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 2: Axiom Users Guide
This book and the Axiom software is licensed as follows:

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of The Numerical Algorithms Group Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.
<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael Albaugh</td>
<td>Cyril Alberga</td>
<td>Roy Adler</td>
</tr>
<tr>
<td>Christian Aistleitner</td>
<td>Richard Anderson</td>
<td>George Andrews</td>
</tr>
<tr>
<td>S.J. Atkins</td>
<td>Jeremy Avigad</td>
<td>Henry Baker</td>
</tr>
<tr>
<td>Martin Baker</td>
<td>Stephen Balzac</td>
<td>Yurij Baransky</td>
</tr>
<tr>
<td>David R. Barton</td>
<td>Thomas Baruchel</td>
<td>Gerald Baumgartner</td>
</tr>
<tr>
<td>Gilbert Baumslag</td>
<td>Michael Becker</td>
<td>Nelson H. F. Beebe</td>
</tr>
<tr>
<td>Jay Belanger</td>
<td>David Bindel</td>
<td>Fred Blair</td>
</tr>
<tr>
<td>Vladimir Bondarenko</td>
<td>Mark Botch</td>
<td>Raoul Bourquin</td>
</tr>
<tr>
<td>Alexandre Bouyer</td>
<td>Karen Braman</td>
<td>Wolfgang Brehm</td>
</tr>
<tr>
<td>Peter A. Broadbery</td>
<td>Martin Brock</td>
<td>Manuel Bronstein</td>
</tr>
<tr>
<td>Christopher Brown</td>
<td>Stephen Buchwald</td>
<td>Florian Bundschuh</td>
</tr>
<tr>
<td>Luanne Burns</td>
<td>William Burge</td>
<td>Ralph Byers</td>
</tr>
<tr>
<td>Quentin Carpent</td>
<td>Pierre Casteran</td>
<td>Robert Cavines</td>
</tr>
<tr>
<td>Bruce Char</td>
<td>Ondrej Certik</td>
<td>Tzu-Yi Chen</td>
</tr>
<tr>
<td>Bobby Cheng</td>
<td>Cheekai Chin</td>
<td>David V. Chudnovsky</td>
</tr>
<tr>
<td>Gregory V. Chudnovsky</td>
<td>Mark Clements</td>
<td>James Cloos</td>
</tr>
<tr>
<td>Jia Zhao Cong</td>
<td>Josh Cohen</td>
<td>Christophe Conil</td>
</tr>
<tr>
<td>Don Coppersmith</td>
<td>George Corliss</td>
<td>Robert Corless</td>
</tr>
<tr>
<td>Gary Cornell</td>
<td>Meino Cramer</td>
<td>Jeremy Du Croz</td>
</tr>
<tr>
<td>David Cyganski</td>
<td>Nathaniel Daly</td>
<td>Timothy Daly Sr.</td>
</tr>
<tr>
<td>Timothy Daly Jr.</td>
<td>James H. Davenport</td>
<td>David Day</td>
</tr>
<tr>
<td>James Demmel</td>
<td>Didier Deshommes</td>
<td>Michael Dewar</td>
</tr>
<tr>
<td>Inderjit Dhillon</td>
<td>Jack Dongarra</td>
<td>Jean Della Dora</td>
</tr>
<tr>
<td>Gabriel Dos Reis</td>
<td>Claire DiCrescendo</td>
<td>Sam Dooley</td>
</tr>
<tr>
<td>Zlatko Drmac</td>
<td>Lionel Ducos</td>
<td>Iain Duff</td>
</tr>
<tr>
<td>Lee Duhem</td>
<td>Martin Dunstan</td>
<td>Brian Dupee</td>
</tr>
<tr>
<td>Dominique Duval</td>
<td>Robert Edwards</td>
<td>Heo Eide-Goodman</td>
</tr>
<tr>
<td>Lars Erickson</td>
<td>Mark Fahey</td>
<td>Richard Fateman</td>
</tr>
<tr>
<td>Bertfried Fauser</td>
<td>Stuart Feldman</td>
<td>John Fletcher</td>
</tr>
<tr>
<td>Brian Ford</td>
<td>Albrecht Fortenbacher</td>
<td>George Frances</td>
</tr>
<tr>
<td>Constantine Frangos</td>
<td>Timothy Freeman</td>
<td>Korrinn Fu</td>
</tr>
<tr>
<td>Marc Gaetano</td>
<td>Rudiger Gebauer</td>
<td>Van de Geijn</td>
</tr>
<tr>
<td>Kathy Gerber</td>
<td>Patricia Gianni</td>
<td>Gustavo Goertkin</td>
</tr>
<tr>
<td>Samantha Goldrich</td>
<td>Holger Gollan</td>
<td>Teresa Gomez-Diaz</td>
</tr>
<tr>
<td>Laureano Gonzalez-Vega</td>
<td>Stephen Gortler</td>
<td>Johannes Grabmeier</td>
</tr>
<tr>
<td>Matt Grayson</td>
<td>Klaus Ebbe Grue</td>
<td>James Griesmer</td>
</tr>
<tr>
<td>Vladimir Grinberg</td>
<td>Oswald Gschneider</td>
<td>Ming Gu</td>
</tr>
<tr>
<td>Jocelyn Guidry</td>
<td>Gaetan Hache</td>
<td>Steve Hague</td>
</tr>
<tr>
<td>Satoshi Hamaguchi</td>
<td>Sven Hammarling</td>
<td>Mike Hansen</td>
</tr>
<tr>
<td>Richard Hanson</td>
<td>Richard Harke</td>
<td>Bill Hart</td>
</tr>
<tr>
<td>Vilya Harvey</td>
<td>Martin Hassner</td>
<td>Arthur S. Hathaway</td>
</tr>
<tr>
<td>Dan Hatton</td>
<td>Waldek Hebisch</td>
<td>Karl Hegboom</td>
</tr>
<tr>
<td>Ralf Hemmecke</td>
<td>Henderson</td>
<td>Antoine Hersen</td>
</tr>
<tr>
<td>Nicholas J. Higham</td>
<td>Hoon Hong</td>
<td>Roger House</td>
</tr>
<tr>
<td>Gernot Hueber</td>
<td>Pietro Iglio</td>
<td>Alejandro Jakubi</td>
</tr>
<tr>
<td>Richard Jenks</td>
<td>Bo Kagstrom</td>
<td>William Kahan</td>
</tr>
<tr>
<td>Kyriakos Kalorkoti</td>
<td>Kai Kaminski</td>
<td>Grant Keady</td>
</tr>
<tr>
<td>Wilfrid Kendall</td>
<td>Tony Kennedy</td>
<td>David Kincaid</td>
</tr>
<tr>
<td>Keshav Kini</td>
<td>Ted Kosan</td>
<td>Paul Kosinski</td>
</tr>
<tr>
<td>Igor Kozachenko</td>
<td>Fred Krogh</td>
<td>Klaus Kusche</td>
</tr>
</tbody>
</table>
Bernhard Kutzler  Tim Lahey  Larry Lambe
Kaj Laurson  Charles Lawson  George L. Legendre
Franz Lehner  Frederic Lehobey  Michel Levauud
Howard Levy  J. Lewis  Ren-Cang Li
Rudiger Loos  Craig Lucas  Michael Lucks
Richard Luczak  Cammi Maguire  Francois Maltey
Osni Marques  Alasdair McAndrew  Bob McElrath
Michael McGettrick  Edi Meier  Ian Meikle
David Mentre  Victor S. Miller  Gerard Milmeister
Mohammed Mobarak  H. Michael Moeller  Michael Monagan
Marc Moreno-Maza  Scott Morrison  Joel Moses
Mark Murray  William Naylor  Patrice Naudin
C. Andrew Neff  John Nelder  Godfrey Nolan
Arthur Norman  Jinzhong Niu  Michael O'Connor
Summat Oemrawsingh  Kostas Oikonomou  Humberto Ortiz-Zuazaga
Julian A. Padget  Bill Page  David Parnas
Susan Pelzel  Michel Petitot  Didier Pinchon
Ayal Pinkus  Frederick H. Pitts  Frank Pfennig
Jose Alfredo Portes  E. Quintana-Orti  Gregorio Quintana-Orti
Beresford Parlett  A. Petitet  Andre Platzer
Peter Poromaas  Claude Quitte  Arthur C. Ralfs
Norman Ramsey  Anatoly Rapoportirenko  Guillaume Reis
Huan Ren  Albert D. Rich  Michael Richardson
Jason Riedy  Renaud Rioboo  Jean Rivlin
Nicolas Robidoux  Simon Robinson  Raymond Rogers
Michael Rothstein  Martin Rubey  Jeff Rutter
Philip Santas  Alfred Scheerhorn  William Schelter
Gerhard Schneider  Martin Schoenert  Marshall Schor
Fritjof Schulze  Fritz Schwarz  Steven Segletes
V. Sima  Nick Simicich  William Sit
Elena Smirnova  Jacob Nyyf€€er Smith  Matthieu Sozeau
Ken Stanley  Jonathan Steinbach  Fabio Stumbo
Christine Sundaresan  Klaus Sutner  Robert Sutor
Moss E. Sweedler  Eugene Surowitz  Max Tegmark
T. Doug Telford  James Thatcher  Laurent Thery
Balbir Thomas  Mike Thomas  Dylan Thurston
Francoise Tisseur  Steve Toleque  Raymond Toy
Barry Trager  Themis T. Tsikas  Gregory Vanuxem
Kresimir Veselic  Christof Voemel  Bernhard Wall
Stephen Watt  Andreas Weber  Jaap Weel
Juergen Weiss  M. Weller  Mark Wegman
James Wen  Thorsten Werther  Michael Wester
R. Clint Whaley  James T. Wheeler  John M. Wiley
Berhard Will  Clifton J. Williamson  Stephen Wilson
Shmuel Winograd  Robert Wisbauer  Sandra Wityak
Waldemar Wiwianka  Knut Wolf  Yanyang Xiao
Liu Xiaojun  Clifford Yapp  David Yun
Qian Yun  Vadim Zhytnikov  Richard Zippel
Evelyn Zoernack  Bruno Zuercher  Dan Zwillinger
## 3 Axiom and Category Theory

3.1 Covariance and Contravariance ........................................ 41

3.2 Axiom Type Lattice .................................................. 42

3.3 Terms to Understand .................................................. 42

3.4 Category Definition ................................................ 43

3.5 Monoids and Groups ................................................ 44

## 4 Axiom Implementation Details

4.1 Makefile .......................................................... 45

## 5 Writing Spad Code

5.1 The Description: label and the \texttt{describe} command ............. 47

## 6 Writing test cases

## A The Principles of Axiom

## B The Axiom Conventions

## C Example Code

C.1 domain \texttt{WP WeightedPolynomials} .......................... 57

C.2 domain \texttt{OWP OrdinaryWeightedPolynomials} ............... 59

C.3 domain \texttt{WP2 WeightedPolynomials2} ...................... 60

C.4 domain \texttt{FCOMP FourierComponent} ....................... 63

C.5 domain \texttt{FSERIES FourierSeries} ......................... 64

## D The Makefile

Bibliography

Index

Index
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))
Chapter 1

The Axiom System by James H. Davenport

This is (mostly) quoted verbatim, with permission, from Davenport[Dave92a].

1.1 A little history

In 1978 the present author spent two months at IBM Yorktown Heights, as part of the Computer Algebra Group, which had developed the Scratchpad-1 computer algebra system. Though this system never saw the light of day outside IBM, it was at the time a competitor for Macsyma and Reduce. All systems had struggled with the problems of writing ever more complicated algebraic algorithms, and handling the growth of the systems. Zippel has estimated that Macsyma, at its heyday, contained six different Gaussian elimination algorithms, whether because they were handling different data types, or because the authors were unaware of the other ones, or for more subtle reasons. Davenport[Dave81] had been having similar problems with Reduce, and a feeling of discontent among algorithm implementors was common.

The reactions of the computer algebra system-builders to the complexity are interesting to tabulate.
Macsyma embarked on the, ultimately futile, “new rational function” project, which was to have re-written much of the algebraic kernel in a more mathematically structured way, but which was unable to maintain backwards compatibility.

Reduce developed into Reduce 3, with its concept of domains, which meant that new constant domains could be added relatively simply. See Bradford et al.[Brad86]

Waterloo’s emerging team decided that none of the existing solutions was right, and opted for a new design based round a very small kernel, typically not knowing any algebra, and loadable modules, which would contain the algebraic knowledge of the system. This became the Maple system. Over the years, the definition of the kernel has changed, as it became obvious that certain algorithms, e.g. modular ones, could not be implemented efficiently in the interpreted modules.

MuMath and its successor Derive based themselves on the philosophy that “whatever fitted on a PC” was much closer to most users’ requirements than “the best possible mathematics”, and, in terms of the number of users, they are certainly correct.

IBM embarked on a lengthy period of algebraic reflection, prototyping and experimentation, sporadically reported in the literature Davenport & Jenks[Dave81a] and Jenks & Trager[Jenk81] first product of this process. Axiom is the end product of this process of reflection.

1.2 Axiom’s philosophy

Axiom shares with Maple the desire to build a system consisting of a kernel and loadable modules, written in an appropriately high-level language: the kernel knows very little algebra (in the classical sense of the term) and the modules define what algebraic facilities are present. The resemblance ends here, though. Maple’s modules are interpreted, sacrificing the ultimate in performance for small size and rapid loading, whereas Axiom’s modules are compiled into machine code for speed, much as in Reduce or Macsyma, but also for type analysis and constructing the database for the information system. There are certainly arguments in favour of Maple’s approach, though it does assume that the designers know a priori which primitives to build into the kernel to make a fast algebra system. Hence Maple is relatively fast at those operations for which it was designed, but attempts to make it into, say, a computational group theory package, have not been particularly successful, since the inner loops have been taking place in interpreted code. Another difference is, regrettably, that Axiom’s kernel is currently far larger than Maple’s, largely due to the genericity of the type system it implements.

Axiom’s kernel designers took the view that they did not know what algebraic facilities would be wanted, nor how they would be programmed. The only real assumption made was that the computations would be largely symbolic, and hence big integers, lists, vectors and trees were emphasised over, say, the efficient compilation of fixed-precision floating-point, which is left to externally-compiled code. There are also assumptions in the user interface about the wish to convert symbols into polynomials where appropriate, but these are in fact not fixed. The graphics facilities provided are more oriented towards the production of graphs of functions than other kinds of illustrations, but new facilities could easily be added.

Principle 1. AXIOM has an interpreter for interactive use, much like any other system,
and a compiler for creating new user-defined data types. The compiler emphasises strict type-checking, whilst the interpreter is more oriented towards ease of use.

The complexity of the algebraic facilities envisaged for Axiom required a data-typing mechanism over and above that provided by Lisp. To quote a very simple example, \(1 + x + x^2\) could be either a polynomial or a truncated Taylor series, but the square of the polynomial is \(1 + 2x + 3x^2 + 2x^3 + x^4\) whereas the square of the series is \(1 + 2x + 3x^2\). Similarly, if 2 represents the integer 2, then \(2 + 2 = 4\), whereas if 2 represents the congruence class “integers congruent to 2 modulo 3” then \(2 + 2 = 1\) (and, of course, \(4 = 1\) as well). Similarly, the list \((1,2)\) is not the same as the list \((2,1)\), but they should be regarded as the same if they represent (unordered) sets, and so on. In fact, the data typing requirements of computer algebra are so dynamic – the authors cannot predict what types the users will call for, explicitly or implicitly – and so rich that no existing language was suitable for expressing them. Hence the decision was taken that Axiom would have its own independent typing system.

This typing system, which underpins much of the rest of Axiom, has to solve the problems that Macsyma and Reduce have. Macsyma’s typing system suffers, essentially, from the \(n^2\) problem – every type has to know about every other type. This works when there are only a few types, and Macsyma has “general expression”, “rational function” (printed with /R/), and “Taylor series” (printed with /T/): adding more types would be difficult.

Reduce’s method of specifying domains is largely global: for example one specifies the modular domain by issuing the command on modular; one has then to be sure that all calculations are intended to be modular and that values being computed will not later be used as polynomial exponents, loop indices or whatever. There is much scope for hard-to-detect bugs in this area. The other drawback of Reduce’s scheme is that it really only applies to constant domains. This works well for floating point or complex coefficients, but has its weaknesses when it comes to handling Taylor series. For example

1: load tps; \% truncated power series;

2: ps(cos x,x,0);
\[
\begin{align*}
1 & \quad 2 \quad 1 \quad 4 \quad 1 \quad 6 \quad 7 \\
\{1 - (---)*X + (----)*X - (-----)*X + O(X )\} \\
2 & \quad 24 \quad 720
\end{align*}
\]

3: ws-1;
\[
\begin{align*}
1 & \quad 2 \quad 1 \quad 4 \quad 1 \quad 6 \quad 7 \\
\{- (----)*X + (-----)*X - (-----)*X + O(X )\} \\
2 & \quad 24 \quad 720
\end{align*}
\]

4: ps(sin x,x,0);
\[
\begin{align*}
1 & \quad 3 \quad 1 \quad 5 \quad 7 \\
\{X - (---)*X + (-----)*X + O(X )\} \\
6 & \quad 120
\end{align*}
\]

5:ws-x;
\[
\begin{align*}
1 & \quad 3 \quad 1 \quad 5 \quad 7
\end{align*}
\]
4

CHAPTER 1. THE AXIOM SYSTEM BY JAMES H. DAVENPORT

\[-X + \{X - (-\cdots)X + (\cdots\cdots)X + O(X)\}\]

6 \quad 120

6: \text{ws 4 - ps}(x,x,0);

\begin{align*}
1 & \\
3 & \\
1 & \\
5 & \quad 7 \\
6 & \\
120 & \\
\end{align*}

\[-(\cdots)X + (\cdots\cdots)X + O(X)\]

6 \quad 120

The spurious power series expansion at line 6: is necessary to avoid the confusion at line 5:, where the variable \(x\) is separated from the power series in \(x\).

1.3 Axiom’s typing scheme

The typing scheme of Axiom can be described as a two-level typing scheme with single inheritance of types and multiple inheritance of meta-types. What does this mean, when stripped of the jargon? The first piece of jargon we wish to remove is the word “type”, which is so heavily used in computer science that it has practically ceased to have any meaning at all. The word that most nearly corresponds in Axiom is the word domain, as we shall see.

Definition 1. A domain is a set of values (possibly infinite), and the operations which can be performed on them.

This corresponds rather closely to a “data type” in much modern programming language theory.

Principle 2. Every internal Axiom data object belongs to one and only one domain.

Thus the integer “2” belongs to the domain \texttt{Integer}, whereas the congruence class modulo 3 “2” belongs to the domain \texttt{IntegerMod(3)}, which can also be written as \texttt{IntegerMod 3}, thanks to the following.

Convention 1. Juxtaposition corresponds to (unary) function application.

This corresponds with the traditional mathematical convention that \(\sin x\) means the same as \(\sin(\text{\(x\)})\). The user should be warned, however, that juxtaposition has a high precedence, and that \(\sin x**2\) parses as \((\sin x)**2\) and not as \(\sin(x**2)\). This just shows the richness of mathematical notation that formal grammars of any kind find hard to capture.

Principle 2 can be seen in the following mini-session with Axiom:

\begin{verbatim}
->a:Integer
->b:IntegerMod(3)

->a:=2

(3) 2

->a:=a+a

(4) 4

->b:=2

(5) 2
\end{verbatim}
1.3. AXIOM’S TYPING SCHEME

\[ \text{->b:=b+b} \]

(6) 1

The first two lines declare the domains to which the values of a and b may belong – loosely speaking the types of the variables a and b – then they are given values, which are added. As we said earlier, \( a + a = 4 \), whereas \( b + b = 1 \). This may appear confusing, so let’s run through the same session, but asking Axiom to print out the domains of the various values. This is done by means of the system command \( \text{set message type on} \).

**Convention 2 (borrowed from APL).** All system commands, i.e. those that do not perform, or affect the performance of, algebraic operations, begin with \( ) \). In general, they may be contracted as far as is unambiguous, so that \( \text{set message type on} \) can be contracted as far as \( \text{se m ty on} \).

In addition to the Hyperdoc help system, information about system commands can be found using \( \text{help} \).

\[ \text{->a:Integer} \]  
\[ \text{Type: Void} \]

\[ \text{->b:IntegerMod(3)} \]  
\[ \text{Type: Void} \]

\[ \text{->a:=2} \]  
\[ (3) \text{ 2} \]  
\[ \text{Type: Integer} \]

\[ \text{->a:=a+a} \]  
\[ (4) \text{ 4} \]  
\[ \text{Type: Integer} \]

\[ \text{->b:=2} \]  
\[ (5) \text{ 2} \]  
\[ \text{Type: IntegerMod 3} \]

\[ \text{->b:=b+b} \]  
\[ (6) \text{ 1} \]  
\[ \text{Type: IntegerMod 3} \]

Note that the declarations themselves are algebraic commands, and therefore their results must belong to a domain: in this case the \( \text{Void} \) domain. The numbers before the values can be used to refer to these values later.

**Convention 3.** The symbol \( \% \) refers to the most recently computed proper value (i.e. not of the \( \text{Void} \) domain). \( \%n \) or \( \%n \), refers to the value numbered \( n \), if \( n \) is a positive integer. If \( n \) is a negative integer, \( \%n \) refers to the value of the \( |n| \)'th previous step. Also, \( \pi \) refers to \( \pi \approx 2.718281828 \) and \( \sqrt{-1} \) to \( i \).

Note that \( \% \) is not a synonym for \( \%(-1) \), since if the previous step were a declaration, then \( \%(-1) \) would belong to the domain \( \text{Void} \), whereas \( \% \) would refer to the last non-void object.

Principle 2 has an apparent exception, which we can see in the example above if, instead of writing \( \text{a:=a+a} \), we had just tried \( \text{a+a} \), i.e. asked for the value to calculated, but not to
replace the old a.

\[
\text{Type: PositiveInteger}
\]

The 4 now belongs to \textbf{PositiveInteger} whereas it used to belong to \textbf{Integer}, yet we are performing the same calculation. The answer is that \textbf{PositiveInteger} is not actually a separate domain from \textbf{Integer}, rather it is a \textit{sub-domain} (a concept we shall define formally later). Whilst it is possible for users to add new sub-domains to Axiom, there are two built-in ones, with the inclusion relationships

\[\text{PositiveInteger} \subseteq \text{NonNegativeInteger} \subseteq \text{Integer}\]

and a general rule about \textit{Union} domains that will be explained later. An element of a domain which is also an element of a sub-domain can move freely to a larger sub-domain, or to the whole domain, as required. The reason for the existence of these sub-domains is to allow more thorough type-checking: for example a square matrix has to have a dimension which is a \textbf{NonNegativeInteger}, and it only makes sense to raise polynomials to \textbf{NonNegativeInteger} powers. Similarly, the argument to \textbf{IntegerMod} must be a \textbf{PositiveInteger}. In order to make \textit{interactive} use easier, the interpreter will automatically convert elements of sub-domains into those sub-domains. This can be summarised as follows.

\textbf{Principle 3}. Values can freely move from sub-domains to larger ones, and, in the interpreter only, in the other direction, provided that this conversion is legitimate.

Compilers clearly can’t move from a large domain to a smaller one, since they have no idea whether such a contraction will always be possible – if the programmer knows that it will always be possible, they have to declare the fact.

\textbf{Aren’t all these types confusing?}

The casual user need not concern themselves with the type system: those functions that most other systems provide, and which correspond to general algebra and calculus, work through the type system provided. For example, the following session could be taken from any algebra system.

\[
\text{->sin(x)}
\]

\[
(1) \text{sin(x)}
\]

\[
\text{->integrate(%,x)}
\]

\[
(2) \text{ - cos(x)}
\]

\[
\text{->series (%,x=%pi/2)}
\]

\[
(3)\begin{array}{c}
\%pi & 1 & \%pi & 3 & 1 & \%pi & 5 & 1 & \%pi & 7 \\
(x - ---) & - & (x - ---) & + & (x - ---) & - & (x - ---) \\
2 & 6 & 2 & 120 & 2 & 5040 & 2 \\
\end{array} \\
+ \begin{array}{c}
1 & \%pi & 9 & 1 & \%pi & 11 & \%pi & 12 \\
--- & (x - ---) & - & (x - ---) & + & O((x - ---)) \\
\end{array}
\]
1.3. AXIOM’S TYPING SCHEME

We note that the second use of integrate did not require, and indeed cannot be given, a variable. Since the expression is a series in \( x - \pi/2 \), it can only be integrated with respect to \( x \), and the type system ensures this. In fact the domains of these results are, respectively, \texttt{Expression Integer}, which is the workhorse for much of calculus, \texttt{Union(Expression Integer,List Expression Integer)} and \texttt{UnivariatePuiseuxSeries(Expression Integer,x,}\( \pi/2 \)) for the last two. These last two require some explanation, which is given in the section 1.4 “Some AXIOM facilities” on page 8.

Axiom naturally manipulates various types of composite data structures: lists, vectors, sets and so on.

**Convention 4 (a convention of the library, rather than of the kernel).** Parentheses – \( () \) – are used for grouping and function application, brackets – \([ ] \) – are used for constructing lists, and braces – \{ \} – are used for constructing sets.

The difference between lists and sets is that lists can contain repetitions, and order matters, whereas sets, as in mathematics, are unordered and without repetition.

\[
->[2,1,2,1]
\]

(1) \[2,1,2,1] \hspace{2cm} \text{Type: List PositiveInteger}

\[
->{2,1,2,1}
\]

(2) \{1,2\} \hspace{2cm} \text{Type: Set PositiveInteger}

Suppose we had a list of objects, and wished to convert it into a set, e.g., in the situation above, we do not want to retype the 2,1,2,1. This is handled by a very general mechanism in Axiom.

**Convention 5.** The :: in x operator, used as in

\[\text{Axiom object :: Axiom domain}\]

\[
\text{can be used to convert the object to lie in the specified domain.}
\]

The :: operator is partially built into the Axiom kernel. When new data types are defined, the definition includes some coerce functions between the new type and some existing types. However, the :: operator is more than just one of these programmed conversions: it is at least what an algebraist would call the transitive closure of these operations, so that if there are coerce functions from A to B, and from B to C, then :: can convert from A to C. In
fact, it is more than this: if a functor, such as \texttt{List} possesses a \texttt{map} operation of signature
\[(A \to B, \text{List } A) \to \text{List } B\]
and it is possible to coerce objects from \(A\) to \(B\), then the system will be able to coerce objects from \texttt{List } A\ to \texttt{List } B. More details are given in Sutor & Jenks [Jenk92].

**Principle 4.** The interpreter is responsible for performing any chain of coercions necessary to understand the user’s intentions, or when required to do so by an explicit use of :: . The compiler will perform a chain of coercions when instructed to do so by the :: operator in compiled code.

So we could replace command (2) above by

```plaintext
->%::Set PositiveInteger
(2) \{1,2\}
```

A large number of coercions are performed automatically. Even the simple computation \(x + 1\) causes three coercions:

1. the variable \(x\) from the domain \texttt{Variable} to the domain \texttt{Polynomial Integer};
2. the number 1 from the domain \texttt{PositiveInteger} to the domain \texttt{Polynomial Integer}, passing via \texttt{Integer};
3. the result \(x + 1\) from the domain \texttt{Polynomial Integer} to the domain \texttt{OutputForm}, using sub-coercions of \(x\) and 1 to \texttt{OutputForm}.

All printing actually takes place from the domain \texttt{OutputForm}, which is also the starting point for conversions to TeX format, Fortran etc. This means that a new domain which can be printed at all (i.e. which can be coerced to \texttt{OutputForm}) can be printed in TeX, Fortran and indeed in any other ways that get added later, without having to modify the domain at all.

### 1.4 Some AXIOM facilities

Computer algebra is often also called “symbolic manipulation”, and Axiom excels at manipulating symbols as such. A symbol can be as simple as \(x\) or as complex as

\[
\Omega (\theta)
\]

(1) \(x\) \((a,b)\)

\[
7\ 1,2\quad 1
\]

obtained via the following command:

```plaintext
\texttt{script(x,[[1,2],[paren theta],[Omega],[7],[a,script(b,[[1]])]])}
```

This symbol can be converted into TEX format by means of the \texttt{outputAsTex} function: the result is shown below.

\[
\Omega \ x_{1,2} (a, b_1)
\]

It is still a single symbol, and the command

```plaintext
->\texttt{integrate(sin }%\texttt{,%)}
```

\[
\Omega (\theta)
\]
1.4. SOME AXIOM FACILITIES

(4) \[- \cos(\begin{array}{c}x \\ 7,1,2 \\ 1 \end{array},a,b)\]

Type: Union(Expression(Integer),...)

is no different from \texttt{integrate(sin(x),x)}.

Axiom has a rich integrator, based on the developments by Bronstein[Bron90a]. As we saw earlier, and just above, it seems to give a rather complicated domain* for the result: why not just \texttt{Expression Integer}? This expression certainly looks like an expression with integer entries, and seems to behave as one. First, we need to explain what \texttt{Union} is.

**Principle 5.** Any set of Axiom domains $D_1, \ldots, D_n$ can be combined into a (disjoint) union domain, denoted $\text{Union}(D_1, \ldots, D_n)$. The $D_i$ are called the branches of the union. The operations available on this union domain are:

- equality – two elements are equal if they come from the same branch and are equal in that branch;
- coercion to \texttt{OutputForm};
- coercion from each $D_i$ to the union domain;
- coercion to each $D_i$ from the union domain, which may fail if the union object is not in the correct branch;
- an \texttt{in x predicate case}, for testing if the union object actually is in a particular branch or not.

These union domains correspond to what some other languages call “sum types”. A particularly useful case is exemplified by the “exact quotient” operation on \texttt{Integer}: its return type is $\text{Union}(\text{Integer}, \text{"failed"})$, where the special token \texttt{failed} is returned if the division is not exact.

So we are saying that Axiom’s integrator can return either an expression, or a list of expressions. A simple example of it doing the latter is the following.

\[\rightarrow \text{integrate}(1/(x**2-a),x)\]

(4) \[\begin{array}{c}
\log(\frac{x + a}{a - 2a x}) \\
\atan(\frac{2}{x - a}) \\
\end{array}\]

Here, there are two possible answers, depending on the sign of $a$. Since Axiom has no way of knowing which is required, it returns both, and leaves it to the user, or the caller of the \texttt{integrate} command, to supply the higher knowledge necessary to determine which element of the list to use (and it may not always be the same one). In some sense, they are equally correct, as we can check by differentiating them.

\[\rightarrow \text{[differentiate(f,x) for f in %]}\]

(5) \[\begin{array}{c}
\frac{1}{2} \\
\end{array}\]
Note the neat way of handling the list. There are many other such list handling techniques, and the user can also use \texttt{map}, a function provided on most of the library’s compound data types. This is how \texttt{map} could differentiate the elements of that list.

\[
\texttt{-> map(f+->differentiate(f,x),}\%4)
\]

\[
\begin{array}{llllllllll}
1 & 1 & & & & & & & & \\
2 & 2 & & & & & & & & \\
x & - & a & x & - & a
\end{array}
\]

**Convention 6 (Of the library authors).** *The notation*

\[
\text{list of variables} + - \to expression
\]

*defines an anonymous function of those variables. It corresponds to the lambda-calculus expression “variables.expression”.*

As we have already seen, Axiom has a powerful series capability. As pioneered by Norman [Norm75] in Scratchpad-1, these series are lazy: terms are only evaluated as required for printing, and more terms can always be evaluated as required.

\[
\texttt{-> series(sin(x),x=0)}
\]

\[
\begin{array}{llllllllllll}
1 & 3 & 1 & 5 & 1 & 7 & 1 & 9 & 1 & 11 & 12 \\
6 & 120 & 5040 & 362880 & 39916800
\end{array}
\]

\[(1) x - \frac{1}{x} + \frac{3}{x^3} - \frac{1}{x^5} + \frac{5}{x^5} - \frac{1}{x^7} + \frac{7}{x^7} - \frac{1}{x^9} + \frac{9}{x^9} + O(x^0)
\]

\[
\texttt{-> %/x}
\]

\[
\begin{array}{llllllllllll}
1 & 3 & 1 & 5 & 1 & 7 & 1 & 9 & 1 & 11 & 12 \\
6 & 120 & 5040 & 362880 & 39916800
\end{array}
\]

\[(2) - x - \frac{1}{x} + \frac{3}{x^3} - \frac{1}{x^5} + \frac{5}{x^5} - \frac{1}{x^7} + \frac{7}{x^7} - \frac{1}{x^9} + \frac{9}{x^9} + O(x^0)
\]

\[
\texttt{-> %-1}
\]

\[
\begin{array}{llllllllllll}
1 & 3 & 1 & 5 & 1 & 7 & 1 & 9 & 1 & 11 & 12 \\
6 & 120 & 5040 & 362880 & 39916800
\end{array}
\]

The number of terms initially calculated (and therefore displayed) is controlled by the system command \texttt{set streams calculate}. The type of this result is \texttt{UnivariatePuiseuxSeries(Expression Integer,x,0)}.

The last two parameters are clearly the variable and the point about which we are expanding, but what on earth is a Puiseux series? Why cannot Axiom give us the familiar Taylor series, which this certainly looks like being? A Taylor series represents a continuous function \(a_0x^0 + a_1x^1 + \cdots\). One way of generalising this is to allow meromorphic functions, i.e. those with a point singularity of the \(1/x\) (more generally \(1/x^n\)) variety. In order to represent these, we have to allow (a finite number of) negative exponents in the series – so-called Laurent series.

\[
\texttt{-> series(1/sin(x),x=0)}
\]

\[
\begin{array}{llllllllllll}
1 & 1 & 7 & 3 & 31 & 5 & 127 & 7 & 73 & 9 & 10 \\
6 & 360 & 15120 & 604800 & 3421440
\end{array}
\]

\[(4) x - \frac{1}{x} + \frac{1}{x} - \frac{1}{x} + \frac{3}{x} + \frac{-1}{x} + \frac{5}{x} + \frac{-1}{x} + \frac{7}{x} + \frac{-1}{x} + \frac{9}{x} + \frac{-1}{x} + \frac{10}{x} + O(x^0)
\]
However, even these are not as general as one would like, since they are incapable of representing multi-valued functions like $\sqrt{x}$. To do this, we have to allow fractional exponents (of bounded denominator), which gives us Puiseux series.

$$\sqrt{x} = \frac{1}{2}x^{1/2} + \frac{3}{12}x^{3/2} + \frac{7}{160}x^{7/2} + O(x)$$

$$\frac{1}{2}x^{1/2} = \frac{9}{6}x^{1/2} + \frac{1}{360}x^{3/2} + O(x)$$

Note that Axiom, like any other algebra system, cannot prove that the difference of these two series is identically zero, merely that in going all the way up to the limit required by set streams calculate, it can find no non-zero terms. Puiseux series have many uses in algebraic geometry. See Davenport\cite{Dave81}

### How does one keep track of all this?

There seem to be so many different names and domains around in the Axiom system. How does one keep track of them all, and know what to use? There is an on-line help system Hyperdoc, with tutorial material and information arranged by subject, but the system itself provides some help.

**Convention 7.** The names of Axiom functions are either special symbols (such as $+$) or complete English words strung together. In this case, every word after the first is capitalised. Thus `integrate` but `complexIntegrate`. In addition:

- all boolean predicates end in a `?`, as in `odd?`, which tests if a number is odd
- all destructive functions which operate on data structures end in a `!`, as in `reverse!`, which reverses a list destructively.

Conversely, the names of domains (and other constructors we will come to later) consist of English words strung together, all of which are capitalised, as in `IntegerMod` or `UnivariatePuiseuxSeries`.

One can search (case-insensitively) for all functions whose name contains a particular word by using the system command `)what operations`, contractible to `)wo`, as in

```->)what operations series```
Operations whose names satisfy the above pattern(s):

series   seriesSolve

To get more information about an operation, say series, issue the command )display op series.

As it says, the command )display operations, contractible to )d o, can be used to find out what the arguments of an operation should be. However, in order to explain this, we have to delve rather deeper into Axiom’s type system. The user of the Nag Fortran library sees nothing strange in writing one subroutine to multiply real matrices, and a different one to multiply complex matrices. Indeed, one would be hard put to do anything else in Fortran 77. The user of the Nag Ada library, in contrast, would expect to find a generic matrix multiplication routine, which could be called for any built-in real or complex type, and possibly for additional user-defined types. Axiom’s type system is much closer to the Ada one than the Fortran one, but in fact even more general than the Ada model.

Just as one can make various different modular domains by applying the functor\(^1\) \texttt{IntegerMod} to different integers, so one can make different matrix domains by applying the functor \texttt{Matrix} to different domains for the coefficients.

\[
\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} :: \texttt{Matrix IntegerMod 3}
\]

\[
\begin{array}{ll}
+1 & 2+ \\
(1) & 1+ \\
0 & 1+
\end{array}
\]

Type: \texttt{Matrix(IntegerMod(3))}

\[
\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} :: \texttt{Matrix IntegerMod 5}
\]

\[
\begin{array}{ll}
+1 & 2+ \\
(2) & 1+ \\
3 & 4+
\end{array}
\]

Type: \texttt{Matrix(IntegerMod(5))}

\[
\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} :: \texttt{Matrix Float}
\]

\[
\begin{array}{ll}
+1.0 & 2.0+ \\
(3) & 1+ \\
3.0 & 4.0+
\end{array}
\]

Type: \texttt{Matrix(Float)}

In each case, the coefficient arithmetic is done according to the correct rules of the coefficient domain.

\[
\begin{pmatrix} 1,2 \\ 3,4 \end{pmatrix} ** 2
\]

\[
\begin{array}{ll}
+1 & 1+ \\
(4) & 1+ \\
0 & 1+
\end{array}
\]

Type: \texttt{Matrix(IntegerMod(3))}

\[
\begin{pmatrix} 1,2 \\ 3,4 \end{pmatrix} ** 2
\]

\[
\begin{array}{ll}
+2 & 0+ \\
(5) & 0+
\end{array}
\]

\(^1\) Axiom terminology, borrowed from category theory, uses the word “functor” for those functions that yield domains as their result, whereas “function” is reserved for operations whose value is an Axiom object.
1.5. CATEGORIES

Principle 6. The Axiom library declares a family of second-order types, known as categories. The categories are arranged in a directed acyclic graph, and each domain belong to a specific category, and to all the ancestors of that category. The specification of a category includes

- all its direct ancestors,
- any additional operations that this category supports, and
- any additional axioms that the operations must satisfy.

The operation Join is used to construct new categories.

This may appear a bit abstract, so let’s look at an example from the foundations of the Axiom library. The fundamental category in Axiom is SetCategory.

Convention 8. Whenever a category, or domain, is being discussed in Axiom, the symbol \% stands for the domain in question, or for any domain from the category in question.

With the help of that notation, we can ask Axiom what the definition of SetCategory is.

```
->)show SetCategory
  SetCategory is a category constructor.
  Abbreviation for SetCategory is SETCAT
  This constructor is exposed in this frame.
  Issue )edit bookvol10.2.pamphlet to see algebra source code for SETCAT
```
This output shows that there are two operations defined on any domain, denoted by \(\%\), which belongs to the category \textit{SetCategory}: an infix operation \(=\) which takes two arguments from \(\%\) and yields a \texttt{Boolean} result, and an operation called \texttt{coerce}, which takes an element of \(\%\), and yields an \texttt{OutputForm}. How does this grow into more useful categories? A graphic representation of some of the first few extensions is given below.

\[
\text{SetCategory} \downarrow \ AbelianSemiGroup \downarrow \ AbelianMonoid \downarrow \ CancellationAbelianMonoid \downarrow \ AbelianGroup \downarrow \ \text{AbelianGroup}
\]

\[
\text{AbelianSemiGroup} \downarrow \ AbelianMonoid \downarrow \ CancellationAbelianMonoid \downarrow \ AbelianGroup \downarrow \ \text{AbelianGroup}
\]

\[
\text{AbelianGroup} \downarrow \ \text{AbelianGroup}
\]

\[
\text{AbelianGroup} \downarrow \ \text{AbelianGroup}
\]

\[\rightarrow \]

\[
\text{AbelianGroup} \downarrow \ \text{AbelianGroup}
\]

\[
\text{AbelianGroup} \downarrow \ \text{AbelianGroup}
\]

\[
\text{AbelianGroup} \downarrow \ \text{AbelianGroup}
\]

\[
\text{AbelianGroup} \downarrow \ \text{AbelianGroup}
\]

where the arrows indicate a “direct descendant” relationship.

\textit{AbelianSemiGroup} is defined to have one new operator:

\[
+ : \% \times \% \rightarrow \%
\]

satisfying the associative and commutative axioms:

\[
a \times (b + c) = (a + b) + c
\]

\[
a + b = b + a
\]

\textit{AbelianMonoid} introduces a new nullary operator\(^2\)

\[
0 : \rightarrow \%
\]

satisfying the obvious axiom

\[
0 + a = a
\]

\textit{CancellationAbelianMonoid} is the category of abelian monoids with the cancellation axiom:

\[
a + b = a + c \Rightarrow b = c
\]

Constructively, this is represented by a partial subtraction operator, whose signature is defined as:

\[
- : \% \times \% \rightarrow \text{Union}(\%, \"failed\")
\]

While such an operation could be defined for any \textit{AbelianMonoid}, or even any \textit{AbelianSemiGroup}, it is the cancellation axiom that ensures that \(-\) has a unique value. This operator is subsumed in the \(-\) operation defined on \textit{AbelianGroups}.

\(^2\) Technically speaking, it is a constant, rather than a nullary operator, which means that the value is computed once and for all when the type is created. The difference is essentially one of efficiency, and we will not discuss it further here.
AbelianGroup adds one further unary operator \(^3\)

\[ - : \% \mapsto \% \]

This operator satisfies the axiom

\[ a + (-a) = 0 \]

The first \(\downarrow\) introduces an operator

\[ <: \% \times \% \mapsto Boolean \]

satisfying the usual axioms:

\[ a < b \lor b < c \Rightarrow a < c \]
\[ \neg (a < b) \lor \neg (b < a) \Rightarrow a = b \]
\[ a < b \Rightarrow \neg (b < a) \]

Subsequent \(\downarrow\) in this diagram introduce no new operators, but one more axiom is introduced, when OrderedAbelianSemiGroup is defined:

\[ a < b \Rightarrow a + c < b + c \]

This is typical of what happens when two categories are merged to form a new named category: we keep the same operators, but are interested in the interaction between them, which requires the introduction of new axioms to define this interaction. Subsequent \(\downarrow\) in the chain represent the straightforward merging of ancestors.

It should be noted that none of these definitions are hard-coded in the Axiom kernel: merely the mechanism for understanding them is part of the kernel. The definitions are written in the Axiom category definition language, and could be modified or extended to suit different kinds of mathematics. See Lambe[Lamb89], Schwarz[Schw88] The code reads, in essence, as given below.

**Convention 9.** Axiom comments can be introduced by \(--\) or \(++\). Those beginning \(++\) are intended for the user, and can be retrieved by the on-line help system.

For example, the comment following \(\text{zero?}\) in AbelianMonoid is retrieved when the operation name \(\text{zero?}\) is searched for.

\[ \text{SetCategory()}: \text{Category} == \text{Join(Object, CoercibleTo OutputForm)} \text{ with} \]

\[ "==" : (\%,\%) \mapsto Boolean \]

\(^3\) In practice, it also adds a binary operator

\[ - : \% \times \% \mapsto \% \]

satisfying the axiom

\[ a - b = a + (-b) \]

From a logical point of view, the binary operator is redundant. It is present in Axiom for two reasons. The first is legibility of programs: \(a - b\) is easier to read than \(a + (-b)\). The second is efficiency: while the binary operator can always be implemented in terms of the unary operator, and indeed has a default definition implementing it this way, it is not necessarily very efficient to do so. For example, if \% is a matrix type, implementing the binary operator in terms of the unary operator causes two new matrices to be allocated instead of one.
++ Axioms:
++ associative("+":(%,%)->%) || \space{ (x+y)+z = x+(y+z) } 
++ commutative("+":(%,%)->%) || \spad{ x+y = y+x } 

AbelianSemiGroup(): Category == SetCategory with

"+": (%,%) -> %
++ Axioms:
++ leftIdentity("+":(%,%)->%,0) || \spad{ 0+x=x }
++ rightIdentity("+":(%,%)->%,0) || \spad{ x+x=x }

AbelianMonoid(): Category == AbelianSemiGroup with

0: constant -> % ++ 0 is the additive identity element

zero?: % -> Boolean ++zero?(x) tests if x is equal to 0

add

zero? x == x = 0

The clause after add introduces a default definition of zero? in terms of = and 0.

Principle 7. Categories can introduce default definitions of operations, which will take
effect in any domain belonging to that category unless overridden by a definition in that
domain, or in a more specific category.

Further details of the general mechanism are given in Jenks et al. [Jenk92], and the actual
categories implemented in the Axiom library are described in Davenport & Trager [Dave90]
and Davenport et al. [Dave91]. However, we can see that it is possible to define such a system
of categories which will act, in effect, as a type system for the domains themselves.

Principle 8. The functors of Axiom are strongly typed: each parameter which is an Axiom
object is specified to come from a particular domain; each parameter which is an Axiom
domain is specified to belong to a particular Axiom category. Similarly, the domain returned
by a particular functor is specified to belong to a particular category. All construction of
domains must satisfy these constraints on the functors.

To take the example of Matrix, the definition of the functor could begin as follows:

Matrix(R:Ring): MatrixCategory(R, Vector R, Vector R) ==

where MatrixCategory is a category, itself with three parameters, which defines the various
operations that must be satisfied by all kinds of matrices, not just those defined by Matrix,
which defines dense matrices stored in a two-dimensional array with no special properties.

We can discover what Axiom’s current definition of the operations on a ring are.

-> )show Ring
Ring is a category constructor
Abbreviation for Ring is RING
This constructor is exposed in this frame.
Issue )edit bookvol10.2.pamphlet to see algebra source code for RING

------------------------- Operations -------------------------
?** : (%,%) -> %
?*? : (Integer,%) -> %
?*? : (NonNegativeInteger,%) -> %
?*? : (PositiveInteger,%) -> %
?*** : (%,NonNegativeInteger) -> %
?*** : (%,PositiveInteger) -> %
?+? : (%,%) -> %
??: : (%,%) -> %
?- : % -> %
?=?: (%,%) -> Boolean
1.5. CATEGORIES

In particular we have the operations of addition, subtraction and multiplication that are required to make sense of the definition of matrix with entries from $R$. In fact the actual definition of $\text{Matrix}$ is more complicated, and the category of the result is defined to be $\text{Matrix}(R:\text{Ring}): \text{MatrixCategory}(R,\text{Row},\text{Col})$ with:

- \(\text{diagonalMatrix}: \text{Vector } R \to \%\)
  \[
  \text{diagonalMatrix}(v) \text{ returns a diagonal matrix where the elements of } v \text{ appear on the diagonal.}
  \]

- if $R$ has Field then
  \(\text{inverse}: \% \to \text{Union}(\%,,\text{"failed"})\)

Note the conditional definition: the coefficients $R$ need only be a \text{Ring}, but if, in addition, they are a \text{Field}, i.e. division is possible, then it makes sense to talk about the inverse of a matrix. Of course, the inverse operation may fail if the matrix is singular, so the return domain of \text{inverse} is defined to be \text{Union}(\%,\text{"failed"}). \text{MatrixCategory} is not itself a descendant of \text{Ring}, because matrices can only be added, multiplied etc. if they conform. However, square matrices do form a ring, and Axiom knows this.

**Convention 10.** The infix binary predicate \text{has} can be used to test if domains belong to categories, or if they have specified attributes.

- \(\% \to \text{SquareMatrix}(2,\text{Integer}) \text{ has Ring}
  \]

This implies that a square matrix domain would itself be an acceptable parameter to matrix.

- \([[[[1,2],[3,4]],1,0],[0,[[5,6],[7,8]],1]]::\text{Matrix SquareMatrix}(2,\text{Integer})
  \]

- \(\% \to \text{transpose} \% \]

\[
\begin{array}{cccccc}
+1 & 2+ & +1 & 0+ & +0 & 0+ \\
+3 & 4+ & +0 & 1+ & +0 & 0+ \\
+0 & 0+ & +5 & 6+ & +1 & 0+ \\
0+ & +7 & 8+ & +0 & 1+ & \\
\end{array}
\]

Type: Matrix(SquareMatrix(2,\text{Integer}))
This is not necessarily the most stunning application of Axiom, but it does show that the type system can be used to construct some truly amazing objects. We notice also that the type system interpreted some occurrences of 1 and 0 as requiring appropriate matrices as their values, in order to make the command type-consistent.

In practice these extremely complex types are often used in the middle of a calculation, even when the final result is quite simple or straightforward. Grabmeier\[Grab91a\] gives an example from genetics, where one of the intermediate objects in his construction belonged to the domain

\[
\text{List PolynomialIdeals(Fraction Integer, DirectProduct(4,NonNegativeInteger), } [x1,x2,x3,x4], \text{DistributedMultivariatePolynomial([x1,x2,x3,x4], Fraction Integer))}
\]

\textbf{Convention 11.} Every Axiom constructor, i.e. functor or category, has an abbreviation, consisting of at most eight upper-case letters (seven in the case of categories). These serve two purposes: they can be used on input and output in order to make the names of the types shorter, and they denote the directory in which the corresponding Axiom library lives. The defaults for category \texttt{Cat}, with abbreviation \texttt{CAT}, are called \texttt{Cat&}, with abbreviation \texttt{CAT-}.

For example, the abbreviation for \texttt{Integer} is \texttt{INT}, and the compiled library lives in the directory \texttt{INT.nrlib}. Grabmeier's type can therefore also be written as

\[
\text{List IDEAL(FRAC INT,DIRPROD(4,NNI),[x1,x2,x3,x4],DMP([x1,x2,x3,x4],FRAC INT)}
\]

which is certainly shorter, even though still somewhat of a mouthful.
1.5. CATEGORIES

Using the \texttt{display} command

Let us begin with a very simple example of the \texttt{display} command.

\texttt{-> \texttt{d op pop!}}

There are 4 exposed functions called \texttt{pop!}

1. \texttt{ArrayStack(D1) -> D1 from ArrayStack(D1) if D1 has SETCAT}
2. \texttt{Dequeue(D1) -> D1 from Dequeue(D1) if D1 has SETCAT}
3. \texttt{D \texttt{\to} D1 from D if D has SKAGG(D1) and D1 has TYPE}
4. \texttt{Stack(D1) -> D1 from Stack(D1) if D1 has SETCAT}

Examples of \texttt{pop!} from \texttt{ArrayStack}

\begin{verbatim}
a:ArrayStack INT:= arrayStack [1,2,3,4,5]
pop! a
a
\end{verbatim}

Examples of \texttt{pop!} from \texttt{Dequeue}

\begin{verbatim}
a:Dequeue INT:= dequeue [1,2,3,4,5]
pop! a
a
\end{verbatim}

Examples of \texttt{pop!} from \texttt{StackAggregate}

\begin{verbatim}
a:Stack INT:= stack [1,2,3,4,5]
pop! a
a
\end{verbatim}

Examples of \texttt{pop!} from \texttt{Stack}

\begin{verbatim}
a:Stack INT:= stack [1,2,3,4,5]
pop! a
a
\end{verbatim}

There are 4 exposed functions listed, with their argument types, result type and (following the word \texttt{from}) the source of their implementation: a combination known collectively as a signature. Once one knows that \texttt{SKAGG} is an abbreviation for \texttt{StackAggregate}, which is easy enough to find out

\texttt{->\texttt{d abbrev SKAGG}}

\texttt{SKAGG abbreviates category StackAggregate}

Axiom functions have a special syntax (a tagged comment) that provides examples of functions from the algebra source code. For example, the \texttt{Stack} domain has the \texttt{pop!} function.

\begin{verbatim}
  pop_.! : % \to S
  ++ pop! returns the top element of the stack, destructively
  ++ modifying the stack to remove that element.
  ++
  ++X a:Stack INT:= stack [1,2,3,4,5]
  ++X pop! a
  ++X a
\end{verbatim}

The special form of the ++ comment, which is ++X provides an example of the use of the function. So, in the above output, we see

Examples of \texttt{pop!} from \texttt{Stack}
a:Stack INT:= stack [1,2,3,4,5]
pop! a
a

which shows three commands a user can execute to demonstrate the function

\[ \text{a:Stack INT:= stack [1,2,3,4,5]} \]

1. \[ [1,2,3,4,5] \]  
   Type: Stack(Integer)

2. \[ \text{pop! a} \]
   \[ 1 \]  
   Type: PositiveInteger

3. \[ \text{a} \]
   \[ [2,3,4,5] \]  
   Type: Stack(Integer)

making it clear that the stack \( a \) has been modified.

A more complicated example would be given by \( \text{\texttt{\textbackslash d op integrate}} \), and we will only explain some of the entries. The Axiom library has 32 exposed integrations and 10 unexposed ones.

\[ \text{\textbf{[29]} D \rightarrow D from D} \]
\[ \text{if D has UPXSCAT(D1) and D1 has RING and D1 has ALGEBRA(} \]
\[ \text{FRAC(INT))} \]

\[ \text{\textbf{[7]} (D2,Symbol) \rightarrow \text{Union}(D2,\text{List(D2)) from FunctionSpaceIntegration(} \]
\[ \text{D4,D2)} \]
\[ \text{if D4 has Join(EuclideanDomain,OrderedSet,} \]
\[ \text{CharacteristicZero,RetractableTo(Integer),} \]
\[ \text{LinearlyExplicitRingOver(Integer)) and D2 has Join(} \]
\[ \text{TranscendentalFunctionCategory,PrimitiveFunctionCategory,} \]
\[ \text{AlgebraicallyClosedFunctionSpace(D4))} \]

The function \[ \text{[29]} \] is the one we used to integrate the Puiseux series: \( \text{UPXSCAT} \) is the abbreviation for \( \text{UnivariatePuiseuxSeriesCategory} \), and \( D1 \), the coefficients of the series, must be an algebra over the rational numbers (otherwise we could not regard the exponents as coefficients, which one has to do when integrating \( x^{p/q} \) to \( \frac{q}{p+q} x^{1+p/q} \)) and a ring.

The function \[ \text{[7]} \] is the one we used for most of the other integrations we have performed so far. Although the signature looks fairly lengthy, we have already analysed the fact that the signature is of the form

\[ (D2,Symbol) \rightarrow \text{Union}(D2,\text{ListD2}) \]

The rest of the lines are merely explaining what properties \( D2 \) must have. \( D4 \) plays the role of the coefficients in the function being integrated. We always had \( \text{Integer} \) in there, but this is not strictly necessary.

\[ \text{\rightarrow 2*cos(4*x)**3::Expression RomanNumeral} \]

3

1. \[ \text{II cos(IV x)} \]  
   Type: Expression(RomanNumeral)

2. \[ \text{\rightarrow integrate(\%x,x)} \]
\[
\frac{2}{\cos(IV \ x) + II \sin(IV \ x)}
\]

Type: Union(Expression(RomanNumeral), List(Expression(RomanNumeral)))

It can clearly be seen that, in Expression RomanNumeral, the coefficients are indeed members of RomanNumeral, but the exponents are not.
Chapter 2

How does one program in the Axiom system by James H. Davenport

This is (mostly) quoted verbatim, with permission, from Davenport[Dave92b].

2.1 Introduction

Axiom can be used in essentially three ways. The first corresponds to the “pocket calculator” style of use – simple commands can be typed and the answer is printed. These commands can be issued from the keyboard in traditional style, or via the Hyperdoc menu system, or through “.input” files, and are handled by what is generally called the “Axiom interpreter”. This interpreter does more than traditional computer algebra systems do, since Axiom is a typed system, and the interpreter has to do type inference.

The second style corresponds to what might be called the “programmable pocket calculator” style, where simple functions are defined, or variables given values for later use. An example of a simple function would be

\[
\text{fac n == if n < 3 then n else n*fac(n-1)}
\]

as a definition of the factorial function. A slightly more complicated example (take from IBM[IBMx91] and Jenks & Sutor[Jenk92]) goes as follows.

\[
\text{mersenne i == 2**i - 1}
\]

This line defines a function for computing the \(i\)-th Mersenne number.

\[
\text{mersenneIndex := [n for n in 1.. | prime?(mersenne(n))]}
\]

This line, which produces the following output from Axiom,

\[
\begin{verbatim}
(2) [2,3,5,7,13,17,19,31,61,89,...]  Type: Stream(PositiveInteger)
\end{verbatim}
\]

computes an infinite (but lazily evaluated) list of the indices of those Mersenne numbers which are actually prime.
mersennePrime n == mersenne mersenneIndex(n)

This defines a function which produces the \( n \)-th Mersenne prime. It can be used as in the following input line (and corresponding output).

\[
\text{mersennePrime 5}
\]

\[
(4) \quad 8191 \quad \text{Type: PositiveInteger}
\]

In this style, we have various "one-liners" which interact with each other, much as programmed functions on a pocket calculator.

In the third style, we define new complete data types to Axiom. It is this third style of programming that this paper addresses.

## 2.2 Programming concepts

Axiom has several fundamental concepts, which we have to outline briefly.

**Domain** A domain is what many other languages would call an abstract data type, i.e. a specification of certain data objects and the operations on them. A typical domain would be Integer, whose elements are the underlying integers of the implementation (infinite precision, of course), and which supports the following operations, as given by the Axiom command \( \text{)show Integer} \), or by using the Hyperdoc browser. We remind the reader that \% is Axiom’s notation for the “current domain”, i.e. Integer in this case, and that all operations are prefix unless shown otherwise (e.g. the infix \* and quo).

Integer is a domain constructor
Abbreviation for Integer is INT
This constructor is exposed in this frame.
Issue \text{)edit bookvol10.3.pamphlet} to see algebra source code for INT

```
------------------------------- Operations --------------------------------
?*? : (%,%) -> % ?*? : (Integer,%) -> %
?*? : (NonNegativeInteger,%) -> % ?*? : (PositiveInteger,%) -> %
?**? : (%,NonNegativeInteger) -> % ?**? : (%,PositiveInteger) -> %
?+: % -> % ?-? : (%,%) -> %
?-? : % -> % ?<=?: (%,%) -> Boolean
?>?: (%,%) -> Boolean ??>?: (%,%) -> Boolean
D : % -> % D : (%,NonNegativeInteger) -> %
OMwrite : (%,Boolean) -> String OMwrite : % -> String
1 : () -> % 0 : () -> %
?~?: (%,NonNegativeInteger) -> % ?~?: (%,PositiveInteger) -> %
abs : % -> % addmod : (%,%,%) -> %
associates? : (%,%) -> Boolean base : () -> %
binomial : (%,%) -> % bit? : (%,%) -> Boolean
coerce : Integer -> % coerce : % -> %
coerce : Integer -> % coerce : % -> OutputForm
coerce : % -> String coerce : % -> DoubleFloat
coerce : % -> InputForm coerce : % -> Pattern(Integer)
convert : % -> Float convert : % -> Pattern(Integer)
convert : % -> Integer
```

2.2. PROGRAMMING CONCEPTS

It must be stressed that the use of \texttt{show} or the browser is essential to understanding what is already present in Axiom, and what one has to add to produce a valid domain. If fact, a user cannot write a domain, merely a function (see later) which, when called, will create a domain.
Category

The set of all domains (declared as) belonging to this category, i.e. having the stated operations and associated axioms. For example, the domain Integer belongs to the category Ring, which has the following operations, again given by \texttt{show Ring} or the browser.

Ring is a category constructor

Abbreviation for Ring is RING

This constructor is exposed in this frame.

Issue \texttt{edit bookvol10.2.pamphlet} to see algebra source code for RING

---

<table>
<thead>
<tr>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ast\ast\ast) : (%,%) \rightarrow %</td>
</tr>
<tr>
<td>(\ast\ast\ast) : (NonNegativeInteger,%) \rightarrow %</td>
</tr>
<tr>
<td>(\ast\ast\ast) : (PositiveInteger,%) \rightarrow %</td>
</tr>
<tr>
<td>(\ast\ast\ast\ast\ast) : (%,NonNegativeInteger) \rightarrow %</td>
</tr>
<tr>
<td>(\ast\ast\ast\ast\ast) : (%,PositiveInteger) \rightarrow %</td>
</tr>
<tr>
<td>(\ast\ast) : (%,%) \rightarrow %</td>
</tr>
<tr>
<td>(-) : % \rightarrow %</td>
</tr>
<tr>
<td>(\ast) : % \rightarrow %</td>
</tr>
<tr>
<td>1 : () \rightarrow %</td>
</tr>
<tr>
<td>(\ast\ast) : (%,NonNegativeInteger) \rightarrow %</td>
</tr>
<tr>
<td>(\ast\ast) : (%,PositiveInteger) \rightarrow %</td>
</tr>
<tr>
<td>coerce : Integer \rightarrow %</td>
</tr>
<tr>
<td>hash : % \rightarrow SingleInteger</td>
</tr>
<tr>
<td>one? : % \rightarrow Boolean</td>
</tr>
<tr>
<td>sample : () \rightarrow %</td>
</tr>
<tr>
<td>characteristic : () \rightarrow NonNegativeInteger</td>
</tr>
<tr>
<td>subtractIfCan : (%,%) \rightarrow Union(%,&quot;failed&quot;)</td>
</tr>
</tbody>
</table>

Note that 0 and 1 are nullary operations, since their actual value may well be very different in different domains belonging to the category Ring, e.g. in the ring of \(n\)-by-\(n\) square matrices, the 1 is the identity matrix, and not the matrix consisting entirely of 1.

Categories can be parameterized, as in \texttt{Algebra(R)}, where \(R\) is some \texttt{CommutativeRing}, which gives the category of all algebras over \(R\).

Functor

A function which takes arguments which are either individual objects, in which case the domain they come from is specified, or domains, in which case the category they come from is specified, and which returns a domain specified to live in a particular category. For Example:

- \texttt{Z=Integer} is the result of applying the function \texttt{Integer} to no arguments;
- \texttt{Q} is the result of applying the function \texttt{Fraction} (which requires an \texttt{IntegralDomain} as its argument) to the \texttt{IntegralDomain Integer}. The result is a Field.
- \texttt{Z[y]} is the result of applying the functor \texttt{UnivariatePolynomial} to the object \(y\) (from the domain \texttt{Symbol}) and the domain \texttt{Integer} (from the category \texttt{Ring}). The result is declared to be a \texttt{Ring}, or, more precisely, to belong to the category \texttt{UnivariatePolynomialCategory(R)}, where \(R\) is the \texttt{Ring} supplied.
2.3. A FIRST PROBLEM – WEIGHTED POLYNOMIALS

Package
Very like a function, but does not specify a new data object, merely some new functions. A typical example would be UnivariatePolynomialFunctions2, which defines the operation
\[ \text{map}:(R \to S, \text{UnivariatePolynomial}(x,R)) \to \text{UnivariatePolynomial}(y,S) \]
where \( x \) and \( y \) are elements of \( \text{Symbol} \), and \( R \) and \( S \) belong to the category \( \text{Ring} \). In mathematical terms, this function takes a function \( \theta \) from \( R \) to \( S \), and performs the corresponding function from \( R[\!\!x\!\!] \) to \( S[\!\!y\!\!] \). To get the actual function from \( R[\!\!x\!\!] \) to \( S[\!\!y\!\!] \), one would have to use Axiom’s notation for “lambda expressions”, viz. \( \text{map}(f,#1) \), i.e. that function which, given an element \( p \) of \( \text{UnivariatePolynomial}(x,R) \) computes \( \text{map}(f,p) \).

Constructor
The generic term including category, functor, and package.

2.3 A first problem – Weighted Polynomials

The problem definition

Our aim here is to emulate CAMAL’s (Fitch[Fitc74]) handling of “weighted polynomials”, a concept which is also found in Reduce (Hearn[Hear87]) via the commands \texttt{weight} and \texttt{wtlevel}. For those not familiar with the idea, we give a quick summary here. We will then develop two alternative implementations incrementally. The complete definitions will be given in appendices.

Some of the polynomial variables have a (positive integer) \texttt{weight} associated to them. If \( x \) has weight \( k \), the \( x^n \) has weight \( kn \), and the weight of a monomial is the sum of the weights of the powers in it. This means that the weight of a product of two monomials is the sum of the weights. A certain integer (the \texttt{weight level}) is chosen, and all monomials of weight exceeding this are dropped. If we call this dropping operation \( \lfloor \cdot \rfloor \) (by analogy with the rounding of integers), we see that \( \lfloor f + g \rfloor = \lfloor f \rfloor + \lfloor g \rfloor \), and that \( \lfloor fg \rfloor = \lfloor \lfloor f \rfloor \lfloor g \rfloor \rfloor \).

The outline implementation we suggest is conceptually similar to that of Reduce (Hearn[Hear87]). The weight is stored as the exponent of a virtual variable (\( k* \) in Reduce), and monomials are stored as coefficients of the appropriate power of this variable. Reduce does not ensure that \( k* \) has to be the most significant variable, in terms of polynomial ordering, and hence the truncation is not as efficient as it might be.

The key Axiom functions that we need to use are briefly explained now.

- **Polynomial** is the type Axiom assigns by default to polynomial-like objects. These types can be seen in Axiom after every object is computed (use the Axiom system command \texttt{set message type on} if they are not being shown). This is a functor, which, given a \( \text{Ring} \), returns a domain in \( \text{PolynomialCategory}(\text{Ring},\text{Symbol},\text{IndexedExponents}(\text{Symbol})) \), i.e. the variables are the elements of the \( \text{Symbol} \) domain, which corresponds to ordinary symbols, and the exponents are from \( \text{IndexedExponents} \), which gives a non-negative integer for every symbol (for which it is non-zero). Hence this type is a traditional sparse multivariate polynomial, and is Axiom’s default type. There are others, in particular dense polynomials and polynomials represented in a distributed, rather than recursive, fashion, but these do not appear unless explicitly called for.

- **PolynomialCategory** is the category of the result of Polynomial, as is show by the
browser (the Parents option on Polynomial). This category takes as arguments a Ring \( R \), an OrderedSet, known typically as VarSet, which represents the “variables” of the polynomial structure, and an OrderedAbelianMonoidSup\(^1\), known as \( E \), which represents the exponents of the polynomial structure. So a domain in the category PolynomialCategory\((R, E, \text{VarSet})\) is a representation of polynomials with variables \( \text{VarSet} \) and coefficients from \( R \), using \( E \) as the representation of the exponents.

- **PolynomialRing** is used in the implementation of Polynomial (via SparseMultivariatePolynomial, as can be found by the Lineage option of the browser). This functor takes as arguments a Ring \( R \), the ring of coefficients, and an OrderedAbelianMonoid \( E \), the set of exponents, and produces formal polynomials. In particular, if \( E \) is \( \mathbb{N} \) (the domain NonNegativeInteger in Axiom parlance), one gets the standard univariate polynomials, where no name has been given to the variable.

- **FreeModule** is used in the implementation of PolynomialRing, as can be found by using the Lineage option of the browser. This takes as arguments a Ring \( R \) and an OrderedSet \( S \), and generates the free module over \( R \) whose generators are indexed by the elements of \( S \). PolynomialRing builds on this, by keeping the definition of addition etc., but adding definitions of multiplication, relying on the addition between the exponents to define the multiplication of polynomials. Let us see precisely how this is defined (we have deleted lines redundant to the expository points we wish to make)

```plaintext
PolynomialRing(R:Ring, E:OrderedAbelianMonoid): 
FiniteAbelianMonoidRing(R, E) with 
  if R has canonicalUnitNormal then canonicalUnitNormal 
== FreeModule(R, E) add 
  Term:= Record(k:E, c:R) 
  Rep:= List Term 
  1 == [[0$E,1$R]] 
  p1, p2: % 
  if R has EntireRing then 
    p1 * p2 == 
      null p1 => 0 
      null p2 => 0 
      p1.first.k = 0 => p1.first.c * p2 
      ps = 1 => p1 
      +/[[[t1.k+t2.k, t1.c*t2.c]$Term for t2 in p2]] for t1 in reverse(p1) 
      -- This ‘reverse’ is an efficiency improvement: 
      -- reduces both time and space [Abbot/Bradford/Davenport] 
  else 
    p1 * p2 == 
      null p1 => 0 
      null p2 => 0 
      p1.first.k = 0 => p1.first.c * p2 
      p2 = 1 => p1 
      +/[[[t1.k+t2.k, r]$Term for t2 in p2] | (r:=t1.c*t2.c) ^= 0] 
```

\(^1\) An OrderedCancellationAbelianMonoid is a cancellation abelian monoid which is also a totally ordered set, such that the ordering is compatible with addition: \( x \leq y \Rightarrow x + z \leq y + z \). Since it is a cancellation abelian monoid, i.e. satisfies \( x + y = y + z \Rightarrow x = y \), there is a partial subtraction operation: \( x - y \) is the unique \( z \) such that \( z + y = x \), if it exists. An OrderedAbelianMonoidSup is an OrderedCancellationAbelianMonoid in which, in addition, there is an operation \( \text{sup} \) with respect to the partial ordering induced by subtraction. In other words \( \text{sup}(x, y) - x \) and \( \text{sup}(x, y) - y \) exist, and \( \text{sup}(x, y) \) is minimal (with respect to \(<\)) with this property.
2.3. A FIRST PROBLEM – WEIGHTED POLYNOMIALS

for t1 in reverse(p1)]
   -- This 'reverse' is an efficiency improvement:
   -- reduces both time and space [Abbot/Bradford/Davenport]

if R has CommutativeRing then
   p ** nn ==
      null p => 0
   nn = 0 => 1
   p.rest = [] => [[nn * p.first.k, p.first.c ** nn]]
   binomThmExpt([p.first],p.rest,nn)

binomThmExpt(x,y,nn) ==
   nn = 0 => 1
   ans,xn,yn:=%
   bincoef: Integer
   powl: List(%):= [x]
   for i in 2..n repeat powl:=[x * powl.first, :powl]
   yn:=y; ans:=powl.first; i:=1; bincoef:=nn
   for xn in powl.rest repeat
      ans:= bincoef * xn * yn + ans
   bincoef:= (nn-1) * bincoef quo (i+1); i:=i+1
   -- last I and BINCOEF unused
   yn:= y * yn
   ans + yn
else
   p ** nn == repeatMultExpt(p,nn)

repeatMultExpt(x,nn) ==
   nn = 0 => 1
   y:= x
   for i in 2..nn repeat y:= x * y
   y

The returned domain belongs to the category FiniteAbelianMonoidRing, with the additional property canonicalUnitNormal (see Davenport & Trager [Dave90] for an explanation of this property) if the ground ring R has this property. The implementation is to take FreeModule(R,E), and to add (hence the use of this keyword) certain additional operations – we have just quoted the definition of the unit and multiplication, in fact there are more. We find it convenient to work in terms of the internal representation of FreeModule, hence the Rep line (which in turn relies on the definition of Term). We will see further examples of this methodology later on, as method (4) for the definition of Axiom functors.

The problem specification

Proceeding in a top-down fashion, we can see that we are going to need a construction which takes as arguments a Ring R, some weights for some symbols, and an initial weight level. This will return a Ring as result, the ring of weighted polynomials, in the named symbols

---

An “abelian monoid ring” bears the obvious relationship to a “group ring”: viz. it is the set of formal sums of ring elements, indexed by elements of the abelian monoid, with addition etc. being defined component-wise, and multiplication making use of the addition of abelian monoid indices. The use of the word “finite” here is to indicate that we consider only finite sums, i.e. the ring element is zero for all but finitely many elements of the abelian monoid.
(at least), over \( R \), with the weight level as specified. At this point, certain design decisions need to be made.

- Should the weight level be changeable? In Reduce, it is, and advice from the theory of repeated approximation (see, for example, Barton & Fitch\cite{Bart72}) led us to believe that the weight level should be changeable.

- Should the weights themselves be changeable? In Reduce, they are not, and making them changeable would require the re-computation of the weights of all products. Of course, the user of Axiom is free to build two different domains with different weights assigned in each, a flexibility that is not possible in Reduce, and we believe that this should be sufficient.

- How should the weights be represented. We could produce a separate Axiom data type, we could accept them as equations, and insist at run time that they be of the form “symbol=non-negative integer”, or we could treat them as a list of symbols and a corresponding list of non-negative integers. For simplicity, we chose the last as the user interface (but see later for the internal handling).

- Should the weighted polynomials contain only the symbols specified in the weight list, or others? This is debatable, but it seemed simpler, as the implementation progressed, to allow other symbols, which then effectively have a weight of 0.

We can now probably write the specification part of this functor.

\texttt{)}abbrev domain OWP OrdinaryWeightedPolynomials

\texttt{OrdinaryWeightedPolynomials}(R:\texttt{RIng},
\quad v1:\texttt{List Symbol},
\quad wl:\texttt{List NonNegativeInteger},
\quad wtlevel:\texttt{NonNegativeInteger})::

\texttt{Ring with}

\texttt{if R has CommutativeRIng then Algebra(R)}

\texttt{coerce : \% -> Polynomial(R)}
\quad \texttt{++ coerce will convert back into a Polynomial(R), ignoring weights}

\texttt{coerce : Polynomial(R) -> \%}
\quad \texttt{++ coerce a Polynomial(R) into Weighted form,}
\quad \texttt{++ applying weights and ignoring terms}

\texttt{if R has Field then}

\texttt{"/" : (\%,\%) -> Union(\%,"failed")}
\quad \texttt{++ a / b, the division only works if minimum weight}
\quad \texttt{++ of divisor is zero, and if R is a Field}

\texttt{changeWeightLevel : NonNegativeInteger -> Void}
\quad \texttt{++ changeWeightLevel changes the weight level to the new value given:}
\quad \texttt{++ NB: previously calculated terms are not affected}

The \texttt{)}abbrev line is necessary for the definition of any functor, since the abbreviation (up to eight letters, or seven for a category) defines, among other things, the name of the directory in which the compiled code will be stored.
Axiom comments begin with either -- or ++. The former fulfil the traditional role of commenting programs. The latter, which can only appear in the appropriate contexts, are picked out by the program that builds the HyperDoc database, and can be retrieved by HyperDoc when it comes to describing operations (as in the case of the \texttt{changeWeightLevel} operation quoted above) or the whole constructors.

Axiom checks the first word of ++ comments on functions to see that it is the name of the function. If not, it complains.

In general, there should be examples given for every function, using the ++ X syntax. Comment lines starting with these three characters are displayed as help text when the user asks about a function with \texttt{display operation}.

We could now start implementing this data type, but a thought crosses our mind. While \texttt{Polynomial} is Axiom's default representation, it is not the only one, and it would be a pity for this "weighted polynomial" facility not to be available for other implementations as well. Hence we decide that we will implement \texttt{OrdinaryWeightedPolynomials} in terms of a more general constructor, which takes the polynomial type as an argument. This leads to the following implementation for the body of \texttt{OrdinaryWeightedPolynomials}.

\begin{verbatim}
== WeightedPolynomials(R,Symbol,IndexedExponents(Symbol), Polynomial(R),vl,wl,wtlevel)
This is essentially and add form in which nothing is being added: the operations of \texttt{OrdinaryWeightedPolynomials} will be precisely those of \texttt{WeightedPolynomials}.
The header of \texttt{WeightedPolynomials} now practically writes itself.
)abbrev domain WP WeightedPolynomials
WeightedPolynomials(R:RIng,VarSet: OrderedSet, E:OrderedAbelianMonoidSup, P:PolynomialCategory(R,E,Varset), vl:List Varset, wl:List NonNegativeInteger, wtlevel:NonNegativeInteger):
Ring with
  if R has CommutativeRing then Algebra(R)
  coerce : % -> P
    ++ coerce converts back into a "P", ignoring weights
  if R has Field then
    "/": (%,%) -> Union(%,"failed")
      ++ a / b division only works if minimum weight
      ++ of divisor is zero, and if R is a Field
    coerce : P -> %
      ++ coerce a "P" into Weighted form, applying weights and ignoring terms
  changeWeightLevel : NonNegativeInteger -> Void
      ++ changeWeightLevel changes the weight level to the new value given:
      ++ NB: previously calculated terms are not affected
\end{verbatim}

How are we going to implement this type? There are various possibilities for implementing a functor in Axiom.

(1) Direct re-use of another domain, as
OrdinaryWeightedPolynomials re-uses WeightedPolynomials

(2) An existing domain with new operators added by means of an add clause. The previous method can be viewed as a trivial case of this method.

(3) A new implementation, where the representation of the data objects is defined, but all operations are defined from scratch (or provided by the default definitions given in certain categories)

(4) A hybrid approach, where a domain is added to, but we also quote its representation in order to dive into its internals. This is quite common (see, for example, the definition of PolynomialRing in terms of FreeModule), but also the most dangerous, as the domain to which one adds is no longer being treated as a “black box”, but rather as something one can dive into at will. Any changes in the representation of the domain being added to can invalidate the new domain being built.

The problem implementation

Let us first try the third method, where we use PolynomialRing as our representation. The essentials of our implementation will then look as follows (the details of coerce (p36) etc. will be discussed later).

```plaintext
Rep := PolynomialRing(P,NonNegativeInteger)
w,x1,x2:%
0 == 0$Rep
1 == 1$Rep
x1 = x2 ==
   -- Note that we must strip out any terms greater than wtlevel
   while degree x1 > wtlevel repeat
      x1 := reductum x1
   while degree x2 > wtlevel repeat
      x1 := reductum x2
   x1 =$Rep x2
x1 + x2 == x1 +$Rep x2
-x1 == -$Rep x1
x1 * x2 ==
   -- Note that this is probably an extremely inefficient definition
   w:=x1 * $Rep x2
   while degree(w) > wtlevel repeat
      w:=reductum w
   w
```

One important point that crops up here is the necessity to distinguish the operations of the representation (PolynomialRing) from those of the type being defined. Since elements can be viewed as belonging to either the data type or its representation, there is a potential for ambiguity, when the data type and the representation have operations of the same signature. In this case, the unqualified operation name will refer to that of the data type, and that of the representation has to be obtained by use of the $Rep syntax – meaning use the operation
from the data type \texttt{Rep}. A trivial example of the definition of $+$ given above, which can be paraphrased in English as “to add two elements of \texttt{WeightedPolynomials}, add them as if they were elements of the representation, i.e. elements of \texttt{PolynomialRing}.” Slightly more complicated is the definition of equality, which can be paraphrased in English as “to test two elements of \texttt{WeightedPolynomials} for equality, first remove any terms of weight greater than the current weight level, then test them for equality as elements of the representation, i.e. elements of \texttt{PolynomialRing}”. It is perhaps worth noting that a side-effect of this is that calls to \texttt{changeWeightLevel} can affect the result of equality tests.

The reader may well say “\texttt{Ring} was meant to define many more operations than you have done there—where are the rest?” The answer is that these are provided by the defaulting operations in the various categories. For example, the first operation we have not defined is the multiplication operation with signature "*: (\texttt{Integer},\%) $\rightarrow$ \%". This is acquired by default from the category \texttt{AbelianGroup}, an ancestor of \texttt{Ring}, where the operation is defined by

\texttt{AbelianGroup() : Category == SIG where}

\begin{verbatim}
SIG ==> CancellationAbelianMonoid with

  "*" : (\texttt{Integer},\%) $\rightarrow$ \%
  ++ n*x is the product of x by the integer n.
\end{verbatim}

The \texttt{PolynomialRing} implementation

Here we use \texttt{PolynomialRing} as our base type, as well as our representation, in what corresponds to method (4) of the choice outlined earlier. Again, we have left the various definitions of \texttt{coerce} (p36) etc. for later consideration: we focus here on the differences between this implementation and the previous one.

\begin{verbatim}
== PolynomialRing(P,NonNegativeInteger)
add
  --representations
  Term := Record(k:NonNegativeInteger,c:P)
  Rep := List Term
  w,x1,x2:%

  x1 * x2 ==
  null x1 => 0
  null x2 => 0
  r:P
  x1.first.k = 0 =>
    [[t2.k,r]$Term for t2 in x2 | (r:=x1.first.c * t2.c) ^= 0]
  x2 = 1 => x1
  +/[[[n,r]$Term for t2 in x2 | (n:=t1.k+t2.k) <= wtlevel and
                           (r:=t1.c*t2.c) ^= 0]
        for t1 in reverse(x1)]
  -- This 'reverse' is an efficiency improvement:  
  -- reduces both time and space [Abbott/Bradford/Davenport]

  import RepeatedSquaring(%)

  x:% ** n:NonNegativeInteger ==
\end{verbatim}
zero? n => 1
    expt(x,n pretend PositiveInteger)

We still need a definition of equality, since the definition from PolynomialRing is not adequate, as it does not take account of the current value of the weight level. While the algorithm is very similar, the implementation has to be in terms of the newly-defined Rep, which is a list of terms. Hence degree(x1) is replaced by x1.first.k. Similarly, the construction *$Rep does not work, since the Rep is now just a list of objects, and has to be replaced by an explicit copy of the definition of equality from PolynomialRing, which is in fact inherited from IndexedDirectProductAbelianGroup.

We no longer need definitions of 0 and 1, which are picked up from PolynomialRing, nor definitions of addition and subtraction. We do, however, need a definition of multiplication, since the definition in PolynomialRing does not drop terms greater than the weight level. This definition is based on that given earlier for multiplication in PolynomialRing.

In addition, we now need a definition of exponentiation. The reason for this is related to one of the major stumbling-blocks people find when programming in Axiom, so we shall analyse it carefully. We have already said that it is not necessary to provide all the definitions required for a data type, as they could be picked up from defaulting packages. When methodologies (2) or (4) are used, there are in fact two places where such missing definitions could be picked up from: the defaulting packages or the so-called add chain – the functor which is quoted in the add clause, or, recursively, the functor which is quoted in its add clause, and so on. Which should we use? The rule in Axiom is quite simple, though its implications are profound.

**Principle 9:** A function is first searched for in the implementation of a given functor, then recursively up the add chain, without examining defaulting packages. If this fails to find a definition, then the defaulting packages are searched, from most specific to most general.

The implications of this rule for exponentiation are as follows.

(i) There is a default definition of exponentiation in Monoid, and hence in Ring, which works by repeated squaring. This definition would be perfectly adequate for our use (using multiplication we have just defined in WeightedPolynomials).

(ii) There are other definitions of exponentiation in PolynomialRing, as we have seen earlier, which use the binomial theorem if the coefficient ring is commutative, and a repeated multiplication algorithm otherwise.

(iii) Therefore, by the rule quoted above, it is one of the definitions in (ii) which will be used. Hence, they will use the definition of multiplication defined in PolynomialRing, and so will not take advantage of the weight level.

Hence, in order to get a satisfactory implementation of exponentiation, we need to repeat the defaulting definition, or provide some definition that will use the multiplication of WeightedPolynomials.

A related issue comes up in the definitions of zero? and one?. These are defined, in AbelianMonoid and Monoid respectively, to have defaulting definitions zero? x => x=0 and one? x => x=1. Since these definitions happen not to be over-ridden in the add chain, they are the definitions that apply in WeightedPolynomials, and so use WeightedPolynomials’ definition of equality.

However, were a later author of PolynomialRing to add other definitions, these would be picked up instead.
2.3. A FIRST PROBLEM – WEIGHTED POLYNOMIALS

Miscellaneous definitions

Here we give some miscellaneous definitions that should form part of the implementation of WeightedPolynomials. The first three lines deal with the definition of changeWeightLevel. 

\[ \text{n:NonNegativeInteger} \]
\[ \text{changeWeightLevel(n)} == \]
\[ \text{wtlevel:=n} \]

We had earlier decide to represent the weights by a list of variables and a corresponding list of weights, but this is rather clumsy for internal manipulation. Hence the next few lines define an internal data structure called lookupList, initialize it, and provide a local function (i.e. one not usable outside the body of the functor) for looking up the weight attached to a particular variable.

\[ \text{lookupList: List Record(var:VarSet, weight:NonNegativeInteger)} \]
\[ \text{if #vl ^= #wl then error "incompatible length lists in WeightedPolynomial"} \]
\[ \text{lookupList:=[[v,n] for v in vl for n in wl]} \]
\[ \text{lookup:Varset -> NonNegativeInteger} \]
\[ \text{lookup v ==} \]
\[ \text{l:=lookupList} \]
\[ \text{while l ^= [] repeat} \]
\[ \text{v = l.first.var => return l.first.weight} \]
\[ \text{l:=l.rest} \]
\[ 0 \]

We now have to have some method of creating elements of the domain WeightedPolynomials. The obvious way is to provide a coercion operator from \( P \) (which in the case of OrdinaryWeightedPolynomials will be the usual type Polynomial of Axiom) to WeightedPolynomials. This is the function of the next few lines. \text{coerce} itself is simple: it just calls \text{innercoerce}, passing it the weight level. \text{innercoerce} recursively deconstructs the input polynomial, decreasing the weight level as appropriate.

\[ \text{p:P} \]
\[ \text{z:Integer} \]
\[ \text{innercoerce:(p,z) -> %} \]
\[ \text{innercoerc(p,z) ==} \]
\[ \text{z < 0 => 0} \]
\[ \text{zero? p => 0} \]
\[ \text{mv:= mainVariable p} \]
\[ \text{mv case "failed" => [[0,p]]} \]
\[ \text{n:=lookup(mv)} \]
\[ \text{up:=univariate(p,mv)} \]
\[ \text{ans:%} \]
\[ \text{ans:=0} \]
\[ \text{while not zero? up repeat} \]
\[ \text{d:=degree up} \]
\[ \text{f:=n*d} \]
\[ \text{lcup:=leadingCoefficient up} \]
\[ \text{up:=up-leadingMonomial up} \]
\[ \text{mon:=monomial(1,mv,d)} \]
f <= z => ans:=ans+[[tm.k+f,mon*tm.c] for tm in innercoerce(lcup,z-f)]

ans

coerce(p):% == innercoerce(p,wtlevel)

The inverse operation is much simpler: we have merely to add up the coefficients.

coerce(w):P == "+"/[tm.c for tm in w]

The last definition is that of coercion from \texttt{WeightedPolynomials} into \texttt{OutputForm} – Axiom’s type for output (and conversion to TeX etc.). This is fairly simple: the main complexity is in the specification. Here we have decided that a single term of zero weight will print as such, but that otherwise each group of terms of a particular weight will be printed parenthesised (even if there is only one term of that weight). Clearly, it would be possible to adapt this definition to almost any other desired behaviour.

coerce(p:%):OutputForm ==
zero? p => (0$Integer)::OutputForm
p.first.k = 0 => p.first.c::OutputForm
reduce("+",(reverse [paren(t1.c::OutputForm) for t1 in p])::List OutputForm

2.4 A second problem – FourierSeries

The problem definition

Our aim here is to implement an equivalent of CAMAL’s (Fitch[Fitc74]) handling of truncated Fourier series. We have some domain of “angles” – in CAMAL’s case linear combinations with integer coefficients (lying in the range $-63 \ldots 63$) of the eight angular variables $s, \ldots, z$. We can build sin or cos of these variables, and use them as coefficients in polynomial expressions, where products of trigonometric functions are always linearised. There are more operations provided in CAMAL, e.g. integration with respect to an angular variable, but we will not bother with these for simplicity of exposition.

Within CAMAL, the coefficients of expressions in these trigonometric functions will therefore not involve other trigonometric functions, but will involved weighted polynomials. These we have already defined, and there seems no absolute need to use weighted polynomials, though they are in practice the most common type of coefficient required. However, we probably need to assume that the coefficients commute with each other and with the trigonometric terms, since otherwise the linearisation of products is not well-defined. Furthermore, since

$$\sin(A) \sin(B) = \frac{\cos(A - B) - \cos(A + B)}{2}$$

we must be able to divide by two. For simplicity, therefore, we insist on the ability to divide by any non-zero integer, i.e. that the coefficients should be an Algebra over \(\mathbb{Q}\), the Axiom type \texttt{Fraction Integer}.

The problem specification

The header of our type Fourier Series now nearly writes itself. The arguments of the trigonometric functions had better be an ordered set, so that we can order the various trigonometric functions, and an abelian group so that the addition and subtraction rules can take place.
2.4. A SECOND PROBLEM – FOURIERSERIES

It would therefore be possible to require that the domain of these arguments should be an
OrderedAbelianGroup, but this may be too strong, and we will restrict ourselves to insisting on
Join(OrderedSet,AbelianGroup)\(^3\)

FourierSeries(R:Join(CommutativeRing,Algebra(Fraction Integer)),
E:Join(OrderedSet,AbelianGroup)):

\[
\text{Algebra}(R) \text{ with }
\]

if E has canonical and R has canonical then canonical

\[
\text{coerce : } R \rightarrow \%
\]
++ coerce converts coefficients into Fourier Series

\[
\text{coerce : FourierComponents}(E) \rightarrow \%
\]
++ coerce converts sin/cos terms into Fourier Series

\[
\text{makeSin : } (E,R) \rightarrow \%
\]
++ makeSin makes a sin expression with given argument and coefficient

\[
\text{makeCos : } (E,R) \rightarrow \%
\]
++ makeCos makes a cos expression with given argument and coefficient

The operations here (with the exception of the last \text{coerce} (p39), will be explained in the
next section, are pretty obvious. What about the line containing the word \textit{canonical}? Axiom’s
definition of the attribute \textit{canonical} is that a domain is canonical if mathematical
equality implies equality of data structure. In particular, it authorises the use of hash-based
techniques. There is a discussion in Davenport & Trager[Dave90] and more detail is available
in Davenport et al.[Dave88]. In our case, we are saying that, if the coefficients and arguments
are canonical, then the data type returned will also be.

The obvious implementation of this is via some kind of \texttt{FreeModule}, using \(R\) as the coeffi-
cients and the trigonometric functions as the indices. However, we first need to define the
trigonometric functions themselves, and this is the purpose of the next section. We will return
to the type \texttt{FourierSeries} in the section following.

The FourierComponent implementation

It would be possible to use Axiom’s general-purpose type \texttt{Expression} to represent trigono-
metric functions, but we settled, for pedagogic reasons and partly to keep our code reasonably
self-contained, on a separate data type.

The requirements on this data type are quite straight-forward. It should provide ways of
making sin and cos functions, and the result should be an \texttt{OrderedSet} so that it can be
passed to \texttt{FreeModule}. The header is then equally straight-forward.

FourierComponent(E:OrderedSet):

\[
\text{OrderedSet with }
\]

\[
\text{sin : } E \rightarrow \%
\]
++ sin makes a sin kernel for use in Fourier series

\(^3\) An \texttt{OrderedAbelianGroup} would also have the property that \(a < b \Rightarrow a + c < b + c\), but we probably do not need this
CHAPTER 2. HOW DOES ONE PROGRAM IN THE AXIOM SYSTEM BY JAMES H. DAVENPORT

\[
\cos : E \rightarrow \% \\
++ \cos \text{ makes a cos kernel for use in Fourier series}
\]

\[
\sin? : \% \rightarrow \text{Boolean} \\
++ \sin? \text{ is true if term is a sin, otherwise false}
\]

\[
\text{argument} : \% \rightarrow E \\
++ \text{argument returns the argument of a given sin/cos expression}
\]

Here method (3) seems an appropriate way of defining the data type – all we need store is the argument and a flag indicating whether we have a sin or cos expression. The first part of the implementation is trivial.

\[
\begin{align*}
\text{add} \\
\text{--represenations} \\
\text{Rep} := \text{Record(SinIfTrue:Boolean, arg:E)} \\
e : E \\
x, y : \% \\
\sin e &= [\text{true}, e] \\
\cos e &= [\text{false}, e] \\
\sin? x &= x.\text{SinIfTrue} \\
\text{argument} x &= x.\text{arg}
\end{align*}
\]

The harder question is the order to be imposed on FourierComponent. We have chosen, for no very good reason, to use the order of the arguments, and break ties by sorting \(\cos a\) as less than \(\sin a\). Clearly this definition could be adapted to any other strategy.

\[
\begin{align*}
x < y &= \\
x.\text{arg} < y.\text{arg} &\Rightarrow \text{true} \\
y.\text{arg} < x.\text{arg} &\Rightarrow \text{false} \\
x.\text{SinIfTrue} &\Rightarrow \text{false} \\
y.\text{SinIfTrue}
\end{align*}
\]

The last task of this method of printing the results – again this is achieved by means of a conversion to OutputForm. We have used the constructor \texttt{bracket}, which places the argument in square brackets, in order to distinguish these elements from the ordinary Expression constructions of Axiom.

\[
\begin{align*}
\text{coerce}(x):\text{OutputForm} &= \\
\text{hconcat}(\text{if x.\text{SinIfTrue} then "sin" else "cos")::OutputForm,} \\
\text{bracket((x.\text{arg})::OutputForm)})
\end{align*}
\]

The FourierSeries implementation

Now that we have FourierComponent, we can define FourierSeries. We chose again to use method (4), basing the definition on FreeModule\(\langle R,\text{FourierComponent}(E)\rangle\). Hence the start of the definition looks as follows.

\[
\begin{align*}
\text{== FreeModule}\langle R,\text{FourierComponent}(E)\rangle \text{ add} \\
\text{-- represenations}
\end{align*}
\]
A SECOND PROBLEM – FOURIER SERIES

Term := Record(k:FourierComponent(E),c:R)
Rep := List Term
multiply : (Term,Term) -> %
w,x1,x2 : %
t1,t2 : Term
n : NonNegativeInteger
z : Integer
e : FourierComponent(E)
a : E
r : R

multiply (p39) is a local function, to be defined later, which will multiply two terms. The result may well not be a single term, due to linearisation, but is an element of the FourierSeries domain. We know that \( \cos 0 = 1 \) and \( \sin 0 = 0 \). Furthermore, in order to ensure the ‘‘canonical’’ part, we must be careful about trigonometric functions with negative arguments (the concept of ‘‘negative’’ makes sense: an element is negative if it is less than 0). The following definitions help implement this policy.

1 == [[\cos 0,1]]

coerce e ==
    sin? e and zero? argument e => 0
    if argument e < 0 then
        not sin? e => e:=\cos(- argument e)
        return [[\sin(- argument e),-1]]
    [[e,1]]

makeCos(a,r) ==
a < 0 => [[\cos(-a),r]]
[[\cos a,r]]

makeSin(a,r) ==
zero? a => []
a < 0 => [[\sin(-a),-r]]
[[\sin a,r]]

The operations of addition and subtraction, as well as multiplication by elements of \( R \), are all well-inherited from FreeModule. We do however have to define multiplication of two Fourier series, and this is done below.

multiply(t1,t2) ==
r:=(t1.c*t2.c)*(1/2)
s1:=argument t1.k
s2:=argument t2.k
sum:=s1+s2
diff:=s1-s2
sin? t1.k =>
sin? t2.k =>
    makeCos(diff,r)+makeCos(sum,-r)
    makeSin(sum,r) + makeSin(diff,r)
sin? t2.k =>
    makeSin(sum,r) + makeSin(diff,r)
makeCos(diff,r) + makeCos(sum,r)
\[
x_1 + x_2 ==
\text{null } x_1 \Rightarrow 0
\text{null } x_2 \Rightarrow 0
+\sum[\text{multiply}(t_1, t_2) \text{ for } t_2 \text{ in } x_2] \text{ for } t_1 \text{ in } x_1
\]
Chapter 3

Axiom and Category Theory

3.1 Covariance and Contravariance

Axiom has an order relation between types. The types can be in one of five possible relationships.

A type can be more general than another type. For example, Integer is more general than PositiveInteger.

A type can be more specific than another type. Conversely PositiveInteger is more specific than Integer.

A type can be equal to another type.

A type can be converted or coerced to another type. For example, Fraction(Polynomial(Integer)) can be coerced to Polynomial(Fraction(Integer)).

A type can be unrelated to another type. String and Expression are not related.

Covariance is converting from a wider type to a narrower type. For instance, converting from Matrix(Float) to Matrix(Integer).

Contravariance is converting from a narrower type to a wider type. For instance, converting from Matrix(Integer) to Matrix(Float).

Invariance means that one type cannot convert to another. For instance, a Matrix(Float) which contains numbers which cannot be represented as Integers cannot be converted to a Matrix(Integer).

These facts form an order relation, which by definition is reflexive, transitive and antisymmetric.

Reflexive means that Integer = Integer.

Transitive means that PositiveInteger < Integer < Float implies that PositiveInteger < Float.

Antisymmetric means that PositiveInteger < Float implies not(Float < PositiveInteger).
CHAPTER 3. AXIOM AND CATEGORY THEORY

3.2 Axiom Type Lattice

The types in Axiom form a lattice based on the order relationship. It is a lattice because Axiom supports multiple inheritance.

References of interest include:

Michael Barr and Charles Wells ‘‘Category Theory for Computing Science’’ 1998

Saunders Mac Lane ‘‘Categories for the Working Mathematician’’

Steve Awodey ‘‘Category Theory’’
ftp://sumin.in.ua/Books/DVD-021/Awodey_S._Category_Theory(en)(305s).pdf

‘‘Introduction to Category Theory’’
www.youtube.com/watch?v=eu0rj5C2OtG

Luca Cardelli and Peter Wegner ‘‘On understanding types, data abstraction and
luccardelli.name/Papers/OnUnderstanding.A4.pdf

staffwww.dcs.shef.ac.uk/people/A.Simons/research/reports/addaxiom.pdf

Dana Scott ‘‘Data Types as Lattices’’
www.cs.ox.ac.uk/files/3287/PRG05.pdf

Roland Backhouse and Marcel Bijsterveld ‘‘Category Theory as Coherently Constructive
Lattice Theory’’ November 1994

3.3 Terms to Understand

Suppose we wish to join Complex with Polynomial(Integer). What would elements
of this combination look like?

The union of the two is a co-product of topological spaces.
The simple combination is not simply adding elements since

\[ i + x^2 \]

is not a valid combination.

We need the algebraic co-product, known as the tensor product. We end up with
a domain of Complex(Polynomial(Integer)).

\[ \rightarrow a: \text{Complex(POLY(INT))} : = \%i+3*\text{x} \]

\[ 3x + \%i \quad \text{Type: Complex(Polynomial(Integer))} \]

\[ \rightarrow a: \text{POLY(COMPLEX(INT))} \]

\[ 3x + \%i \quad \text{Type: Polynomial(Complex(Integer))} \]
3.4 Category Definition

A category has four parts. We need a set of objects, usually represented as dots. We need a set of arrows (maps, morphisms), from dot to dot. We need a way to compose arrows in an associative manner. We need an identity arrow from a dot to itself.

The set of all arrows from dot A to dot B is written as $\text{Hom}_c(A,B)$ or, sometimes $C(A,B)$. Notice that the set $C(A,B)$ is disjoint from $C(A,D)$ since each arrow has a unique domain and co-domain.

For the example of the category Set, the objects are sets and the arrows are functions between sets. For the category Ring, the objects are rings and the arrows are ring homomorphisms. Similarly for the category Group, the dots are groups and the arrows are group homomorphisms. For a fixed Ring R, the category R-Mod has dots which are left R-modules and the arrows are R-module homomorphisms. We can also look at the category Mod-R which has dots of right R-modules and arrows which are R-module homomorphisms. For the category K, if K is a field, the dots are K-vector spaces and the arrows are K-linear transformations.

In Axiom the dots are Types (such as Integer or Character) and the arrows are functions between them with signature:

\[ f : \text{Integer} \rightarrow \text{Character} \]

Relations between categories is called a functor. A functor $F$ takes things in category $C$ into things in category $D$. We need a function on objects which maps objects of $C$ to objects of $D$. We need a function on arrows which take arrows of $C$ to arrows of $D$.

The categories $C$ and $D$ well defined structure. They have a domain and co-domain of arrows. They have identity arrows. There is a rule of composition of arrows. These form commutative diagrams.

First we have to make sure the functor $F$ maintains the domain and co-domain structure of $C$. When we apply functor $F$ to $C$ we need to preserve all of the structure so $F$ has to be defined on all of these properties. If we look at two dots in category C and a function $f$ which is an arrow in C

\[ f \]

\[ A \rightarrow B \]

then the functor $F$ has to operate on everything so we get:

\[ Ff \]

\[ FA \rightarrow FB \]

This means that if $\text{dom}$ is the domain function in $C$ then the functor $F$ commutes with $\text{dom}$. That is, applying $F(\text{dom}(f)) = \text{dom}(F(f))$.

Next we have to make sure the functor $F$ maintains the identity arrow of $C$. From the above we know that $F(\text{identity}(x)) = \text{identity}(F(x))$.

Finally we have to make sure that the rule for composition of arrows in $C$ is preserved. So the functor $F$ has to make sure that what composes in $C$ also composes with the same diagram in $D$.

Some standard functors are the identity functor $1_c$ which just maps $C$ to $C$. We can form a functor which forgets properties so that the category Group could map
to its underlying set. We can lift a category by forgetting properties, for example, lifting the category of Abelian Group $C$ to Group $D$ by ‘forgetting’ the commutative property of $C$. Similarly the category Ring or the category Module can be mapped to the underlying Abelian Group. There is also the Constant functor which maps all of the dots in $C$ to a single dot in $D$ and all of the arrows in $C$ to the identity arrow in $D$.

The category $\text{CommutativeRing} \, R$ can be mapped to a Group with the functor $GL_n$ which is the group of invertible $N \times N$ matrices with entries in the $\text{CommutativeRing} \, R$.

### 3.5 Monoids and Groups

Given a single element set and a set of arrows from that element to itself we know from the associative property that $(fg)h = f(gh)$ and from the identity property that $ef = f = fe$.

A 1-object category is a monoid. A 1-object category where all of the arrows are invertible is a group.

If we restrict the category so there is at most one arrow between any two objects in the set then we have an ordered set.

A functor $F$ from category $C$ to category $D$ consists of

- object function takes objects of $C$ to objects of $D$
- arrow function takes arrows of $C$ to arrows of $D$

Structurally we have 3 things to preserve.

- domains and co-domains of arrows. In order to preserve structure the functor $F$ has to commute with the domain and co-domain functions. That is, $F(\text{dom}(f)) = \text{dom}(F(f))$ and $F(\text{co-dom}(f)) = \text{co-dom}(F(f))$.
- identity arrows. The functor $F$ must preserve identity so $F(id(x)) = id(F(x))$.
- composition properties of arrows. The functor $F$ must take commuting diagrams to commuting diagrams.
Chapter 4

Axiom Implementation Details

4.1 Makefile

This book is actually a literate program\cite{Knut92} and can contain executable source code. In particular, the Makefile for this book is part of the source of the book and is included below.
Chapter 5

Writing Spad Code

5.1 The Description: label and the )describe command

The describe command will print out the comments associated with Axiom source
code elements. For the category, domain, and package sections the text is taken
from the Description: keyword.

This information is stored in a database and can be queried with
 )lisp (getdatabase '|Integer| 'documentation)
for the Integer domain. However, this information has other uses in the system
so it contains tags and control information. Most tags are removed by the describe
function since the output is intended to be displayed in ASCII on the terminal.

The Description: keyword is in the comment block just after the abbreviation
command. It is freeform and the paragraph will be reflowed automatically to allow
for about 60 characters per line, adjusted for spaces. The Description: section
should be written after the keyword in the ‘‘++’’ comments as in:

 )abbrev package D03AGNT d03AgentsPackage
++ Description:
++ This package does some interesting stuff. We can write multiple
++ lines but they should all line up with the first character of
++ the Description keyword. Special \spad{terms} will be removed.
++
++ The above line will force a newline. So will ending a line with \br
++ \tab{5}This will allow primitive formatting\br
++ \tab{5}So you can align text\br
++ \tab{10}Start in column 11\tab{5}and skip 5 spaces\br
++ \tab{10}End in column 11\tab{7}and count out the needed spaces\br
++ \tab{5}note that the last line will not need the br command

As the comment says, the Description should all be aligned under the ‘‘D’’ in
Description. You can indent using \tab{n} which will insert n spaces. You can
force a newline in two ways. Either include a blank line (with the ‘‘++’’ comments)
or use the \br keyword.

Due to lousy parsing algorithms for comments there are various ways this can all
go wrong.
There should not be any macros between the Description: section and the beginning
of the definition. This is wrong. It will cause the
\)
describe package d03AgentsPackage
to give the wrong output because it does not find the end of the description section
properly.
\)abbrev package D03AGNT d03AgentsPackage
++ Description:
++ This description does not work

LEDF ==> List Expression DoubleFloat
d03AgentsPackage(): E == I where

In the Description: section the \tab{nn} function will be transformed into nn
spaces. If you end each line with a \br you can control alignment.
++ Description:
++ This is an example of a table alignment\br
++ \tab{5}First Item\tab{5} This will line up with the following line\br
++ \tab{5}Second Item\tab{4} This will line up with the following line\br
++ \tab{5}Third Item\tab{5} This will line up with the following line

If the main body of the category, domain, or package begins with properties rather
than functions the Description will be incorrectly recorded. This is a known
bug finding the end of the Description section. For instance, this
++ Description:
++ The category of Lie Algebras.
++ It is used by the domains of non-commutative algebra,
++ LiePolynomial and XPBWPolynomial.

LieAlgebra(R: CommutativeRing): Category == Module(R) with

NullSquare
  ++ \axiom{NullSquare} means that \axiom{[x,x] = 0} holds.
JacobiIdentity
  ++ \axiom{JacobiIdentity} means that
  ++ \axiom{[x,[y,z]]+[y,[z,x]]+[z,[x,y]] = 0} holds.
construct: ($,$) -> $
  ++ \axiom{construct(x,y)} returns the Lie bracket of \axiom{x}
  ++ and \axiom{y}.

will give the output

{JacobiIdentity} means that \{x,[y,z]]+[y,[z,x]]+[z,[x,y]] = 0 holds.

but reordering it to read:
++ Description:
++ The category of Lie Algebras.
++ It is used by the domains of non-commutative algebra,
++ LiePolynomial and XPBWPolynomial.

LieAlgebra(R: CommutativeRing): Category == Module(R) with

NullSquare
  ++ \axiom{NullSquare} means that \axiom{[x,x] = 0} holds.
JacobiIdentity
  ++ \axiom{JacobiIdentity} means that
  ++ \axiom{[x,[y,z]]+[y,[z,x]]+[z,[x,y]] = 0} holds.
construct: ($,$) -> $
  ++ \axiom{construct(x,y)} returns the Lie bracket of \axiom{x}
  ++ and \axiom{y}.
++ \texttt{NullSquare} means that \texttt{[x,x] = 0} holds.
++ \texttt{JacobiIdentity} means that
++ \texttt{[x,[y,z]]+[y,[z,x]]+[z,[x,y]] = 0} holds.

will give the output

The category of Lie Algebras. It is used by the domains of non-commutative algebra, LiePolynomial and XPBWPolynomial. which is correct.
Chapter 6

Writing test cases
Appendix A

The Principles of Axiom

**Principle 1.** AXIOM has an interpreter for interactive use, much like any other system, and a compiler for creating new user-defined data types. The compiler emphasises strict type-checking, whilst the interpreter is more oriented towards ease of use.

**Principle 2.** Every internal Axiom data object belongs to one and only one domain.

**Principle 3.** Values can freely move from sub-domains to larger ones, and, in the interpreter only, in the other direction, provided that this conversion is legitimate.

**Principle 4.** The interpreter is responsible for performing any chain of coercions necessary to understand the user’s intentions, or when required to do so by an explicit use of ::. The compiler will perform a chain of coercions when instructed to do so by the :: operator in compiled code.

**Principle 5.** Any set of Axiom domains $D_1, \ldots, D_n$ can be combined into a (disjoint) union domain, denoted $\text{Union}(D_1, \ldots, D_n)$. The $D_i$ are called the branches of the union. The operations available on this union domain are:

- equality – two elements are equal if they come from the same branch and are equal in that branch;
- coercion to OutputForm;
- coercion from each $D_i$ to the union domain;
- coercion to each $D_i$ from the union domain, which may fail if the union object is not in the correct branch;
- an in predicate case, for testing if the union object actually is in a particular branch or not.

These union domains correspond to what some other languages call “sum types”. A particularly useful case is exemplified by the “exact quotient” operation on Integer: its return type is $\text{Union(Integer, "failed"}$, where the special token failed is returned if the division is not exact.

**Principle 6.** The Axiom library declares a family of second-order types, known as categories. The categories are arranged in a directed acyclic graph, and each domain belong to a specific category, and to all the ancestors of that category. The specification of a category includes
• all its direct ancestors,
• any additional operations that this category supports, and
• any additional axioms that the operations must satisfy.

The operation Join is used to construct new categories.

Principle 7. Categories can introduce default definitions of operations, which will take effect in any domain belonging to that category unless overridden by a definition in that domain, or in a more specific category.

Principle 8. The functors of Axiom are strongly typed: each parameter which is an Axiom object is specified to come from a particular domain; each parameter which is an Axiom domain is specified to belong to a particular Axiom category. Similarly, the domain returned by a particular functor is specified to belong to a particular category. All construction of domains must satisfy these constraints on the functors.

Principle 9. A function is first searched for in the implementation of a given functor, then recursively up the add chain, without examining defaulting packages. If this fails to find a definition, then the defaulting packages are searched, from most specific to most general.
Appendix B

The Axiom Conventions

Convention 1. Juxtaposition corresponds to (unary) function application.

Convention 2 (borrowed from APL). All system commands, i.e. those that do not perform, or affect the performance of, algebraic operations, begin with ). In general, they may be contracted as far as is unambiguous, so that )set message type on can be contracted as far as )set msg type on.

Convention 3. The symbol % refers to the most recently computed proper value (i.e. not of the Void domain). %%(n), or %%n, refers to the value numbered n, if n is a positive integer. If n is a negative integer, %%(n) refers to the value of the |n|th previous step. Also, %pi refers to π, %e to e ≈ 2.718281828 and %i to $\sqrt{-1}$

Convention 4 (a convention of the library, rather than of the kernel). Parentheses – () – are used for grouping and function application, brackets – [] – are used for constructing lists, and braces – {} – are used for constructing sets.

Convention 5. The :: in x operator, used as

Axiom object :: Axiom domain

can be used to convert the object to lie in the specified domain.

Convention 6 (Of the library authors). The notation

list of variables + -> expression

defines an anonymous function of those variables. It corresponds to the lambda-calculus expression “\text{variables.expression}”.

Convention 7. The names of Axiom functions are either special symbols (such as +) or complete English words strung together. In this case, every word after the first is capitalised. Thus integrate but complexIntegrate. In addition:

- all boolean predicates end in a ? , as in odd?, which tests if a number is odd
- all destructive functions which operate on data structures end in a ! , as in reverse!
  which reverses a list destructively.

Conversely, the names of domains (and other constructors we will come to later) consist of English words strung together, all of which are capitalised, as in IntegerMod or UnivariatePuiseuxSeries.
Convention 8. Whenever a category, or domain, is being discussed in Axiom, the symbol % stands for the domain in question, or for any domain from the category in question.

Convention 9. Axiom comments can be introduced by --- or ++. Those beginning ++ are intended for the user, and can be retrieved by the on-line help system.

Convention 10. The infix binary predicate has can be used to test if domains belong to categories, or if they have specified attributes.

Convention 11. Every Axiom constructor, i.e. functor or category, has an abbreviation, consisting of at most eight upper-case letters (seven in the case of categories). These serve two purposes: they can be used on input and output in order to make the names of the types shorter, and they denote the directory in which the corresponding Axiom library lives. The defaults for category Cat, with abbreviation CAT, are called Cat&, with abbreviation CAT−.
Appendix C

Example Code

C.1 domain WP WeightedPolynomials

— domain WP WeightedPolynomials —

)abbrev domain WP WeightedPolynomials
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Late Updated: 13 July 2016 by Tim Daly
++ Basic Functions: Ring, changeWeightLevel
++ Related Constructors: PolynomialRing
++ Also See: OrdinaryWeightedPolynomials
++ AMS Classificaitons:
++ Keywords:
++ References:
++ Description:
++ 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
WeightedPolynomials(R,VarSet,E,P,vl,wl,wtlevel) : SIG == CODE where
R : Ring
Varset : OrderedSet
E : OrderedAbelianMonoidSup
P : PolynomialCategory(R,E,Varset)
v1 : List Varset
wl : List NonNegativeInteger
wtlevel : NonNegativeInteger

SIG => Ring with

    if R has CommutativeRing then Algebra(R)

    coerce : % -> P
    ++ coerce converts back into "P", ignoring weights
if R has Field then

"/" : (%,%) -> Union(%,"failed")

coerce : P -> %
++ coerce a "P" into Weighted form, applying weights and ignoring terms

changeWeightLevel : NonNegativeInteger -> Void
++ changeWeightLevel changes the weight level to the new value given:
++ NB: previously calculated terms are not affected

CODE ==> add

-- representations
Rep := PolynomialRIng(P,NonNegativeInteger)
p : P
w,x1,x2 : %
n : NonNegativeInteger
z : Integer

changeWeightLevel(n) ==
  wtlevel := n

lookupList : List Record(var:Varset, weight:NonNegativeInteger)
if #vl ^= #wl then error "incompatible length lists in WeightedPolynomial"
lookupList := [[v,n] for v in vl for n in wl]

-- local operations

lookup : Varset -> NonNegativeInteger
lookup v ==
  l := lookupList
  while l ^= [] repeat
    v = l.first.var => return l.first.weight
    l := l.rest
  0

innercoerce : (p,z) -> %
innercoerce(p,z) ==
z < 0 => 0
zero? p => 0
mv := mainVariable p
mv case "failed" => monomial(p,0)
n := lookup(mv)
up := univariate(p,mv)
ans : %
ans := 0
while not zero? up repeat
  d := degree up
  f := n+d
  lcup := leadingCoefficient up
C.2. Domain OWP OrdinaryWeightedPolynomials

\[
\begin{align*}
\text{coerce}(p):% & = \text{innercoerce}(p,\text{wtlevel}) \\
\text{coerce}(w):P & = \"+\"/[c \text{ for } c \text{ in } \text{coefficients } w] \\
\text{coerce}(p:%):\text{OutputForm} & = \\
& \text{zero? } p \Rightarrow (0$\text{Integer}$)::\text{OutputForm} \\
& \text{degree } p = 0 \Rightarrow \text{leadingCoefficient}(p)::\text{OutputForm} \\
& \text{reduce}(\"+\",(\text{reverse } [\text{paren}(c::\text{OutputForm}) \text{ for } c \text{ in } \text{coefficients } p]) \\
& \quad ::\text{List } \text{OutputForm}) \\
0 & = 0$\text{Rep} \\
1 & = 1$\text{Rep} \\
x1 = x2 & = \\
& \quad \text{-- Note that we must strip out any terms greater than wtlevel} \\
& \quad \text{while degree } x1 > \text{wtlevel} \text{ repeat} \\
& \quad x1 := \text{reductum } x1 \\
& \quad \text{while degree } x2 > \text{wtlevel} \text{ repeat} \\
& \quad x2 := \text{reductum } x2 \\
& \quad x1 = $\text{Rep} x2 \\
x1 + x2 & = \\
& x1 +$\text{Rep} x2 \\
x1 * x2 & = \\
& \quad \text{-- Note that this is probably an extremely inefficient definition} \\
& \quad w := x1 *$\text{Rep} x2 \\
& \quad \text{while degree}(2) > \text{wtlevel} \text{ repeat} \\
& \quad w := \text{reductum } w \\
\end{align*}
\]

C.2 domain OWP OrdinaryWeightedPolynomials


-- domain OWP OrdinaryWeightedPolynomials --

)abbrev domain OWP OrdinaryWeightedPolynomials
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated 13 July 2016 by Tim Daly
++ Basic Functions: Ring, changeWeightLevel
++ Related Constructors: WeightedPolynomials
++ Also See: PolynomialRing
++ AMS classifications:
++ Keywords:
++ References:
++ Description:
++ 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
OrdinaryWeightedPolynomials(R,vl,wl,wtlevel) : SIG == CODE where
   R : Ring
   vl : List Symbol
   wl : List NonNegativeInteger
   wtlevel : NonNegativeInteger
SIG ==> Ring with
   if R has CommutativeRing then Algebra(R)
   coerce : % -> Polynomial(R)
      ++ coerce converts back into a Polynomial(R), ignoring weights
   coerce : Polynomial(R) -> %
      ++ coerce a Polynomial(R) into Weighted form,
      ++ applying weights and ignoring terms
   if R has Field then
      "/" : (%,%) -> Union(%,"failed")
      ++ a / b only works if minimum weight of divisor is zero,
      ++ and if R is a Field
   changeWeightLevel : NonNegativeInteger -> Void
      ++ This changes the weight level to the new value given:
      ++ NB: previously calculated terms are not affected
CODE ==> WeightedPolynomials(R,Symbol,IndexedExponents(Symbol),
   Polynomial(R),vl,wl,wtlevel)

C.3 domain WP2 WeightedPolynomials2

— domain WP2 WeightedPolynomials2 —
)abbrev domain WP2 WeightedPolynomials2
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated 13 July 2016 by Tim Daly
C.3. **DOMAIN WP2 WEIGHTEDPOLYNOMIALS2**

++ Basic Functions: Ring, changeWeightLevel
++ Related Constructors: PolynomialRing
++ Also See: OrdinaryWeightedPolynomials
++ AMS classifications:
++ Keywords:
++ References:
++ Description:
++ 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

WeightedPolynomials2(R,Varset,E,P,vl,wl,wtlevel) : SIG == CODE where
R : Ring
Varset : OrderedSet
E : OrderedAbelianMonoidSup
P : PolynomialCategory(R,E,VarSet)
v1 : List VarSet
wl : List NonNegativeInteger
wtlevel : NonNegativeInteger

SIG ==> Ring with

if R has CommutativeRing then Algebra(R)

coerce : % -> P
++ coerce converts back into a "P", ignoring weights

if R has Filed then

"/" : (%,%) -> Union(%,"failed")
++ a / b division only works if minimum weight of divisor is zero,
++ if R is a Field

coerce : P -> %
++ coerce a "P" into Weighted form, applying weights and ignoring terms

changeWeightLevel : NonNegativeInteger -> Void
++ This changes the weight level to the new value given:
++ NB: previously calculated terms are not affected

CODE ==> PolynomialRing(P,NonNegativeInteger) add

-- representations
Term := Record(k:NonNegativeInteger,c:P)
Rep := List Term
p : P
w,x1,x2 : %
n : NonNegativeInteger
z : Integer

changeWeightLevel(n) ==
  wtlevel := n
APPENDIX C. EXAMPLE CODE

lookupList : List Record(var: VarSet, weight:NonNegativeInteger)

if #vl ^= #wl then error "incompatible length lists in WeightedPolynomial"

lookupList := [[v,n] for v in vl for n in wl]

-- local operation

lookup : VarSet -> NonNegativeIntger
lookup v ==
  l := lookupList
  while l ^= [] repeat
    v = l.first.var => return l.first.weight
    l := l.rest
  0

innercoerce:(p,z) -> %
innercoerce(p,z) ==
z < 0 => 0
zero? p => 0
mv := mainVariable p
mv case "failed" => [[0,p]]
n := lookup(mv)
up := univariate(p,mv)
ans : %
an := 0
while not zero? up repeat
  d := degree up
  f := n*d
  lcup := leadingCoefficient up
  up := up - leadingMonomial up
  mon := monomial(1,mv,d)
  f < z => ans:=ans+[[tm.k+f,mon*tm.c] for tm in innercoerce(lcup,z-f)]
ans

coerce(p):% ==
  innercoerce(p,wtlevel)

coerce(w):P ==
  "*"/[tm.c for tm in w]

x1 = x2 ==
  -- Not that we must strip out any terms greater than wtlevel
  while not null x1 and x1.first.k > wtlevel repeat
    x1 := x1.rest
  while not null x2 and x2.first.k > wtlevel repeat
    x2 := x2.rest
  while not null x1 and not null x2 repeat
    x1.first.k ^= x2.first.k => return false
    x1.first.c ^= x2.first.c => return false
    x1 := x1.rest
    x2 := x2.rest
null x1 and null x2
\[x_1 \times x_2 ==
\begin{align*}
\text{null } x_1 & \Rightarrow 0 \\
\text{null } x_2 & \Rightarrow 0 \\
r : P \\
x_1.\text{first}.k = 0 & \Rightarrow \left[ 0 \text{ for } t_2 \text{ in } x_2 \mid (r := x_1.\text{first}.c \times t_2.c) \neq 0 \right] \\
\text{null } x_1 & \Rightarrow x_1 \\
+ \left[ 0 \text{ for } t_2 \text{ in } x_2 \mid (n := t_1.k + t_2.k) < \text{wtlevel and } (r := t_1.c \times t_2.c) \neq 0 \right] \\
\text{for } t_1 \text{ in reverse}(x_1) \right]
\end{align*}
\]

---

This 'reverse' is an efficiency improvement:
-- reduces both time and space [Abbott/Bradford/Davenport]

---

\[x : \% \times n : \text{NonNegativeInteger} ==
\begin{align*}
\text{zero? } n & \Rightarrow 1 \\
\text{expt}(x, n \text{ pretend PositiveInteger}) \\
\end{align*}
\]

\[\text{coerce}(p: \%) : \text{OutputForm} ==
\begin{align*}
\text{zero? } p & \Rightarrow (0 \text{Integer}) :: \text{OutputForm} \\
p.\text{first}.k = 0 & \Rightarrow p.\text{first}.c :: \text{OutputForm} \\
\text{reduce}("+", (\text{reverse } [\text{paren}(t_1.c :: \text{OutputForm}) \text{ for } t_1 \text{ in } p]) \\
:: \text{List OutputForm})
\end{align*}
\]

---

C.4 domain FCOMP FourierComponent

---

\[\text{FourierComponent}(E) : \text{SIG} == \text{CODE where} \]
\[E : \text{OrderedSet} \]
\[\text{SIG} == \text{OrderedSet with} \]
\[\begin{align*}
\text{sin} & : E \rightarrow \% \\
& \text{++ sin makes a sin kernel for use in Fourier series} \\
\text{cos} & : E \rightarrow \% \\
& \text{++ cos makes a cos kernel for use in Fourier series}
\end{align*}\]
APPENDIX C. EXAMPLE CODE

\hspace{1cm} \textbf{sin? : \% -> Boolean}
++ \textit{sin?} true if term is a \textit{sin}, otherwise false

\hspace{1cm} \textbf{argument : \% -> E}
++ \textit{argument} returns the argument of a given \textit{sin/cos} expression

\textbf{CODE ==> add}

-- \textit{representations}
\textbf{Rep := Record(SinIfTrue:Boolean, arg:E)}
\textbf{e : E}
\textbf{x,y : \%}

\textbf{sin e ==}
\textbf{[true,e]}

\textbf{cos e ==}
\textbf{[false,e]}

\textbf{sin? x ==}
\textbf{x.arg}

\textbf{argument x ==}
\textbf{x.arg}

\textbf{coerce(x):OutputForm ==}
\textbf{hconcat((if x.SinIfTrue then "sin" else "cos")::OutputForm,}
\textbf{bracket((x.arg)::OutputForm))}

\textbf{x < y ==}
\textbf{x.arg < y.arg => true}
\textbf{y.arg < x.arg => false}
\textbf{x.SinIfTrue => false}
\textbf{y.SinIfTrue}

C.5 \hspace{1cm} \textbf{domain FSERIES FourierSeries}

--- \textbf{domain FSERIES FourierSeries} ---

)abbrev domain FSERIES FourierSeries
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 13 July 2016 by Tim Daly
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
FourierSeries(R,E) : SIG == CODE where
  R : Join(CommutativeRing,Algebra(Fraction Integer))
  E : Join(OrderedSet,AbelianGroup)

SIG ==> Algebra(R) with
  if E has canonical and R has canonical the canonical
  coerce : R -> %
    ++ coerce converts coefficients into Fourier Series
  coerce : FourierComponent(E) -> %
    ++ coerce converts sin/cos terms into Fourier Series
  makeSin : (E,R) -> %
    ++ makeSin makes a sin expression with given argument and coefficient
  makeCos : (E,R) -> %
    ++ makeCos makes a cos expression with given argument and coefficient

CODE ==> FreeModule(R,FourierComponent(E)) add

  -- representations
  Term := Record(k:FourierComponent(E),c:R)
  Rep := List Term
  w,x1,x2 : %
  t1,t2 : Term
  n : NonNegativeInteger
  z : Integer
  e : FourierComponent(E)
  a : E
  r : R
  1 ==
    [[cos 0,1]]

  coerce e ==
    sin? e and zero? argument e => 0
    if argument e < 0 then
      not sin? e => e:=cos(- argument e)
      return [[sin(- argument e),-1]]
      [[e,1]]

  multiply : (Term,Term) -> %
  multiply(t1,t2) ==
    r := (t1.c*t2.c)*(1/2)
    s1 := argument t1.k
    s2 := argument t2.k
    sum := s1+s2
    diff := s1-s2
    sin? t1.k =>
      sin? t2.k =>
        makeCos(diff,r) + makeCos(sum,-r)
\[
\text{makeSin}(\text{sum}, r) + \text{makeSin}(\text{diff}, r)
\]
\[
\sin? \ t2.k => \\
\text{makeSin}(\text{sum}, r) + \text{makeSin}(\text{diff}, r)
\]
\[
\text{makeCos}(\text{diff}, r) + \text{makeCos}(\text{sum}, r)
\]

\[
x1 \times x2 == \\
\text{null \ x1 => 0} \\
\text{null \ x2 => 0} \\
+/+/[\text{multiply}(t1, t2) \ \text{for} \ t2 \ \text{in} \ x2] \ \text{for} \ t1 \ \text{in} \ x1]
\]

\[
\text{makeCos}(a, r) == \\
\text{a < 0 => [[cos(-a), r]]} \\
[[\text{cos} \ a, r]]
\]

\[
\text{makeSing}(a, r) == \\
\text{zero? \ a => []} \\
\text{a < 0 => [[sin(-a), -r]]} \\
[[\text{sin} \ a, r]]
\]
Appendix D

The Makefile

__ * __

PROJECT=bookvol2
TANGLE=/usr/local/bin/NOTANGLE
WEAVE=/usr/local/bin/NOWEAVE
LATEX=/usr/bin/latex
MAKEINDEX=/usr/bin/makeindex

all:
${WEAVE} -t8 -delay ${PROJECT}.pamphlet >${PROJECT}.tex
${LATEX} ${PROJECT}.tex 2>/dev/null 1>/dev/null
${MAKEINDEX} ${PROJECT}.idx
${LATEX} ${PROJECT}.tex 2>/dev/null 1>/dev/null
Bibliography


Abstract: This paper describes the applications area of computer programs that carry out formal algebraic manipulation. The first part of the paper is tutorial and several typical problems are introduced which can be solved using algebraic manipulative systems. Sample programs for the solution of these problems using several algebra systems are then presented. Next, two more difficult examples are used to introduce the reader to the true capabilities of an algebra program and these are proposed as a means of comparison between rival algebra systems. A brief review of the technical problems of algebraic manipulation is given in the final section.


Abstract: We describe the methods available in the current REDUCE system for introducing new mathematical domains, and illustrate these by discussing several new domains that significantly increase the power of the overall system.


Abstract: We extend a recent algorithm of Trager to a decision procedure for the indefinite integration of elementary functions. We can express the integral as an elementary function or prove that it is not elementary. We show that if the problem of integration in finite terms is solvable on a given elementary function field $k$, then it is solvable in any algebraic extension of $k(\theta)$, where $\theta$ is a logarithm or exponential of an element of $k$. Our proof considers an element of such an extension field to
be an algebraic function of one variable over $k$. In his algorithm for the integration of algebraic functions, Trager describes a Hermite-type reduction to reduce the problem to an integrand with only simple finite poles on the associated Riemann surface. We generalize that technique to curves over liouvillian ground fields, and use it to simplify our integrands. Once the multiple finite poles have been removed, we use the Puiseux expansions of the integrand at infinity and a generalization of the residues to compute the integral. We also generalize a result of Rothstein that gives us a necessary condition for elementary integrability, and provide examples of its use.


Abstract: This work is concerned with the following question: "When is an algebraic function integrable?". We can state this question in another form which makes clearer our interpretation of integration: "If we are given an algebraic function, when can we find an expression in terms of algebraics, logarithms and exponentials whose derivative is the given function, and what is that expression?". This question can be looked at purely mathematically, as a question in decidability theory, but our interest in this question is more practical and springs from the requirements of computer algebra. Thus our goal is "Write a program which, when given an algebraic function, will produce an expression for its integral in terms of algebraics, exponentials and logarithms, or will prove that there is no such expression".


Abstract: This paper discusses the design and implementation of MODLISP, a LISP-like language enhanced with the idea of MODes. This extension permits, but does not require, the user to declare the types of various variables, and to compile functions with the arguments declared to be of a particular type. It is possible to declare several functions of the same name, with arguments of different type (e.g. PLUS could be declared for Integer arguments, or Rational, or Real, or even Polynomial arguments) and the system will apply the correct function for the types of the arguments.


Abstract: While computer algebra systems have dealt with polynomials and rational functions with integer coefficients for many years, dealing with more general constructs from commutative algebra is a more recent problem. In this paper we explain how one system solves this problem, what types and operators it is necessary to introduce and, in short, how one can construct a computational theory of commutative algebra. Of necessity, such a theory is rather different from the conventional, non-constructive, theory. It is also somewhat different from the theories of Seidenberg [1974] and his school, who are not particularly concerned with practical questions of efficiency.


Abstract: This paper explains how Scratchpad solves the problem of presenting a categorical view of factorization in unique factorization domains, i.e. a view which can be propagated by functors such as SparseUnivariatePolynomial or Fraction. This is not easy, as the constructive version of the classical concept of UniqueFactorizationDomain cannot be so propagated. The solution adopted is based largely on Seidenberg's conditions (F) and (P), but there are several additional points that have to be borne in mind to produce reasonably efficient algorithms in the required generality. The consequence of the algorithms and interfaces presented in this paper is that Scratchpad can factorize in any extension of the integers or finite fields by any combination of polynomial, fraction and algebraic extensions: a capability far more general than any other computer algebra system possesses. The solution is not perfect: for example we cannot use these general constructions to factorize polynomials in $\mathbb{Z}[\sqrt{-5}][x]$ since the domain $\mathbb{Z}[\sqrt{-5}]$ is not a unique factorization domain, even though $\mathbb{Z}[\sqrt{-5}]$ is, since it is a field. Of course, we can factor polynomials in $\mathbb{Z}[\sqrt{-5}][x]$.

Abstract: AXIOM is a computer algebra system superficially like many others, but fundamentally different in its internal construction, and therefore in the possibilities it offers to its users. In these lecture notes, we will

- outline the high-level design of the AXIOM kernel and the AXIOM type system,
- explain some of the algebraic facilities implemented in AXIOM, which may be more general than the reader is used to,
- show how the type system and the information system interact,
- give some references to the literature on particular aspects of AXIOM and,
- suggest the way forward.


Abstract: Axiom is a computer algebra system superficially like many others, but fundamentally different in its internal construction, and therefore in the possibilities it offers to its users and programmers. In these lecture notes, we will explain, by example, the methodology that the author uses for programming substantial bits of mathematics in Axiom.


Abstract: This paper reports ongoing research at the IBM Research Center on the development of a language with extensible parameterized types and generic operators for computational algebra. The language provides an abstract data type mechanism for defining algorithms which work in as general a setting as possible. The language is based on the notions of domains and categories. Domains represent
algebraic structures. Categories designate collections of domains having common operations with stated mathematical properties. Domains and categories are computed objects which may be dynamically assigned to variables, passed as arguments, and returned by functions. Although the language has been carefully tailored for the application of algebraic computation, it actually provides a very general abstract data type mechanism. Our notion of a category to group domains with common properties appears novel among programming languages (cf. image functor of RUSSELL) and leads to a very powerful notion of abstract algorithms missing from other work on data types known to the authors.

Comment: IBM Research Report 8930


Abstract: The main problem which occurs in developing Computer Algebra packages for special areas in mathematics is the complexity. The unique concept which is advocated to cope with that problem is the introduction of suitable abstract data types. The corresponding decomposition into modules makes it much easier to develop, maintain and change the program. After introducing the relevant concepts from software engineering they are elaborated by means of the symmetry analysis of differential equations and the Scratchpad package SPDE which abbreviates Symmetries of Partial Differential Equations.
Index

canonical, 37

tagged comment, 19