass |InnerPolySumType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InnerPolySum")
   (marker :initform 'package)
   (abbreviation :initform 'ISUMP)
   (comment :initform (list
     "Tools for the summation packages of polynomials"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerPolySum|
  (progn
    (push '|InnerPolySum| *Packages*)
    (make-instance '|InnerPolySumType|)))
```

———-

1.68.20 InnerTrigonometricManipulations

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InnerTrigonometricManipulations|
  (progn
    (push '|InnerTrigonometricManipulations| *Packages*)
    (make-instance '|InnerTrigonometricManipulationsType|)))
```

———-

1.68.21 InputFormFunctions1

— **sane** —

```
(defclass |InputFormFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InputFormFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'INFORM1)
   (comment :initform (list
     "Tools for manipulating input forms."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InputFormFunctions1|
  (progn
    (push '|InputFormFunctions1| *Packages*)
    (make-instance '|InputFormFunctions1Type|)))
```

———-

1.68.22 InterfaceGroebnerPackage

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InterfaceGroebnerPackage|
  (progn
    (push '|InterfaceGroebnerPackage| *Packages*)
    (make-instance '|InterfaceGroebnerPackageType|)))
```

———-

1.68.23 IntegerBits

— **sane** —

```
(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 "))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerBits|
  (progn
    (push '|IntegerBits| *Packages*)
    (make-instance '|IntegerBitsType|)))
```

———-

1.68.24 IntegerCombinatoricFunctions

— **sane** —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerCombinatoricFunctions|
  (progn
    (push '|IntegerCombinatoricFunctions| *Packages*)
    (make-instance '|IntegerCombinatoricFunctionsType|)))
```

———-

1.68.25 IntegerFactorizationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerFactorizationPackage|
  (progn
    (push '|IntegerFactorizationPackage| *Packages*)
    (make-instance '|IntegerFactorizationPackageType|)))
```

———-

1.68.26 IntegerLinearDependence

— **sane** —

```
(defclass |IntegerLinearDependenceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerLinearDependence")
   (marker :initform 'package)
```

```
   (abbreviation :initform 'ZLINDEP)
   (comment :initform (list
     "Test for linear dependence over the integers."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerLinearDependence|
  (progn
    (push '|IntegerLinearDependence| *Packages*)
    (make-instance '|IntegerLinearDependenceType|)))
```

———-

1.68.27 IntegerNumberTheoryFunctions

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerNumberTheoryFunctions|
  (progn
    (push '|IntegerNumberTheoryFunctions| *Packages*)
    (make-instance '|IntegerNumberTheoryFunctionsType|)))
```

———-

1.68.28 IntegerPrimesPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerPrimesPackage|
  (progn
    (push '|IntegerPrimesPackage| *Packages*)
    (make-instance '|IntegerPrimesPackageType|)))
```

———-

1.68.29 IntegerRetractions

— **sane** —

```
(defclass |IntegerRetractionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegerRetractions")
   (marker :initform 'package)
   (abbreviation :initform 'INTRET)
   (comment :initform (list
     "Provides integer testing and retraction functions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerRetractions|
  (progn
    (push '|IntegerRetractions| *Packages*)
    (make-instance '|IntegerRetractionsType|)))
```

———-

1.68.30 IntegerRoots

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerRoots|
  (progn
    (push '|IntegerRoots| *Packages*)
    (make-instance '|IntegerRootsType|)))
```

———-

1.68.31 IntegerSolveLinearPolynomialEquation

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegerSolveLinearPolynomialEquation|
  (progn
    (push '|IntegerSolveLinearPolynomialEquation| *Packages*)
    (make-instance '|IntegerSolveLinearPolynomialEquationType|)))
```

———-

1.68.32 IntegralBasisTools

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegralBasisTools|
  (progn
    (push '|IntegralBasisTools| *Packages*)
    (make-instance '|IntegralBasisToolsType|)))
```

———-

1.68.33 IntegralBasisPolynomialTools

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegralBasisPolynomialTools|
  (progn
    (push '|IntegralBasisPolynomialTools| *Packages*)
    (make-instance '|IntegralBasisPolynomialToolsType|)))
```

———-

1.68.34 IntegrationResultFunctions2

— **sane** —

```
(defclass |IntegrationResultFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegrationResultFunctions2")
```

```
   (marker :initform 'package)
   (abbreviation :initform 'IR2)
   (comment :initform (list
     "Internally used by the integration packages"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResultFunctions2|
  (progn
    (push '|IntegrationResultFunctions2| *Packages*)
    (make-instance '|IntegrationResultFunctions2Type|)))
```

———-

1.68.35 IntegrationResultRFToFunction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResultRFToFunction|
  (progn
    (push '|IntegrationResultRFToFunction| *Packages*)
    (make-instance '|IntegrationResultRFToFunctionType|)))
```

———-

1.68.36 IntegrationResultToFunction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationResultToFunction|
  (progn
    (push '|IntegrationResultToFunction| *Packages*)
    (make-instance '|IntegrationResultToFunctionType|)))
```

———-

1.68.37 IntegrationTools

— **sane** —

```
(defclass |IntegrationToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "IntegrationTools")
   (marker :initform 'package)
   (abbreviation :initform 'INTTOOLS)
   (comment :initform (list
     "Tools for the integrator"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntegrationTools|
  (progn
    (push '|IntegrationTools| *Packages*)
    (make-instance '|IntegrationToolsType|)))
```

———-

1.68.38 InternalPrintPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InternalPrintPackage|
  (progn
    (push '|InternalPrintPackage| *Packages*)
    (make-instance '|InternalPrintPackageType|)))
```

———-

1.68.39 InternalRationalUnivariateRepresentationPackage

— **sane** —

```
(defclass |InternalRationalUnivariateRepresentationPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InternalRationalUnivariateRepresentationPackage")
   (marker :initform 'package)
   (abbreviation :initform 'IRURPK)
   (comment :initform nil)
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InternalRationalUnivariateRepresentationPackage|
  (progn
    (push '|InternalRationalUnivariateRepresentationPackage| *Packages*)
    (make-instance '|InternalRationalUnivariateRepresentationPackageType|)))
```

———-

1.68.40 InterpolateFormsPackage

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InterpolateFormsPackage|
  (progn
    (push '|InterpolateFormsPackage| *Packages*)
    (make-instance '|InterpolateFormsPackageType|)))
```

———-

1.68.41 IntersectionDivisorPackage

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IntersectionDivisorPackage|
  (progn
    (push '|IntersectionDivisorPackage| *Packages*)
    (make-instance '|IntersectionDivisorPackageType|)))
```

———-

1.68.42 IrredPolyOverFiniteField

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IrredPolyOverFiniteField|
  (progn
    (push '|IrredPolyOverFiniteField| *Packages*)
    (make-instance '|IrredPolyOverFiniteFieldType|)))
```

———-

1.68.43 IrrRepSymNatPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |IrrRepSymNatPackage|
  (progn
    (push '|IrrRepSymNatPackage| *Packages*)
    (make-instance '|IrrRepSymNatPackageType|)))
```

———-

1.68.44 InverseLaplaceTransform

— sane —

```
(defclass |InverseLaplaceTransformType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "InverseLaplaceTransform")
   (marker :initform 'package)
   (abbreviation :initform 'INVLAPLA)
   (comment :initform (list
     "This package computes the inverse Laplace Transform."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |InverseLaplaceTransform|
  (progn
    (push '|InverseLaplaceTransform| *Packages*)
    (make-instance '|InverseLaplaceTransformType|)))
```

———-

1.69 K

1.69.1 KernelFunctions2

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |KernelFunctions2|
  (progn
    (push '|KernelFunctions2| *Packages*)
    (make-instance '|KernelFunctions2Type|)))
```

———-

1.69.2 Kovacic

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Kovacic|
  (progn
    (push '|Kovacic| *Packages*)
    (make-instance '|KovacicType|)))
```

———-

1.70 L

1.70.1 LaplaceTransform

— sane —

```
(defclass |LaplaceTransformType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LaplaceTransform")
   (marker :initform 'package)
   (abbreviation :initform 'LAPLACE)
   (comment :initform (list
     "This package computes the forward Laplace Transform."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LaplaceTransform|
  (progn
    (push '|LaplaceTransform| *Packages*)
    (make-instance '|LaplaceTransformType|)))
```

———-

1.70.2 LazardSetSolvingPackage

— sane —

```
(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]."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LazardSetSolvingPackage|
  (progn
    (push '|LazardSetSolvingPackage| *Packages*)
    (make-instance '|LazardSetSolvingPackageType|)))
```

———-

1.70.3 LeadingCoefDetermination

— sane —

```
(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'."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LeadingCoefDetermination|
  (progn
```

```
    (push '|LeadingCoefDetermination| *Packages*)
    (make-instance '|LeadingCoefDeterminationType|)))
```

———-

1.70.4 LexTriangularPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LexTriangularPackage|
  (progn
    (push '|LexTriangularPackage| *Packages*)
    (make-instance '|LexTriangularPackageType|)))
```

———-

1.70.5 LinearDependence

— sane —

```
(defclass |LinearDependenceType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearDependence")
   (marker :initform 'package)
   (abbreviation :initform 'LINDEP)
   (comment :initform (list
```

```
     "Test for linear dependence."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearDependence|
  (progn
    (push '|LinearDependence| *Packages*)
    (make-instance '|LinearDependenceType|)))
```

———-

1.70.6 LinearOrdinaryDifferentialOperatorFactorizer

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperatorFactorizer|
  (progn
    (push '|LinearOrdinaryDifferentialOperatorFactorizer| *Packages*)
    (make-instance '|LinearOrdinaryDifferentialOperatorFactorizerType|)))
```

———-

1.70.7 LinearOrdinaryDifferentialOperatorsOps

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearOrdinaryDifferentialOperatorsOps|
  (progn
    (push '|LinearOrdinaryDifferentialOperatorsOps| *Packages*)
    (make-instance '|LinearOrdinaryDifferentialOperatorsOpsType|)))
```

———-

1.70.8 LinearPolynomialEquationByFractions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearPolynomialEquationByFractions|
  (progn
    (push '|LinearPolynomialEquationByFractions| *Packages*)
    (make-instance '|LinearPolynomialEquationByFractionsType|)))
```

———-

1.70.9 LinearSystemFromPowerSeriesPackage

— sane —

```
(defclass |LinearSystemFromPowerSeriesPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "LinearSystemFromPowerSeriesPackage")
   (marker :initform 'package)
```

```
   (abbreviation :initform 'LISYSER)
   (comment :initform (list
     "Part of the PAFF package"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemFromPowerSeriesPackage|
  (progn
    (push '|LinearSystemFromPowerSeriesPackage| *Packages*)
    (make-instance '|LinearSystemFromPowerSeriesPackageType|)))
```

———-

1.70.10 LinearSystemMatrixPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemMatrixPackage|
  (progn
    (push '|LinearSystemMatrixPackage| *Packages*)
    (make-instance '|LinearSystemMatrixPackageType|)))
```

———-

1.70.11 LinearSystemMatrixPackage1

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemMatrixPackage1|
  (progn
    (push '|LinearSystemMatrixPackage1| *Packages*)
    (make-instance '|LinearSystemMatrixPackage1Type|)))
```

———-

1.70.12 LinearSystemPolynomialPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinearSystemPolynomialPackage|
  (progn
    (push '|LinearSystemPolynomialPackage| *Packages*)
    (make-instance '|LinearSystemPolynomialPackageType|)))
```

———-

1.70.13 LinGroebnerPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinGroebnerPackage|
  (progn
    (push '|LinGroebnerPackage| *Packages*)
    (make-instance '|LinGroebnerPackageType|)))
```

———-

1.70.14 LinesOpPack

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LinesOpPack|
  (progn
    (push '|LinesOpPack| *Packages*)
    (make-instance '|LinesOpPackType|)))
```

———-

1.70.15 LiouvillianFunction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LiouvillianFunction|
  (progn
    (push '|LiouvillianFunction| *Packages*)
    (make-instance '|LiouvillianFunctionType|)))
```

———-

1.70.16 ListFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListFunctions2|
  (progn
    (push '|ListFunctions2| *Packages*)
    (make-instance '|ListFunctions2Type|)))
```

———-

1.70.17 ListFunctions3

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListFunctions3|
  (progn
    (push '|ListFunctions3| *Packages*)
    (make-instance '|ListFunctions3Type|)))
```

———-

1.70.18 ListToMap

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ListToMap|
  (progn
    (push '|ListToMap| *Packages*)
    (make-instance '|ListToMapType|)))
```

———-

1.70.19 LocalParametrizationOfSimplePointPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |LocalParametrizationOfSimplePointPackage|
  (progn
    (push '|LocalParametrizationOfSimplePointPackage| *Packages*)
    (make-instance '|LocalParametrizationOfSimplePointPackageType|)))
```

———-

1.71 M

1.71.1 MakeBinaryCompiledFunction

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeBinaryCompiledFunction|
  (progn
    (push '|MakeBinaryCompiledFunction| *Packages*)
    (make-instance '|MakeBinaryCompiledFunctionType|)))
```

———-

1.71.2 MakeFloatCompiledFunction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeFloatCompiledFunction|
  (progn
    (push '|MakeFloatCompiledFunction| *Packages*)
    (make-instance '|MakeFloatCompiledFunctionType|)))
```

———-

1.71.3 MakeFunction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeFunction|
  (progn
    (push '|MakeFunction| *Packages*)
    (make-instance '|MakeFunctionType|)))
```

———-

1.71.4 MakeRecord

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeRecord|
  (progn
    (push '|MakeRecord| *Packages*)
    (make-instance '|MakeRecordType|)))
```

———-

1.71.5 MakeUnaryCompiledFunction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MakeUnaryCompiledFunction|
  (progn
    (push '|MakeUnaryCompiledFunction| *Packages*)
    (make-instance '|MakeUnaryCompiledFunctionType|)))
```

———-

1.71.6 MappingPackageInternalHacks1

— sane —

```
(defclass |MappingPackageInternalHacks1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackageInternalHacks1")
   (marker :initform 'package)
   (abbreviation :initform 'MAPHACK1)
   (comment :initform (list
     "Various Currying operations."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackageInternalHacks1|
  (progn
    (push '|MappingPackageInternalHacks1| *Packages*)
    (make-instance '|MappingPackageInternalHacks1Type|)))
```

———-

1.71.7 MappingPackageInternalHacks2

— sane —

```
(defclass |MappingPackageInternalHacks2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackageInternalHacks2")
   (marker :initform 'package)
   (abbreviation :initform 'MAPHACK2)
   (comment :initform (list
     "Various Currying operations."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackageInternalHacks2|
  (progn
    (push '|MappingPackageInternalHacks2| *Packages*)
    (make-instance '|MappingPackageInternalHacks2Type|)))
```

———-

1.71.8 MappingPackageInternalHacks3

— sane —

```
(defclass |MappingPackageInternalHacks3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackageInternalHacks3")
   (marker :initform 'package)
   (abbreviation :initform 'MAPHACK3)
   (comment :initform (list
     "Various Currying operations."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackageInternalHacks3|
  (progn
    (push '|MappingPackageInternalHacks3| *Packages*)
    (make-instance '|MappingPackageInternalHacks3Type|)))
```

———-

1.71.9 MappingPackage1

— sane —

```
(defclass |MappingPackage1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackage1")
   (marker :initform 'package)
   (abbreviation :initform 'MAPPKG1)
   (comment :initform (list
     "Various Currying operations."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackage1|
  (progn
    (push '|MappingPackage1| *Packages*)
    (make-instance '|MappingPackage1Type|)))
```

———-

1.71.10 MappingPackage2

— sane —

```
(defclass |MappingPackage2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackage2")
   (marker :initform 'package)
   (abbreviation :initform 'MAPPKG2)
   (comment :initform (list
     "Various Currying operations."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackage2|
  (progn
    (push '|MappingPackage2| *Packages*)
    (make-instance '|MappingPackage2Type|)))
```

———-

1.71.11 MappingPackage3

— sane —

```
(defclass |MappingPackage3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MappingPackage3")
   (marker :initform 'package)
   (abbreviation :initform 'MAPPKG3)
   (comment :initform (list
     "Various Currying operations."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackage3|
  (progn
    (push '|MappingPackage3| *Packages*)
    (make-instance '|MappingPackage3Type|)))
```

———-

1.71.12 MappingPackage4

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MappingPackage4|
  (progn
    (push '|MappingPackage4| *Packages*)
    (make-instance '|MappingPackage4Type|)))
```

———-

1.71.13 MatrixCategoryFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixCategoryFunctions2|
  (progn
    (push '|MatrixCategoryFunctions2| *Packages*)
    (make-instance '|MatrixCategoryFunctions2Type|)))
```

———-

1.71.14 MatrixCommonDenominator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixCommonDenominator|
  (progn
    (push '|MatrixCommonDenominator| *Packages*)
    (make-instance '|MatrixCommonDenominatorType|)))
```

———-

1.71.15 MatrixLinearAlgebraFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixLinearAlgebraFunctions|
  (progn
    (push '|MatrixLinearAlgebraFunctions| *Packages*)
    (make-instance '|MatrixLinearAlgebraFunctionsType|)))
```

———-

1.71.16 MatrixManipulation

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MatrixManipulation|
  (progn
    (push '|MatrixManipulation| *Packages*)
    (make-instance '|MatrixManipulationType|)))
```

———-

1.71.17 MergeThing

— sane —

```
(defclass |MergeThingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MergeThing")
   (marker :initform 'package)
   (abbreviation :initform 'MTHING)
   (comment :initform (list
     "This package exports tools for merging lists"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MergeThing|
  (progn
    (push '|MergeThing| *Packages*)
    (make-instance '|MergeThingType|)))
```

———-

1.71.18 MeshCreationRoutinesForThreeDimensions

— sane —

```
(defclass |MeshCreationRoutinesForThreeDimensionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MeshCreationRoutinesForThreeDimensions")
   (marker :initform 'package)
   (abbreviation :initform 'MESH)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MeshCreationRoutinesForThreeDimensions|
  (progn
    (push '|MeshCreationRoutinesForThreeDimensions| *Packages*)
    (make-instance '|MeshCreationRoutinesForThreeDimensionsType|)))
```

———-

1.71.19 ModularDistinctDegreeFactorizer

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularDistinctDegreeFactorizer|
  (progn
    (push '|ModularDistinctDegreeFactorizer| *Packages*)
    (make-instance '|ModularDistinctDegreeFactorizerType|)))
```

———-

1.71.20 ModularHermitianRowReduction

— sane —

```
(defclass |ModularHermitianRowReductionType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ModularHermitianRowReduction")
   (marker :initform 'package)
   (abbreviation :initform 'MHROWRED)
   (comment :initform (list
     "Modular hermitian row reduction."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ModularHermitianRowReduction|
  (progn
    (push '|ModularHermitianRowReduction| *Packages*)
    (make-instance '|ModularHermitianRowReductionType|)))
```

———-

1.71.21 MonoidRingFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonoidRingFunctions2|
  (progn
    (push '|MonoidRingFunctions2| *Packages*)
    (make-instance '|MonoidRingFunctions2Type|)))
```

———-

1.71.22 MonomialExtensionTools

— sane —

```
(defclass |MonomialExtensionToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MonomialExtensionTools")
   (marker :initform 'package)
   (abbreviation :initform 'MONOTOOL)
   (comment :initform (list
     "Tools for handling monomial extensions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MonomialExtensionTools|
  (progn
    (push '|MonomialExtensionTools| *Packages*)
    (make-instance '|MonomialExtensionToolsType|)))
```

———-

1.71.23 MoreSystemCommands

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MoreSystemCommands|
  (progn
    (push '|MoreSystemCommands| *Packages*)
    (make-instance '|MoreSystemCommandsType|)))
```

———-

1.71.24 MPolyCatPolyFactorizer

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MPolyCatPolyFactorizer|
  (progn
    (push '|MPolyCatPolyFactorizer| *Packages*)
    (make-instance '|MPolyCatPolyFactorizerType|)))
```

———-

1.71.25 MPolyCatRationalFunctionFactorizer

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |MPolyCatRationalFunctionFactorizer|
  (progn
    (push '|MPolyCatRationalFunctionFactorizer| *Packages*)
    (make-instance '|MPolyCatRationalFunctionFactorizerType|)))
```

———-

1.71.26 MPolyCatFunctions2

— sane —

```
(defclass |MPolyCatFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MPolyCatFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'MPC2)
   (comment :initform (list
     "Utilities for MPolyCat"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MPolyCatFunctions2|
  (progn
    (push '|MPolyCatFunctions2| *Packages*)
    (make-instance '|MPolyCatFunctions2Type|)))
```

———-

1.71.27 MPolyCatFunctions3

— sane —

```
(defclass |MPolyCatFunctions3Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "MPolyCatFunctions3")
   (marker :initform 'package)
   (abbreviation :initform 'MPC3)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MPolyCatFunctions3|
  (progn
```

```
    (push '|MPolyCatFunctions3| *Packages*)
    (make-instance '|MPolyCatFunctions3Type|)))
```

———-

1.71.28 MRationalFactorize

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MRationalFactorize|
  (progn
    (push '|MRationalFactorize| *Packages*)
    (make-instance '|MRationalFactorizeType|)))
```

———-

1.71.29 MultFiniteFactorize

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultFiniteFactorize|
  (progn
```

```
    (push '|MultFiniteFactorize| *Packages*)
    (make-instance '|MultFiniteFactorizeType|)))
```

———-

1.71.30 MultipleMap

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultipleMap|
  (progn
    (push '|MultipleMap| *Packages*)
    (make-instance '|MultipleMapType|)))
```

———-

1.71.31 MultiVariableCalculusFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultiVariableCalculusFunctions|
```

```
  (progn
    (push '|MultiVariableCalculusFunctions| *Packages*)
    (make-instance '|MultiVariableCalculusFunctionsType|)))
```

———-

1.71.32 MultivariateFactorize

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariateFactorize|
  (progn
    (push '|MultivariateFactorize| *Packages*)
    (make-instance '|MultivariateFactorizeType|)))
```

———-

1.71.33 MultivariateLifting

— sane —

```
(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]."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |MultivariateLifting|
  (progn
    (push '|MultivariateLifting| *Packages*)
    (make-instance '|MultivariateLiftingType|)))
```

———-

1.71.34 MultivariateSquareFree

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |MultivariateSquareFree|
  (progn
    (push '|MultivariateSquareFree| *Packages*)
    (make-instance '|MultivariateSquareFreeType|)))
```

———-

1.72 N

1.72.1 NagEigenPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagEigenPackage|
  (progn
    (push '|NagEigenPackage| *Packages*)
    (make-instance '|NagEigenPackageType|)))
```

———-

1.72.2 NagFittingPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |NagFittingPackage|
  (progn
    (push '|NagFittingPackage| *Packages*)
    (make-instance '|NagFittingPackageType|)))
```

———-

1.72.3 NagLinearEquationSolvingPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagLinearEquationSolvingPackage|
  (progn
    (push '|NagLinearEquationSolvingPackage| *Packages*)
    (make-instance '|NagLinearEquationSolvingPackageType|)))
```

———-

1.72.4 NAGLinkSupportPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NAGLinkSupportPackage|
  (progn
    (push '|NAGLinkSupportPackage| *Packages*)
    (make-instance '|NAGLinkSupportPackageType|)))
```

———-

1.72.5 NagIntegrationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagIntegrationPackage|
  (progn
    (push '|NagIntegrationPackage| *Packages*)
    (make-instance '|NagIntegrationPackageType|)))
```

———-

1.72.6 NagInterpolationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagInterpolationPackage|
  (progn
    (push '|NagInterpolationPackage| *Packages*)
    (make-instance '|NagInterpolationPackageType|)))
```

———-

1.72.7 NagLapack

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagLapack|
  (progn
    (push '|NagLapack| *Packages*)
    (make-instance '|NagLapackType|)))
```

———-

1.72.8 NagMatrixOperationsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagMatrixOperationsPackage|
  (progn
    (push '|NagMatrixOperationsPackage| *Packages*)
    (make-instance '|NagMatrixOperationsPackageType|)))
```

———-

1.72.9 NagOptimisationPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagOptimisationPackage|
  (progn
    (push '|NagOptimisationPackage| *Packages*)
    (make-instance '|NagOptimisationPackageType|)))
```

———-

1.72.10 NagOrdinaryDifferentialEquationsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagOrdinaryDifferentialEquationsPackage|
  (progn
    (push '|NagOrdinaryDifferentialEquationsPackage| *Packages*)
    (make-instance '|NagOrdinaryDifferentialEquationsPackageType|)))
```

———-

1.72.11 NagPartialDifferentialEquationsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |NagPartialDifferentialEquationsPackage|
  (progn
    (push '|NagPartialDifferentialEquationsPackage| *Packages*)
    (make-instance '|NagPartialDifferentialEquationsPackageType|)))
```

———-

1.72.12 NagPolynomialRootsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagPolynomialRootsPackage|
  (progn
    (push '|NagPolynomialRootsPackage| *Packages*)
    (make-instance '|NagPolynomialRootsPackageType|)))
```

———-

1.72.13 NagRootFindingPackage

— sane —

```
(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.)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |NagRootFindingPackage|
  (progn
    (push '|NagRootFindingPackage| *Packages*)
    (make-instance '|NagRootFindingPackageType|)))
```

———-

1.72.14 NagSeriesSummationPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NagSeriesSummationPackage|
  (progn
    (push '|NagSeriesSummationPackage| *Packages*)
    (make-instance '|NagSeriesSummationPackageType|)))
```

———-

1.72.15 NagSpecialFunctionsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |NagSpecialFunctionsPackage|
  (progn
    (push '|NagSpecialFunctionsPackage| *Packages*)
    (make-instance '|NagSpecialFunctionsPackageType|)))
```

———-

1.72.16 NewSparseUnivariatePolynomialFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewSparseUnivariatePolynomialFunctions2|
  (progn
    (push '|NewSparseUnivariatePolynomialFunctions2| *Packages*)
    (make-instance '|NewSparseUnivariatePolynomialFunctions2Type|)))
```

———-

1.72.17 NewtonInterpolation

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewtonInterpolation|
  (progn
    (push '|NewtonInterpolation| *Packages*)
    (make-instance '|NewtonInterpolationType|)))
```

———-

1.72.18 NewtonPolygon

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NewtonPolygon|
  (progn
    (push '|NewtonPolygon| *Packages*)
    (make-instance '|NewtonPolygonType|)))
```

———-

1.72.19 NonCommutativeOperatorDivision

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonCommutativeOperatorDivision|
  (progn
    (push '|NonCommutativeOperatorDivision| *Packages*)
    (make-instance '|NonCommutativeOperatorDivisionType|)))
```

———-

1.72.20 NoneFunctions1

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NoneFunctions1|
  (progn
    (push '|NoneFunctions1| *Packages*)
    (make-instance '|NoneFunctions1Type|)))
```

———-

1.72.21 NonLinearFirstOrderODESolver

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonLinearFirstOrderODESolver|
  (progn
    (push '|NonLinearFirstOrderODESolver| *Packages*)
    (make-instance '|NonLinearFirstOrderODESolverType|)))
```

———-

1.72.22 NonLinearSolvePackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NonLinearSolvePackage|
  (progn
    (push '|NonLinearSolvePackage| *Packages*)
    (make-instance '|NonLinearSolvePackageType|)))
```

———-

1.72.23 NormalizationPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormalizationPackage|
  (progn
    (push '|NormalizationPackage| *Packages*)
    (make-instance '|NormalizationPackageType|)))
```

———-

1.72.24 NormInMonogenicAlgebra

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormInMonogenicAlgebra|
  (progn
    (push '|NormInMonogenicAlgebra| *Packages*)
    (make-instance '|NormInMonogenicAlgebraType|)))
```

———-

1.72.25 NormRetractPackage

— sane —

```
(defclass |NormRetractPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "NormRetractPackage")
   (marker :initform 'package)
   (abbreviation :initform 'NORMRETR)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NormRetractPackage|
  (progn
    (push '|NormRetractPackage| *Packages*)
    (make-instance '|NormRetractPackageType|)))
```

———-

1.72.26 NPCoef

— sane —

```
(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]."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NPCoef|
  (progn
    (push '|NPCoef| *Packages*)
    (make-instance '|NPCoefType|)))
```

———-

1.72.27 NumberFieldIntegralBasis

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumberFieldIntegralBasis|
  (progn
    (push '|NumberFieldIntegralBasis| *Packages*)
    (make-instance '|NumberFieldIntegralBasisType|)))
```

———-

1.72.28 NumberFormats

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumberFormats|
  (progn
    (push '|NumberFormats| *Packages*)
    (make-instance '|NumberFormatsType|)))
```

———-

1.72.29 NumberTheoreticPolynomialFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumberTheoreticPolynomialFunctions|
  (progn
    (push '|NumberTheoreticPolynomialFunctions| *Packages*)
    (make-instance '|NumberTheoreticPolynomialFunctionsType|)))
```

———-

1.72.30 Numeric

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Numeric|
  (progn
    (push '|Numeric| *Packages*)
    (make-instance '|NumericType|)))
```

———-

1.72.31 NumericalOrdinaryDifferentialEquations

— sane —

```
(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."))
  (arglist :initform nil)
  (macros :initform nil)
  (withlist :initform nil)
  (haslist :initform nil)
  (addlist :initform nil)))

(defvar |NumericalOrdinaryDifferentialEquations|
  (progn
    (push '|NumericalOrdinaryDifferentialEquations| *Packages*)
    (make-instance '|NumericalOrdinaryDifferentialEquationsType|)))
```

———-

1.72.32 NumericalQuadrature

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericalQuadrature|
  (progn
    (push '|NumericalQuadrature| *Packages*)
    (make-instance '|NumericalQuadratureType|)))
```

———-

1.72.33 NumericComplexEigenPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |NumericComplexEigenPackage|
  (progn
    (push '|NumericComplexEigenPackage| *Packages*)
    (make-instance '|NumericComplexEigenPackageType|)))
```

———-

1.72.34 NumericContinuedFraction

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericContinuedFraction|
  (progn
    (push '|NumericContinuedFraction| *Packages*)
    (make-instance '|NumericContinuedFractionType|)))
```

———-

1.72.35 NumericRealEigenPackage

— sane —

```
(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."))
   (arglist :initform nil)
```

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

(defvar |NumericRealEigenPackage|
  (progn
    (push '|NumericRealEigenPackage| *Packages*)
    (make-instance '|NumericRealEigenPackageType|)))
```

———-

1.72.36 NumericTubePlot

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |NumericTubePlot|
  (progn
    (push '|NumericTubePlot| *Packages*)
    (make-instance '|NumericTubePlotType|)))
```

———-

1.73 O

1.73.1 OctonionCategoryFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OctonionCategoryFunctions2|
  (progn
    (push '|OctonionCategoryFunctions2| *Packages*)
    (make-instance '|OctonionCategoryFunctions2Type|)))
```

———-

1.73.2 ODEIntegration

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ODEIntegration|
  (progn
    (push '|ODEIntegration| *Packages*)
    (make-instance '|ODEIntegrationType|)))
```

———-

1.73.3 ODETools

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ODETools|
  (progn
    (push '|ODETools| *Packages*)
    (make-instance '|ODEToolsType|)))
```

———-

1.73.4 OneDimensionalArrayFunctions2

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OneDimensionalArrayFunctions2|
  (progn
    (push '|OneDimensionalArrayFunctions2| *Packages*)
    (make-instance '|OneDimensionalArrayFunctions2Type|)))
```

———-

1.73.5 OnePointCompletionFunctions2

— **sane** —

```
(defclass |OnePointCompletionFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OnePointCompletionFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ONECOMP2)
   (comment :initform (list
```

```
     "Lifting of maps to one-point completions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OnePointCompletionFunctions2|
  (progn
    (push '|OnePointCompletionFunctions2| *Packages*)
    (make-instance '|OnePointCompletionFunctions2Type|)))
```

———-

1.73.6 OpenMathPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathPackage|
  (progn
    (push '|OpenMathPackage| *Packages*)
    (make-instance '|OpenMathPackageType|)))
```

———-

1.73.7 OpenMathServerPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OpenMathServerPackage|
  (progn
    (push '|OpenMathServerPackage| *Packages*)
    (make-instance '|OpenMathServerPackageType|)))
```

———-

1.73.8 OperationsQuery

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OperationsQuery|
  (progn
    (push '|OperationsQuery| *Packages*)
    (make-instance '|OperationsQueryType|)))
```

———-

1.73.9 OrderedCompletionFunctions2

— **sane** —

```
(defclass |OrderedCompletionFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OrderedCompletionFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'ORDCOMP2)
```

```
   (comment :initform (list
     "Lifting of maps to ordered completions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderedCompletionFunctions2|
  (progn
    (push '|OrderedCompletionFunctions2| *Packages*)
    (make-instance '|OrderedCompletionFunctions2Type|)))
```

———-

1.73.10 OrderingFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrderingFunctions|
  (progn
    (push '|OrderingFunctions| *Packages*)
    (make-instance '|OrderingFunctionsType|)))
```

———-

1.73.11 OrthogonalPolynomialFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OrthogonalPolynomialFunctions|
  (progn
    (push '|OrthogonalPolynomialFunctions| *Packages*)
    (make-instance '|OrthogonalPolynomialFunctionsType|)))
```

———-

1.73.12 OutputPackage

— sane —

```
(defclass |OutputPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "OutputPackage")
   (marker :initform 'package)
   (abbreviation :initform 'OUT)
   (comment :initform (list
     "OutPackage allows pretty-printing from programs."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |OutputPackage|
  (progn
    (push '|OutputPackage| *Packages*)
    (make-instance '|OutputPackageType|)))
```

———-

1.74 P

1.74.1 PackageForAlgebraicFunctionField

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PackageForAlgebraicFunctionField|
  (progn
    (push '|PackageForAlgebraicFunctionField| *Packages*)
    (make-instance '|PackageForAlgebraicFunctionFieldType|)))
```

———-

1.74.2 PackageForAlgebraicFunctionFieldOverFiniteField

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PackageForAlgebraicFunctionFieldOverFiniteField|
  (progn
    (push '|PackageForAlgebraicFunctionFieldOverFiniteField| *Packages*)
    (make-instance '|PackageForAlgebraicFunctionFieldOverFiniteFieldType|)))
```

———-

1.74.3 PackageForPoly

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PackageForPoly|
  (progn
    (push '|PackageForPoly| *Packages*)
    (make-instance '|PackageForPolyType|)))
```

———-

1.74.4 PadeApproximantPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PadeApproximantPackage|
  (progn
    (push '|PadeApproximantPackage| *Packages*)
    (make-instance '|PadeApproximantPackageType|)))
```

———-

1.74.5 PadeApproximants

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PadeApproximants|
  (progn
    (push '|PadeApproximants| *Packages*)
    (make-instance '|PadeApproximantsType|)))
```

———-

1.74.6 PAdicWildFunctionFieldIntegralBasis

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PAdicWildFunctionFieldIntegralBasis|
  (progn
    (push '|PAdicWildFunctionFieldIntegralBasis| *Packages*)
    (make-instance '|PAdicWildFunctionFieldIntegralBasisType|)))
```

———-

1.74.7 ParadoxicalCombinatorsForStreams

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParadoxicalCombinatorsForStreams|
  (progn
    (push '|ParadoxicalCombinatorsForStreams| *Packages*)
    (make-instance '|ParadoxicalCombinatorsForStreamsType|)))
```

———-

1.74.8 ParametricLinearEquations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricLinearEquations|
  (progn
    (push '|ParametricLinearEquations| *Packages*)
    (make-instance '|ParametricLinearEquationsType|)))
```

———-

1.74.9 ParametricPlaneCurveFunctions2

— sane —

```
(defclass |ParametricPlaneCurveFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricPlaneCurveFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PARPC2)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricPlaneCurveFunctions2|
  (progn
    (push '|ParametricPlaneCurveFunctions2| *Packages*)
    (make-instance '|ParametricPlaneCurveFunctions2Type|)))
```

———-

1.74.10 ParametricSpaceCurveFunctions2

— sane —

```
(defclass |ParametricSpaceCurveFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricSpaceCurveFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PARSC2)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSpaceCurveFunctions2|
  (progn
    (push '|ParametricSpaceCurveFunctions2| *Packages*)
    (make-instance '|ParametricSpaceCurveFunctions2Type|)))
```

———-

1.74.11 ParametricSurfaceFunctions2

— sane —

```
(defclass |ParametricSurfaceFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ParametricSurfaceFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PARSU2)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametricSurfaceFunctions2|
  (progn
    (push '|ParametricSurfaceFunctions2| *Packages*)
    (make-instance '|ParametricSurfaceFunctions2Type|)))
```

———-

1.74.12 ParametrizationPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ParametrizationPackage|
  (progn
    (push '|ParametrizationPackage| *Packages*)
    (make-instance '|ParametrizationPackageType|)))
```

———-

1.74.13 PartialFractionPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartialFractionPackage|
  (progn
    (push '|PartialFractionPackage| *Packages*)
    (make-instance '|PartialFractionPackageType|)))
```

———-

1.74.14 PartitionsAndPermutations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PartitionsAndPermutations|
  (progn
    (push '|PartitionsAndPermutations| *Packages*)
    (make-instance '|PartitionsAndPermutationsType|)))
```

———-

1.74.15 PatternFunctions1

— sane —

```
(defclass |PatternFunctions1Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternFunctions1")
   (marker :initform 'package)
   (abbreviation :initform 'PATTERN1)
   (comment :initform (list
     "Utilities for handling patterns"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternFunctions1|
  (progn
    (push '|PatternFunctions1| *Packages*)
    (make-instance '|PatternFunctions1Type|)))
```

———-

1.74.16 PatternFunctions2

— sane —

```
(defclass |PatternFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PATTERN2)
   (comment :initform (list
     "Lifts maps to patterns"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternFunctions2|
  (progn
    (push '|PatternFunctions2| *Packages*)
    (make-instance '|PatternFunctions2Type|)))
```

———-

1.74.17 PatternMatch

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatch|
  (progn
    (push '|PatternMatch| *Packages*)
    (make-instance '|PatternMatchType|)))
```

———-

1.74.18 PatternMatchAssertions

— sane —

```
(defclass |PatternMatchAssertionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchAssertions")
   (marker :initform 'package)
   (abbreviation :initform 'PMASS)
   (comment :initform (list
     "Attaching assertions to symbols for pattern matching;"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchAssertions|
  (progn
    (push '|PatternMatchAssertions| *Packages*)
    (make-instance '|PatternMatchAssertionsType|)))
```

———-

1.74.19 PatternMatchFunctionSpace

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchFunctionSpace|
  (progn
    (push '|PatternMatchFunctionSpace| *Packages*)
    (make-instance '|PatternMatchFunctionSpaceType|)))
```

———-

1.74.20 PatternMatchIntegerNumberSystem

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchIntegerNumberSystem|
  (progn
    (push '|PatternMatchIntegerNumberSystem| *Packages*)
    (make-instance '|PatternMatchIntegerNumberSystemType|)))
```

———-

1.74.21 PatternMatchIntegration

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchIntegration|
  (progn
    (push '|PatternMatchIntegration| *Packages*)
    (make-instance '|PatternMatchIntegrationType|)))
```

———-

1.74.22 PatternMatchKernel

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchKernel|
  (progn
    (push '|PatternMatchKernel| *Packages*)
    (make-instance '|PatternMatchKernelType|)))
```

———-

1.74.23 PatternMatchListAggregate

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchListAggregate|
  (progn
    (push '|PatternMatchListAggregate| *Packages*)
    (make-instance '|PatternMatchListAggregateType|)))
```

———-

1.74.24 PatternMatchPolynomialCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchPolynomialCategory|
  (progn
    (push '|PatternMatchPolynomialCategory| *Packages*)
    (make-instance '|PatternMatchPolynomialCategoryType|)))
```

———-

1.74.25 PatternMatchPushDown

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchPushDown|
  (progn
    (push '|PatternMatchPushDown| *Packages*)
    (make-instance '|PatternMatchPushDownType|)))
```

———-

1.74.26 PatternMatchQuotientFieldCategory

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchQuotientFieldCategory|
  (progn
    (push '|PatternMatchQuotientFieldCategory| *Packages*)
    (make-instance '|PatternMatchQuotientFieldCategoryType|)))
```

———-

1.74.27 PatternMatchResultFunctions2

— sane —

```
(defclass |PatternMatchResultFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PatternMatchResultFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PATRES2)
   (comment :initform (list
     "Lifts maps to pattern matching results."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchResultFunctions2|
  (progn
    (push '|PatternMatchResultFunctions2| *Packages*)
    (make-instance '|PatternMatchResultFunctions2Type|)))
```

———-

1.74.28 PatternMatchSymbol

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchSymbol|
  (progn
    (push '|PatternMatchSymbol| *Packages*)
    (make-instance '|PatternMatchSymbolType|)))
```

———-

1.74.29 PatternMatchTools

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PatternMatchTools|
  (progn
    (push '|PatternMatchTools| *Packages*)
    (make-instance '|PatternMatchToolsType|)))
```

———-

1.74.30 Permanent

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |Permanent|
  (progn
    (push '|Permanent| *Packages*)
    (make-instance '|PermanentType|)))
```

———-

1.74.31 PermutationGroupExamples

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PermutationGroupExamples|
  (progn
    (push '|PermutationGroupExamples| *Packages*)
    (make-instance '|PermutationGroupExamplesType|)))
```

————-

1.74.32 PiCoercions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PiCoercions|
  (progn
    (push '|PiCoercions| *Packages*)
    (make-instance '|PiCoercionsType|)))
```

———-

1.74.33 PlotFunctions1

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlotFunctions1|
  (progn
    (push '|PlotFunctions1| *Packages*)
    (make-instance '|PlotFunctions1Type|)))
```

———-

1.74.34 PlotTools

— sane —

```
(defclass |PlotToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PlotTools")
   (marker :initform 'package)
   (abbreviation :initform 'PLOTTOOL)
   (comment :initform (list
     "This package exports plotting tools"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PlotTools|
  (progn
    (push '|PlotTools| *Packages*)
    (make-instance '|PlotToolsType|)))
```

———-

1.74.35 ProjectiveAlgebraicSetPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ProjectiveAlgebraicSetPackage|
  (progn
    (push '|ProjectiveAlgebraicSetPackage| *Packages*)
    (make-instance '|ProjectiveAlgebraicSetPackageType|)))
```

———-

1.74.36 PointFunctions2

— sane —

```
(defclass |PointFunctions2Type| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointFunctions2")
   (marker :initform 'package)
   (abbreviation :initform 'PTFUNC2)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointFunctions2|
  (progn
    (push '|PointFunctions2| *Packages*)
    (make-instance '|PointFunctions2Type|)))
```

———-

1.74.37 PointPackage

— sane —

```
(defclass |PointPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointPackage")
   (marker :initform 'package)
   (abbreviation :initform 'PTPACK)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointPackage|
  (progn
    (push '|PointPackage| *Packages*)
    (make-instance '|PointPackageType|)))
```

———-

1.74.38 PointsOfFiniteOrder

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointsOfFiniteOrder|
  (progn
    (push '|PointsOfFiniteOrder| *Packages*)
    (make-instance '|PointsOfFiniteOrderType|)))
```

———-

1.74.39 PointsOfFiniteOrderRational

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointsOfFiniteOrderRational|
  (progn
    (push '|PointsOfFiniteOrderRational| *Packages*)
    (make-instance '|PointsOfFiniteOrderRationalType|)))
```

———-

1.74.40 PointsOfFiniteOrderTools

— sane —

```
(defclass |PointsOfFiniteOrderToolsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PointsOfFiniteOrderTools")
   (marker :initform 'package)
   (abbreviation :initform 'PFOTOOLS)
   (comment :initform (list
     "Utilities for PFOQ and PFO"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PointsOfFiniteOrderTools|
  (progn
    (push '|PointsOfFiniteOrderTools| *Packages*)
    (make-instance '|PointsOfFiniteOrderToolsType|)))
```

———-

1.74.41 PolynomialPackageForCurve

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialPackageForCurve|
  (progn
    (push '|PolynomialPackageForCurve| *Packages*)
    (make-instance '|PolynomialPackageForCurveType|)))
```

———-

1.74.42 PolToPol

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolToPol|
  (progn
    (push '|PolToPol| *Packages*)
    (make-instance '|PolToPolType|)))
```

———-

1.74.43 PolyGroebner

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolyGroebner|
  (progn
    (push '|PolyGroebner| *Packages*)
    (make-instance '|PolyGroebnerType|)))
```

———-

1.74.44 PolynomialAN2Expression

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PolynomialAN2Expression|
  (progn
    (push '|PolynomialAN2Expression| *Packages*)
    (make-instance '|PolynomialAN2ExpressionType|)))
```

———-

1.74.45 PolynomialCategoryLifting

— sane —

```
(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 **."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialCategoryLifting|
  (progn
    (push '|PolynomialCategoryLifting| *Packages*)
    (make-instance '|PolynomialCategoryLiftingType|)))
```

———-

1.74.46 PolynomialCategoryQuotientFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |PolynomialCategoryQuotientFunctions|
  (progn
    (push '|PolynomialCategoryQuotientFunctions| *Packages*)
    (make-instance '|PolynomialCategoryQuotientFunctionsType|)))
```

———-

1.74.47 PolynomialComposition

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialComposition|
  (progn
    (push '|PolynomialComposition| *Packages*)
    (make-instance '|PolynomialCompositionType|)))
```

———-

1.74.48 PolynomialDecomposition

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |PolynomialDecomposition|
  (progn
    (push '|PolynomialDecomposition| *Packages*)
    (make-instance '|PolynomialDecompositionType|)))
```

———-

1.74.49 PolynomialFactorizationByRecursion

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFactorizationByRecursion|
  (progn
    (push '|PolynomialFactorizationByRecursion| *Packages*)
    (make-instance '|PolynomialFactorizationByRecursionType|)))
```

———-

1.74.50 PolynomialFactorizationByRecursionUnivariate

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFactorizationByRecursionUnivariate|
  (progn
    (push '|PolynomialFactorizationByRecursionUnivariate| *Packages*)
    (make-instance '|PolynomialFactorizationByRecursionUnivariateType|)))
```

———-

1.74.51 PolynomialFunctions2

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialFunctions2|
  (progn
    (push '|PolynomialFunctions2| *Packages*)
    (make-instance '|PolynomialFunctions2Type|)))
```

———-

1.74.52 PolynomialGcdPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialGcdPackage|
  (progn
    (push '|PolynomialGcdPackage| *Packages*)
    (make-instance '|PolynomialGcdPackageType|)))
```

———-

1.74.53 PolynomialInterpolation

— **sane** —

```
(defclass |PolynomialInterpolationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialInterpolation")
   (marker :initform 'package)
   (abbreviation :initform 'PINTERP)
   (comment :initform (list
     "This package exports interpolation algorithms"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialInterpolation|
  (progn
    (push '|PolynomialInterpolation| *Packages*)
    (make-instance '|PolynomialInterpolationType|)))
```

———-

1.74.54 PolynomialInterpolationAlgorithms

— **sane** —

```
(defclass |PolynomialInterpolationAlgorithmsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PolynomialInterpolationAlgorithms")
   (marker :initform 'package)
```

```
   (abbreviation :initform 'PINTERPA)
   (comment :initform (list
     "This package exports interpolation algorithms"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialInterpolationAlgorithms|
  (progn
    (push '|PolynomialInterpolationAlgorithms| *Packages*)
    (make-instance '|PolynomialInterpolationAlgorithmsType|)))
```

———-

1.74.55 PolynomialNumberTheoryFunctions

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialNumberTheoryFunctions|
  (progn
    (push '|PolynomialNumberTheoryFunctions| *Packages*)
    (make-instance '|PolynomialNumberTheoryFunctionsType|)))
```

———-

1.74.56 PolynomialRoots

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialRoots|
  (progn
    (push '|PolynomialRoots| *Packages*)
    (make-instance '|PolynomialRootsType|)))
```

———-

1.74.57 PolynomialSetUtilitiesPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialSetUtilitiesPackage|
  (progn
    (push '|PolynomialSetUtilitiesPackage| *Packages*)
    (make-instance '|PolynomialSetUtilitiesPackageType|)))
```

———-

1.74.58 PolynomialSolveByFormulas

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialSolveByFormulas|
  (progn
    (push '|PolynomialSolveByFormulas| *Packages*)
    (make-instance '|PolynomialSolveByFormulasType|)))
```

———-

1.74.59 PolynomialSquareFree

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialSquareFree|
  (progn
    (push '|PolynomialSquareFree| *Packages*)
    (make-instance '|PolynomialSquareFreeType|)))
```

———-

1.74.60 PolynomialToUnivariatePolynomial

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PolynomialToUnivariatePolynomial|
  (progn
    (push '|PolynomialToUnivariatePolynomial| *Packages*)
    (make-instance '|PolynomialToUnivariatePolynomialType|)))
```

———-

1.74.61 PowerSeriesLimitPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PowerSeriesLimitPackage|
  (progn
    (push '|PowerSeriesLimitPackage| *Packages*)
```

```
    (make-instance '|PowerSeriesLimitPackageType|)))
```

———-

1.74.62 PrecomputedAssociatedEquations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrecomputedAssociatedEquations|
  (progn
    (push '|PrecomputedAssociatedEquations| *Packages*)
    (make-instance '|PrecomputedAssociatedEquationsType|)))
```

———-

1.74.63 PrimitiveArrayFunctions2

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveArrayFunctions2|
  (progn
```

```
    (push '|PrimitiveArrayFunctions2| *Packages*)
    (make-instance '|PrimitiveArrayFunctions2Type|)))
```

———-

1.74.64 PrimitiveElement

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveElement|
  (progn
    (push '|PrimitiveElement| *Packages*)
    (make-instance '|PrimitiveElementType|)))
```

———-

1.74.65 PrimitiveRatDE

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveRatDE|
```

```
  (progn
    (push '|PrimitiveRatDE| *Packages*)
    (make-instance '|PrimitiveRatDEType|)))
```

———-

1.74.66 PrimitiveRatRicDE

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrimitiveRatRicDE|
  (progn
    (push '|PrimitiveRatRicDE| *Packages*)
    (make-instance '|PrimitiveRatRicDEType|)))
```

———-

1.74.67 PrintPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PrintPackage|
  (progn
    (push '|PrintPackage| *Packages*)
```

```
    (make-instance '|PrintPackageType|)))
```

———-

1.74.68 PseudoLinearNormalForm

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PseudoLinearNormalForm|
  (progn
    (push '|PseudoLinearNormalForm| *Packages*)
    (make-instance '|PseudoLinearNormalFormType|)))
```

———-

1.74.69 PseudoRemainderSequence

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |PseudoRemainderSequence|
  (progn
    (push '|PseudoRemainderSequence| *Packages*)
    (make-instance '|PseudoRemainderSequenceType|)))
```

———-

1.74.70 PureAlgebraicIntegration

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PureAlgebraicIntegration|
  (progn
    (push '|PureAlgebraicIntegration| *Packages*)
    (make-instance '|PureAlgebraicIntegrationType|)))
```

———-

1.74.71 PureAlgebraicLODE

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |PureAlgebraicLODE|
  (progn
    (push '|PureAlgebraicLODE| *Packages*)
    (make-instance '|PureAlgebraicLODEType|)))
```

———-

1.74.72 PushVariables

— sane —

```
(defclass |PushVariablesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "PushVariables")
   (marker :initform 'package)
   (abbreviation :initform 'PUSHVAR)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |PushVariables|
  (progn
    (push '|PushVariables| *Packages*)
    (make-instance '|PushVariablesType|)))
```

———-

1.75 Q

1.75.1 QuasiAlgebraicSet2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuasiAlgebraicSet2|
  (progn
    (push '|QuasiAlgebraicSet2| *Packages*)
    (make-instance '|QuasiAlgebraicSet2Type|)))
```

———-

1.75.2 QuasiComponentPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |QuasiComponentPackage|
  (progn
    (push '|QuasiComponentPackage| *Packages*)
    (make-instance '|QuasiComponentPackageType|)))
```

———-

1.75.3 QuotientFieldCategoryFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |QuotientFieldCategoryFunctions2|
  (progn
    (push '|QuotientFieldCategoryFunctions2| *Packages*)
    (make-instance '|QuotientFieldCategoryFunctions2Type|)))
```

———-

1.75.4 QuaternionCategoryFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |QuaternionCategoryFunctions2|
  (progn
    (push '|QuaternionCategoryFunctions2| *Packages*)
    (make-instance '|QuaternionCategoryFunctions2Type|)))
```

———-

1.76 R

1.76.1 RadicalEigenPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadicalEigenPackage|
  (progn
    (push '|RadicalEigenPackage| *Packages*)
    (make-instance '|RadicalEigenPackageType|)))
```

———-

1.76.2 RadicalSolvePackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadicalSolvePackage|
  (progn
    (push '|RadicalSolvePackage| *Packages*)
    (make-instance '|RadicalSolvePackageType|)))
```

———-

1.76.3 RadixUtilities

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RadixUtilities|
  (progn
    (push '|RadixUtilities| *Packages*)
    (make-instance '|RadixUtilitiesType|)))
```

———-

1.76.4 RandomDistributions

— **sane** —

```
(defclass |RandomDistributionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RandomDistributions")
   (marker :initform 'package)
   (abbreviation :initform 'RDIST)
   (comment :initform (list
     "This package exports random distributions"))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RandomDistributions|
  (progn
    (push '|RandomDistributions| *Packages*)
    (make-instance '|RandomDistributionsType|)))
```

———-

1.76.5 RandomFloatDistributions

— **sane** —

```
(defclass |RandomFloatDistributionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RandomFloatDistributions")
   (marker :initform 'package)
   (abbreviation :initform 'RFDIST)
   (comment :initform (list
     "This package exports random floating-point distributions"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RandomFloatDistributions|
  (progn
    (push '|RandomFloatDistributions| *Packages*)
    (make-instance '|RandomFloatDistributionsType|)))
```

———-

1.76.6 RandomIntegerDistributions

— **sane** —

```
(defclass |RandomIntegerDistributionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RandomIntegerDistributions")
   (marker :initform 'package)
   (abbreviation :initform 'RIDIST)
   (comment :initform (list
     "This package exports integer distributions"))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |RandomIntegerDistributions|
  (progn
    (push '|RandomIntegerDistributions| *Packages*)
    (make-instance '|RandomIntegerDistributionsType|)))
```

———-

1.76.7 RandomNumberSource

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RandomNumberSource|
  (progn
    (push '|RandomNumberSource| *Packages*)
    (make-instance '|RandomNumberSourceType|)))
```

———-

1.76.8 RationalFactorize

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFactorize|
  (progn
    (push '|RationalFactorize| *Packages*)
    (make-instance '|RationalFactorizeType|)))
```

———-

1.76.9 RationalFunction

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunction|
  (progn
    (push '|RationalFunction| *Packages*)
    (make-instance '|RationalFunctionType|)))
```

———-

1.76.10 RationalFunctionDefiniteIntegration

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionDefiniteIntegration|
  (progn
    (push '|RationalFunctionDefiniteIntegration| *Packages*)
    (make-instance '|RationalFunctionDefiniteIntegrationType|)))
```

———-

1.76.11 RationalFunctionFactor

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionFactor|
  (progn
    (push '|RationalFunctionFactor| *Packages*)
    (make-instance '|RationalFunctionFactorType|)))
```

———-

1.76.12 RationalFunctionFactorizer

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionFactorizer|
  (progn
    (push '|RationalFunctionFactorizer| *Packages*)
    (make-instance '|RationalFunctionFactorizerType|)))
```

———-

1.76.13 RationalFunctionIntegration

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionIntegration|
  (progn
    (push '|RationalFunctionIntegration| *Packages*)
    (make-instance '|RationalFunctionIntegrationType|)))
```

———-

1.76.14 RationalFunctionLimitPackage

— **sane** —

```
(defclass |RationalFunctionLimitPackageType| (|AxiomClass|)
```

```
  ((parents :initform ())
   (name :initform "RationalFunctionLimitPackage")
   (marker :initform 'package)
   (abbreviation :initform 'LIMITRF)
   (comment :initform (list
     "Computation of limits for rational functions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionLimitPackage|
  (progn
    (push '|RationalFunctionLimitPackage| *Packages*)
    (make-instance '|RationalFunctionLimitPackageType|)))
```

———-

1.76.15 RationalFunctionSign

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionSign|
  (progn
    (push '|RationalFunctionSign| *Packages*)
    (make-instance '|RationalFunctionSignType|)))
```

———-

1.76.16 RationalFunctionSum

— **sane** —

```
(defclass |RationalFunctionSumType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalFunctionSum")
```

```
   (marker :initform 'package)
   (abbreviation :initform 'SUMRF)
   (comment :initform (list
     "Computes sums of rational functions"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalFunctionSum|
  (progn
    (push '|RationalFunctionSum| *Packages*)
    (make-instance '|RationalFunctionSumType|)))
```

———-

1.76.17 RationalIntegration

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalIntegration|
  (progn
    (push '|RationalIntegration| *Packages*)
    (make-instance '|RationalIntegrationType|)))
```

———-

1.76.18 RationalInterpolation

— **sane** —

```
(defclass |RationalInterpolationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalInterpolation")
   (marker :initform 'package)
   (abbreviation :initform 'RINTERP)
```

```
   (comment :initform (list
     "This package exports rational interpolation algorithms"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalInterpolation|
  (progn
    (push '|RationalInterpolation| *Packages*)
    (make-instance '|RationalInterpolationType|)))
```

———-

1.76.19 RationalLODE

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalLODE|
  (progn
    (push '|RationalLODE| *Packages*)
    (make-instance '|RationalLODEType|)))
```

———-

1.76.20 RationalRetractions

— sane —

```
(defclass |RationalRetractionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RationalRetractions")
   (marker :initform 'package)
   (abbreviation :initform 'RATRET)
   (comment :initform (list
```

```
     "Rational number testing and retraction functions."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalRetractions|
  (progn
    (push '|RationalRetractions| *Packages*)
    (make-instance '|RationalRetractionsType|)))
```

———-

1.76.21 RationalRicDE

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalRicDE|
  (progn
    (push '|RationalRicDE| *Packages*)
    (make-instance '|RationalRicDEType|)))
```

———-

1.76.22 RationalUnivariateRepresentationPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RationalUnivariateRepresentationPackage|
  (progn
    (push '|RationalUnivariateRepresentationPackage| *Packages*)
    (make-instance '|RationalUnivariateRepresentationPackageType|)))
```

———-

1.76.23 RealPolynomialUtilitiesPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealPolynomialUtilitiesPackage|
  (progn
    (push '|RealPolynomialUtilitiesPackage| *Packages*)
    (make-instance '|RealPolynomialUtilitiesPackageType|)))
```

———-

1.76.24 RealSolvePackage

— **sane** —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealSolvePackage|
  (progn
    (push '|RealSolvePackage| *Packages*)
    (make-instance '|RealSolvePackageType|)))
```

———-

1.76.25 RealZeroPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealZeroPackage|
  (progn
    (push '|RealZeroPackage| *Packages*)
    (make-instance '|RealZeroPackageType|)))
```

———-

1.76.26 RealZeroPackageQ

— **sane** —

```
(defclass |RealZeroPackageQType| (|AxiomClass|)
```

```
  ((parents :initform ())
   (name :initform "RealZeroPackageQ")
   (marker :initform 'package)
   (abbreviation :initform 'REALOQ)
   (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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RealZeroPackageQ|
  (progn
    (push '|RealZeroPackageQ| *Packages*)
    (make-instance '|RealZeroPackageQType|)))
```

———-

1.76.27 RectangularMatrixCategoryFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RectangularMatrixCategoryFunctions2|
  (progn
    (push '|RectangularMatrixCategoryFunctions2| *Packages*)
    (make-instance '|RectangularMatrixCategoryFunctions2Type|)))
```

———-

1.76.28 RecurrenceOperator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RecurrenceOperator|
  (progn
    (push '|RecurrenceOperator| *Packages*)
    (make-instance '|RecurrenceOperatorType|)))
```

———-

1.76.29 ReducedDivisor

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ReducedDivisor|
  (progn
    (push '|ReducedDivisor| *Packages*)
    (make-instance '|ReducedDivisorType|)))
```

———-

1.76.30 ReduceLODE

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ReduceLODE|
  (progn
    (push '|ReduceLODE| *Packages*)
    (make-instance '|ReduceLODEType|)))
```

———-

1.76.31 ReductionOfOrder

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ReductionOfOrder|
  (progn
    (push '|ReductionOfOrder| *Packages*)
    (make-instance '|ReductionOfOrderType|)))
```

———-

1.76.32 RegularSetDecompositionPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularSetDecompositionPackage|
  (progn
    (push '|RegularSetDecompositionPackage| *Packages*)
    (make-instance '|RegularSetDecompositionPackageType|)))
```

———-

1.76.33 RegularTriangularSetGcdPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RegularTriangularSetGcdPackage|
  (progn
    (push '|RegularTriangularSetGcdPackage| *Packages*)
    (make-instance '|RegularTriangularSetGcdPackageType|)))
```

———-

1.76.34 RepeatedDoubling

— **sane** —

```
(defclass |RepeatedDoublingType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RepeatedDoubling")
   (marker :initform 'package)
   (abbreviation :initform 'REPDB)
   (comment :initform (list
     "Implements multiplication by repeated addition"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepeatedDoubling|
  (progn
    (push '|RepeatedDoubling| *Packages*)
    (make-instance '|RepeatedDoublingType|)))
```

———-

1.76.35 RepeatedSquaring

— **sane** —

```
(defclass |RepeatedSquaringType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "RepeatedSquaring")
   (marker :initform 'package)
   (abbreviation :initform 'REPSQ)
   (comment :initform (list
```

```
     "Implements exponentiation by repeated squaring"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepeatedSquaring|
  (progn
    (push '|RepeatedSquaring| *Packages*)
    (make-instance '|RepeatedSquaringType|)))
```

———-

1.76.36 RepresentationPackage1

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepresentationPackage1|
  (progn
    (push '|RepresentationPackage1| *Packages*)
    (make-instance '|RepresentationPackage1Type|)))
```

———-

1.76.37 RepresentationPackage2

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RepresentationPackage2|
  (progn
    (push '|RepresentationPackage2| *Packages*)
    (make-instance '|RepresentationPackage2Type|)))
```

———-

1.76.38 ResolveLatticeCompletion

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ResolveLatticeCompletion|
  (progn
    (push '|ResolveLatticeCompletion| *Packages*)
    (make-instance '|ResolveLatticeCompletionType|)))
```

———-

1.76.39 RetractSolvePackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |RetractSolvePackage|
  (progn
    (push '|RetractSolvePackage| *Packages*)
    (make-instance '|RetractSolvePackageType|)))
```

———-

1.76.40 RootsFindingPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |RootsFindingPackage|
  (progn
    (push '|RootsFindingPackage| *Packages*)
    (make-instance '|RootsFindingPackageType|)))
```

———-

1.77 S

1.77.1 SAERationalFunctionAlgFactor

— sane —

```
(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}."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SAERationalFunctionAlgFactor|
  (progn
    (push '|SAERationalFunctionAlgFactor| *Packages*)
    (make-instance '|SAERationalFunctionAlgFactorType|)))
```

———-

1.77.2 ScriptFormulaFormat1

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |ScriptFormulaFormat1|
  (progn
    (push '|ScriptFormulaFormat1| *Packages*)
    (make-instance '|ScriptFormulaFormat1Type|)))
```

———-

1.77.3 SegmentBindingFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SegmentBindingFunctions2|
  (progn
    (push '|SegmentBindingFunctions2| *Packages*)
    (make-instance '|SegmentBindingFunctions2Type|)))
```

———-

1.77.4 SegmentFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |SegmentFunctions2|
  (progn
    (push '|SegmentFunctions2| *Packages*)
    (make-instance '|SegmentFunctions2Type|)))
```

———-

1.77.5 SimpleAlgebraicExtensionAlgFactor

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SimpleAlgebraicExtensionAlgFactor|
  (progn
    (push '|SimpleAlgebraicExtensionAlgFactor| *Packages*)
    (make-instance '|SimpleAlgebraicExtensionAlgFactorType|)))
```

———-

1.77.6 SimplifyAlgebraicNumberConvertPackage

— sane —

```
(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)"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |SimplifyAlgebraicNumberConvertPackage|
  (progn
    (push '|SimplifyAlgebraicNumberConvertPackage| *Packages*)
    (make-instance '|SimplifyAlgebraicNumberConvertPackageType|)))
```

———-

1.77.7 SmithNormalForm

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SmithNormalForm|
  (progn
    (push '|SmithNormalForm| *Packages*)
    (make-instance '|SmithNormalFormType|)))
```

———-

1.77.8 SortedCache

— sane —

```
(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."))
```

```
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SortedCache|
  (progn
    (push '|SortedCache| *Packages*)
    (make-instance '|SortedCacheType|)))
```

———-

1.77.9 SortPackage

— sane —

```
(defclass |SortPackageType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "SortPackage")
   (marker :initform 'package)
   (abbreviation :initform 'SORTPAK)
   (comment :initform (list
     "This package exports sorting algorithnms"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SortPackage|
  (progn
    (push '|SortPackage| *Packages*)
    (make-instance '|SortPackageType|)))
```

———-

1.77.10 SparseUnivariatePolynomialFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SparseUnivariatePolynomialFunctions2|
  (progn
    (push '|SparseUnivariatePolynomialFunctions2| *Packages*)
    (make-instance '|SparseUnivariatePolynomialFunctions2Type|)))
```

———-

1.77.11 SpecialOutputPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SpecialOutputPackage|
  (progn
    (push '|SpecialOutputPackage| *Packages*)
    (make-instance '|SpecialOutputPackageType|)))
```

———-

1.77.12 SquareFreeQuasiComponentPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeQuasiComponentPackage|
  (progn
    (push '|SquareFreeQuasiComponentPackage| *Packages*)
    (make-instance '|SquareFreeQuasiComponentPackageType|)))
```

———-

1.77.13 SquareFreeRegularSetDecompositionPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularSetDecompositionPackage|
```

```
  (progn
    (push '|SquareFreeRegularSetDecompositionPackage| *Packages*)
    (make-instance '|SquareFreeRegularSetDecompositionPackageType|)))
```

———-

1.77.14 SquareFreeRegularTriangularSetGcdPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SquareFreeRegularTriangularSetGcdPackage|
  (progn
    (push '|SquareFreeRegularTriangularSetGcdPackage| *Packages*)
    (make-instance '|SquareFreeRegularTriangularSetGcdPackageType|)))
```

———-

1.77.15 StorageEfficientMatrixOperations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |StorageEfficientMatrixOperations|
  (progn
    (push '|StorageEfficientMatrixOperations| *Packages*)
    (make-instance '|StorageEfficientMatrixOperationsType|)))
```

———-

1.77.16 StreamFunctions1

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamFunctions1|
  (progn
    (push '|StreamFunctions1| *Packages*)
    (make-instance '|StreamFunctions1Type|)))
```

———-

1.77.17 StreamFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |StreamFunctions2|
  (progn
    (push '|StreamFunctions2| *Packages*)
    (make-instance '|StreamFunctions2Type|)))
```

———-

1.77.18 StreamFunctions3

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamFunctions3|
  (progn
    (push '|StreamFunctions3| *Packages*)
    (make-instance '|StreamFunctions3Type|)))
```

———-

1.77.19 StreamInfiniteProduct

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
```

```
   (addlist :initform nil)))

(defvar |StreamInfiniteProduct|
  (progn
    (push '|StreamInfiniteProduct| *Packages*)
    (make-instance '|StreamInfiniteProductType|)))
```

———-

1.77.20 StreamTaylorSeriesOperations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamTaylorSeriesOperations|
  (progn
    (push '|StreamTaylorSeriesOperations| *Packages*)
    (make-instance '|StreamTaylorSeriesOperationsType|)))
```

———-

1.77.21 StreamTensor

— sane —

```
(defclass |StreamTensorType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "StreamTensor")
   (marker :initform 'package)
   (abbreviation :initform 'STNSR)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |StreamTensor|
  (progn
    (push '|StreamTensor| *Packages*)
    (make-instance '|StreamTensorType|)))
```

———-

1.77.22 StreamTranscendentalFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StreamTranscendentalFunctions|
  (progn
    (push '|StreamTranscendentalFunctions| *Packages*)
    (make-instance '|StreamTranscendentalFunctionsType|)))
```

———-

1.77.23 StreamTranscendentalFunctionsNonCommutative

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
```

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

(defvar |StreamTranscendentalFunctionsNonCommutative|
  (progn
    (push '|StreamTranscendentalFunctionsNonCommutative| *Packages*)
    (make-instance '|StreamTranscendentalFunctionsNonCommutativeType|)))
```

———-

1.77.24 StructuralConstantsPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |StructuralConstantsPackage|
  (progn
    (push '|StructuralConstantsPackage| *Packages*)
    (make-instance '|StructuralConstantsPackageType|)))
```

———-

1.77.25 SturmHabichtPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
```

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

(defvar |SturmHabichtPackage|
  (progn
    (push '|SturmHabichtPackage| *Packages*)
    (make-instance '|SturmHabichtPackageType|)))
```

———-

1.77.26 SubResultantPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SubResultantPackage|
  (progn
    (push '|SubResultantPackage| *Packages*)
    (make-instance '|SubResultantPackageType|)))
```

———-

1.77.27 SupFractionFactorizer

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SupFractionFactorizer|
  (progn
    (push '|SupFractionFactorizer| *Packages*)
    (make-instance '|SupFractionFactorizerType|)))
```

———-

1.77.28 SystemODESolver

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SystemODESolver|
  (progn
    (push '|SystemODESolver| *Packages*)
    (make-instance '|SystemODESolverType|)))
```

———-

1.77.29 SystemSolvePackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SystemSolvePackage|
  (progn
    (push '|SystemSolvePackage| *Packages*)
    (make-instance '|SystemSolvePackageType|)))
```

———-

1.77.30 SymmetricGroupCombinatoricFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SymmetricGroupCombinatoricFunctions|
  (progn
    (push '|SymmetricGroupCombinatoricFunctions| *Packages*)
    (make-instance '|SymmetricGroupCombinatoricFunctionsType|)))
```

———-

1.77.31 SymmetricFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |SymmetricFunctions|
  (progn
    (push '|SymmetricFunctions| *Packages*)
    (make-instance '|SymmetricFunctionsType|)))
```

———-

1.78 T

1.78.1 TableauxBumpers

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TableauxBumpers|
  (progn
    (push '|TableauxBumpers| *Packages*)
    (make-instance '|TableauxBumpersType|)))
```

———-

1.78.2 TabulatedComputationPackage

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TabulatedComputationPackage|
  (progn
    (push '|TabulatedComputationPackage| *Packages*)
    (make-instance '|TabulatedComputationPackageType|)))
```

———-

1.78.3 TangentExpansions

— sane —

```
(defclass |TangentExpansionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TangentExpansions")
   (marker :initform 'package)
   (abbreviation :initform 'TANEXP)
   (comment :initform (list
     "Expands tangents of sums and scalar products."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TangentExpansions|
```

```
  (progn
    (push '|TangentExpansions| *Packages*)
    (make-instance '|TangentExpansionsType|)))
```

———-

1.78.4 TaylorSolve

— sane —

```
(defclass |TaylorSolveType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TaylorSolve")
   (marker :initform 'package)
   (abbreviation :initform 'UTSSOL)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TaylorSolve|
  (progn
    (push '|TaylorSolve| *Packages*)
    (make-instance '|TaylorSolveType|)))
```

———-

1.78.5 TemplateUtilities

— sane —

```
(defclass |TemplateUtilitiesType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TemplateUtilities")
   (marker :initform 'package)
   (abbreviation :initform 'TEMUTL)
   (comment :initform (list
     "This package provides functions for template manipulation"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TemplateUtilities|
  (progn
    (push '|TemplateUtilities| *Packages*)
```

```
    (make-instance '|TemplateUtilitiesType|)))
```

———-

1.78.6 TexFormat1

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TexFormat1|
  (progn
    (push '|TexFormat1| *Packages*)
    (make-instance '|TexFormat1Type|)))
```

———-

1.78.7 ToolsForSign

— sane —

```
(defclass |ToolsForSignType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "ToolsForSign")
   (marker :initform 'package)
   (abbreviation :initform 'TOOLSIGN)
   (comment :initform (list
     "Tools for the sign finding utilities."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ToolsForSign|
  (progn
    (push '|ToolsForSign| *Packages*)
    (make-instance '|ToolsForSignType|)))
```

———-

1.78.8 TopLevelDrawFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctions|
  (progn
    (push '|TopLevelDrawFunctions| *Packages*)
    (make-instance '|TopLevelDrawFunctionsType|)))
```

———-

1.78.9 TopLevelDrawFunctionsForAlgebraicCurves

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctionsForAlgebraicCurves|
  (progn
    (push '|TopLevelDrawFunctionsForAlgebraicCurves| *Packages*)
    (make-instance '|TopLevelDrawFunctionsForAlgebraicCurvesType|)))
```

———-

1.78.10 TopLevelDrawFunctionsForCompiledFunctions

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctionsForCompiledFunctions|
  (progn
    (push '|TopLevelDrawFunctionsForCompiledFunctions| *Packages*)
    (make-instance '|TopLevelDrawFunctionsForCompiledFunctionsType|)))
```

———-

1.78.11 TopLevelDrawFunctionsForPoints

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelDrawFunctionsForPoints|
  (progn
    (push '|TopLevelDrawFunctionsForPoints| *Packages*)
    (make-instance '|TopLevelDrawFunctionsForPointsType|)))
```

———-

1.78.12 TopLevelThreeSpace

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TopLevelThreeSpace|
  (progn
    (push '|TopLevelThreeSpace| *Packages*)
    (make-instance '|TopLevelThreeSpaceType|)))
```

———-

1.78.13 TranscendentalHermiteIntegration

— sane —

```
(defclass |TranscendentalHermiteIntegrationType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalHermiteIntegration")
   (marker :initform 'package)
   (abbreviation :initform 'INTHERTR)
   (comment :initform (list
     "Hermite integration, transcendental case."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalHermiteIntegration|
  (progn
    (push '|TranscendentalHermiteIntegration| *Packages*)
    (make-instance '|TranscendentalHermiteIntegrationType|)))
```

———-

1.78.14 TranscendentalIntegration

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalIntegration|
  (progn
    (push '|TranscendentalIntegration| *Packages*)
    (make-instance '|TranscendentalIntegrationType|)))
```

———-

1.78.15 TranscendentalManipulations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalManipulations|
  (progn
    (push '|TranscendentalManipulations| *Packages*)
    (make-instance '|TranscendentalManipulationsType|)))
```

———-

1.78.16 TranscendentalRischDE

— sane —

```
(defclass |TranscendentalRischDEType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalRischDE")
   (marker :initform 'package)
   (abbreviation :initform 'RDETR)
   (comment :initform (list
     "Risch differential equation, transcendental case."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalRischDE|
  (progn
    (push '|TranscendentalRischDE| *Packages*)
    (make-instance '|TranscendentalRischDEType|)))
```

———-

1.78.17 TranscendentalRischDESystem

— sane —

```
(defclass |TranscendentalRischDESystemType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "TranscendentalRischDESystem")
   (marker :initform 'package)
   (abbreviation :initform 'RDETRS)
   (comment :initform (list
     "Risch differential equation system, transcendental case."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TranscendentalRischDESystem|
  (progn
    (push '|TranscendentalRischDESystem| *Packages*)
    (make-instance '|TranscendentalRischDESystemType|)))
```

———-

1.78.18 TransSolvePackage

— sane —

```
(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 ."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TransSolvePackage|
  (progn
    (push '|TransSolvePackage| *Packages*)
    (make-instance '|TransSolvePackageType|)))
```

———-

1.78.19 TransSolvePackageService

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TransSolvePackageService|
  (progn
    (push '|TransSolvePackageService| *Packages*)
    (make-instance '|TransSolvePackageServiceType|)))
```

———-

1.78.20 TriangularMatrixOperations

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TriangularMatrixOperations|
  (progn
    (push '|TriangularMatrixOperations| *Packages*)
    (make-instance '|TriangularMatrixOperationsType|)))
```

———-

1.78.21 TrigonometricManipulations

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TrigonometricManipulations|
  (progn
    (push '|TrigonometricManipulations| *Packages*)
    (make-instance '|TrigonometricManipulationsType|)))
```

———-

1.78.22 TubePlotTools

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TubePlotTools|
  (progn
    (push '|TubePlotTools| *Packages*)
    (make-instance '|TubePlotToolsType|)))
```

———-

1.78.23 TwoDimensionalPlotClipping

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoDimensionalPlotClipping|
  (progn
    (push '|TwoDimensionalPlotClipping| *Packages*)
    (make-instance '|TwoDimensionalPlotClippingType|)))
```

———-

1.78.24 TwoFactorize

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |TwoFactorize|
  (progn
    (push '|TwoFactorize| *Packages*)
    (make-instance '|TwoFactorizeType|)))
```

———-

1.79 U

1.79.1 UnivariateFactorize

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateFactorize|
  (progn
    (push '|UnivariateFactorize| *Packages*)
    (make-instance '|UnivariateFactorizeType|)))
```

———-

1.79.2 UnivariateFormalPowerSeriesFunctions

— sane —

```
(defclass |UnivariateFormalPowerSeriesFunctionsType| (|AxiomClass|)
  ((parents :initform ())
   (name :initform "UnivariateFormalPowerSeriesFunctions")
   (marker :initform 'package)
   (abbreviation :initform 'UFPS1)
   (comment :initform (list
     "This package has no description"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateFormalPowerSeriesFunctions|
  (progn
    (push '|UnivariateFormalPowerSeriesFunctions| *Packages*)
    (make-instance '|UnivariateFormalPowerSeriesFunctionsType|)))
```

———-

1.79.3 UnivariateLaurentSeriesFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateLaurentSeriesFunctions2|
  (progn
    (push '|UnivariateLaurentSeriesFunctions2| *Packages*)
    (make-instance '|UnivariateLaurentSeriesFunctions2Type|)))
```

———-

1.79.4 UnivariatePolynomialCategoryFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialCategoryFunctions2|
  (progn
    (push '|UnivariatePolynomialCategoryFunctions2| *Packages*)
    (make-instance '|UnivariatePolynomialCategoryFunctions2Type|)))
```

———-

1.79.5 UnivariatePolynomialCommonDenominator

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialCommonDenominator|
  (progn
    (push '|UnivariatePolynomialCommonDenominator| *Packages*)
    (make-instance '|UnivariatePolynomialCommonDenominatorType|)))
```

———-

1.79.6 UnivariatePolynomialDecompositionPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialDecompositionPackage|
  (progn
    (push '|UnivariatePolynomialDecompositionPackage| *Packages*)
    (make-instance '|UnivariatePolynomialDecompositionPackageType|)))
```

———-

1.79.7 UnivariatePolynomialDivisionPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialDivisionPackage|
  (progn
    (push '|UnivariatePolynomialDivisionPackage| *Packages*)
    (make-instance '|UnivariatePolynomialDivisionPackageType|)))
```

———-

1.79.8 UnivariatePolynomialFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))
```

```
(defvar |UnivariatePolynomialFunctions2|
  (progn
    (push '|UnivariatePolynomialFunctions2| *Packages*)
    (make-instance '|UnivariatePolynomialFunctions2Type|)))
```

———-

1.79.9 UnivariatePolynomialMultiplicationPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialMultiplicationPackage|
  (progn
    (push '|UnivariatePolynomialMultiplicationPackage| *Packages*)
    (make-instance '|UnivariatePolynomialMultiplicationPackageType|)))
```

———-

1.79.10 UnivariatePolynomialSquareFree

— sane —

```
(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)."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePolynomialSquareFree|
  (progn
    (push '|UnivariatePolynomialSquareFree| *Packages*)
    (make-instance '|UnivariatePolynomialSquareFreeType|)))
```

———-

1.79.11 UnivariatePuiseuxSeriesFunctions2

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariatePuiseuxSeriesFunctions2|
  (progn
    (push '|UnivariatePuiseuxSeriesFunctions2| *Packages*)
    (make-instance '|UnivariatePuiseuxSeriesFunctions2Type|)))
```

———-

1.79.12 UnivariateSkewPolynomialCategoryOps

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateSkewPolynomialCategoryOps|
  (progn
    (push '|UnivariateSkewPolynomialCategoryOps| *Packages*)
    (make-instance '|UnivariateSkewPolynomialCategoryOpsType|)))
```

———-

1.79.13 UnivariateTaylorSeriesFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesFunctions2|
  (progn
    (push '|UnivariateTaylorSeriesFunctions2| *Packages*)
    (make-instance '|UnivariateTaylorSeriesFunctions2Type|)))
```

———-

1.79.14 UnivariateTaylorSeriesODESolver

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UnivariateTaylorSeriesODESolver|
  (progn
    (push '|UnivariateTaylorSeriesODESolver| *Packages*)
    (make-instance '|UnivariateTaylorSeriesODESolverType|)))
```

———-

1.79.15 UniversalSegmentFunctions2

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UniversalSegmentFunctions2|
  (progn
    (push '|UniversalSegmentFunctions2| *Packages*)
    (make-instance '|UniversalSegmentFunctions2Type|)))
```

———-

1.79.16 UserDefinedPartialOrdering

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UserDefinedPartialOrdering|
  (progn
    (push '|UserDefinedPartialOrdering| *Packages*)
    (make-instance '|UserDefinedPartialOrderingType|)))
```

———-

1.79.17 UserDefinedVariableOrdering

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UserDefinedVariableOrdering|
  (progn
    (push '|UserDefinedVariableOrdering| *Packages*)
    (make-instance '|UserDefinedVariableOrderingType|)))
```

———-

1.79.18 UTSodetools

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |UTSodetools|
  (progn
    (push '|UTSodetools| *Packages*)
    (make-instance '|UTSodetoolsType|)))
```

———-

1.79.19 U32VectorPolynomialOperations

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |U32VectorPolynomialOperations|
  (progn
    (push '|U32VectorPolynomialOperations| *Packages*)
    (make-instance '|U32VectorPolynomialOperationsType|)))
```

———-

1.80 V

1.80.1 VectorFunctions2

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |VectorFunctions2|
  (progn
    (push '|VectorFunctions2| *Packages*)
    (make-instance '|VectorFunctions2Type|)))
```

———-

1.80.2 ViewDefaultsPackage

— sane —

```
(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"))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ViewDefaultsPackage|
  (progn
    (push '|ViewDefaultsPackage| *Packages*)
    (make-instance '|ViewDefaultsPackageType|)))
```

———-

1.80.3 ViewportPackage

— sane —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |ViewportPackage|
  (progn
    (push '|ViewportPackage| *Packages*)
    (make-instance '|ViewportPackageType|)))
```

———-

1.81 W

1.81.1 WeierstrassPreparation

— sane —

```
(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"))
   (arglist :initform nil)
```

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

(defvar |WeierstrassPreparation|
  (progn
    (push '|WeierstrassPreparation| *Packages*)
    (make-instance '|WeierstrassPreparationType|)))
```

———-

1.81.2 WildFunctionFieldIntegralBasis

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |WildFunctionFieldIntegralBasis|
  (progn
    (push '|WildFunctionFieldIntegralBasis| *Packages*)
    (make-instance '|WildFunctionFieldIntegralBasisType|)))
```

———-

1.82 X

1.82.1 XExponentialPackage

— **sane** —

```
(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."))
   (arglist :initform nil)
   (macros :initform nil)
   (withlist :initform nil)
   (haslist :initform nil)
   (addlist :initform nil)))

(defvar |XExponentialPackage|
  (progn
    (push '|XExponentialPackage| *Packages*)
    (make-instance '|XExponentialPackageType|)))
```

———-

1.83 Z

1.83.1 ZeroDimensionalSolvePackage

— sane —

```
(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."))
   (arglist :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|)))
```

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

1.84 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** —

```
(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))))
```

———-

— **sane** —

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

———-

Sane Interpreter

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.

— **insane** —

```
(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)))))
```

———-

— **insane** —

```
(defclass |NonNegativeIntegerType| (|MonoidType| |OrderedAbelianMonoidSupType|)
  ((ancestors :initform '(|OrderedAbelianMonoidSup| |Monoid|))
   (abbreviation :initform 'NNI)))

(defvar |NonNegativeInteger|
  (progn
    (push '|NonNegativeInteger| *Domains*)
    (make-instance '|NonNegativeIntegerType|)))
```

———-

— **insane** —

```
(defclass |ApplicationProgramInterfaceType| (|AxiomClass)
  ((ancestors :initform ())
   (abbreviation :initform 'API)))

(defvar |ApplicationProgramInterface|
  (progn
    (push '|ApplicationProgramInterface| *Domains*)
    (make-instance '|ApplicationProgramInterfaceType|)))
```

———-

Bibliography

[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

[Brui68a] N.G. de Bruijn. AUTOMATH, A Language for Mathematics. technical report 68-WSK-05, Technische Hogeschool Eindhoven, 1968.

[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

[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.

[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

[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.pd

[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.

[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.

[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

[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

[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. Howewer, 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.

[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.

[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:

(p510) package DFSFUN DoubleFloatSpecialFunctions

Index