axiom™

The 30 Year Horizon

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

Volume 4: Axiom Developers Guide

July 9, 2020

6d805049a0a4f2fb41b0cee798ebf040181ab1d8

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

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

Paul Kosinski
Klaus Kusche
Larry Lambe
George L. Legendre
Michel Levaud
Ren-Cang Li
Craig Lucas
Camm Maguire
William Martin
Alasdair McAndrew
Bob McNeill
David Mentre
Gerard Milmeister
H. Michael Moeller
Scott Morrison
William Naylor
John Nelder
Jinzhong Niu
Kostas Oikonomou
Bill Page
Susan Pelzel
Ayal Pinkus
Jose Alfredo Portes
Beresford Parlett
Peter Poromaas
Arthur C. Ralfs
Guilherme Reis
Michael Richardson
Robert Risch
Simon Robinson
Martin Rubey
Philip Santas
William Schelter
Marshall Schor
Steven Segletes
William Sit
Matthieu Sozeau
Jonathan Steinbach
Klaus Sutner
Eugene Surowitz
T. Doug Telford
Balbir Thomas
Dylan Thurston
Dick Toupin
Hale Trotter
Kresimir Veselic
Bernhard Wall
Andreas Weber
Juergen Weiss
James Wen
R. Clint Whaley
Berhard Will
Shmuel Winograd
Waldemar Wiwianka
Liu Xiaojun
Qian Yun
Evelyn Zoernack

Igor Kozachenko
Bernhard Kutzler
Kaj Laurson
Franz Lehner
Howard Levy
John Lipson
Michael Lucks
Dave Mainey
Ursula Martin
Bob McElrath
Edi Meier
Jonathan Millen
William Miranker
Michael Monagan
Joel Moses
Patrice Naudin
Godfrey Nolan
Michael O'Connor
Humberto Ortiz-Zuazaga
David Parnas
Michel Petitot
Frederick H. Pitts
E. Quintana-Orti
A. Petitet
Greg Puhak
Norman Ramsey
Huan Ren
Jason Riedy
Jean Rivlin
Raymond Rogers
Jeff Rutter
David Saunders
Gerhard Schneider
Frithjof Schulze
V. Sima
Elena Smirnova
Srinivasan Seshan
Fabio Stumbo
Robert Sutor
Yong Kiam Tan
James Thatcher
Mike Thomas
Francoise Tisseur
Raymond Toy
Themos T. Tsikas
Christof Voemel
Paul Wang
Jaap Weel
M. Weller
Thorsten Werther
James T. Wheeler
Clifton J. Williamson
Robert Wisbauer
Knut Wolf
Clifford Yapp
Vadim Zhytnikov
Bruno Zuercher

Fred Krogh
Tim Lahey
Charles Lawson
Frederic Lehobey
J. Lewis
Rudiger Loos
Richard Luczak
Francois Maltey
Osni Marques
Michael McGettrick
Ian Meikle
Victor S. Miller
Mohammed Mobarak
Marc Moreno-Maza
Mark Murray
C. Andrew Neff
Arthur Norman
Summat Oemrawsingh
Julian A. Padget
Norm Pass
Didier Pinchon
Frank Pfenning
Gregorio Quintana-Orti
Andre Platzer
Claude Quitte
Anatoly Raportirenko
Albert D. Rich
Renaud Rioboo
Nicolas Robidoux
Michael Rothstein
R.W Ryniker II
Alfred Scheerhorn
Martin Schoenert
Fritz Schwartz
Nick Simicich
Jacob Nyffeler Smith
Ken Stanley
Christine Sundaresan
Moss E. Sweedler
Max Tegmark
Laurent Thery
Carol Thompson
Steve Toleque
Barry Trager
Gregory Vanuxem
E.G. Wagner
Stephen Watt
Al Weis
Mark Wegman
Michael Wester
John M. Wiley
Stephen Wilson
Sandra Wityak
Yanyang Xiao
David Yun
Richard Zippel
Dan Zwillinger

Contents

New Foreword

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

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

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

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

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

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

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

Confronting every new programmer learning a new language are

- The Cave of Artifacts
- The Forest of Tooling
- The Mountain of Language
- The Cloud Castle of Mindset

– Daniel Higginbotham in Clojure for the Brave and True

0.1 MODLISP by James Davenport

This is based on a paper by Davenport and Jenks [Dave80a].

It provides historical details of the original SCRATCHPAD II design. Some of these details have changed over time but it is important to know what motivates some of the design choices. The verbatim text of the paper follows.

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 uesr 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.

The MODLISP language differs from other abstract data type languages such as CLU[Lisk77, Lisk77a, Lisk79] and Russell [Dona77] in that it allows dynamic construction of new parameterised data types and possesses a unified semantics covering interpreted and compiled code, which can call one another at will. In short, it is LISP-like.

0.1.1 Introduction

MODLISP is a system developed from LISP with the primary aim of supporting research in computer algebra. It has been realised for some years ([Loos74, Jenk77]) that progress in computer algera required the development of systems which were able to deal with formal "modes" such as "polynomial" or "matrix".

Just as LISP has been the vehicle of choice for implementing computer algebra systems in the past, it was felt that implementing such a system was best done in a typed variant of LISP, and since none such was available at the time the project started, MODLISP was born. However, MODLISP is more than just an implementation vehicle. It is an interesting language in its own right, with powerful features for the manipulation of abstract and parameterised data types.

0.1.2 Interpretive Structure

The evaluator is the key to any LISP system, and so it is to MODLISP. The quintessential difference between MODLISP and LISP 1.5 can be discerned in the evaluator: whereas a conventional LISP evaluator has two arguments (the expression to be evaluated and the environment in which it is to be evaluated), the MODLISP evaluator has three (the expression

to be evaluated, the desired mode of the answer, and the environment in which it is to be evaluated). We present in Appendix A a somewhat simplified description of the MODLISP evaluator. Here we make some brief remarks on the evaluation strategy.

The major difference between MODLISP and LISP 1.5 lies in the area of function application. A conventional LISP system's strategy, when faced with an operator-operand form to evaluate, is to evaluate the CAR (i.e. the operator part) and then (assuming it is not some sort of macro) to evaluate the arguments recursively in a left-right manner, and then to APPLY the operator part to the list of arguments in some manner (evalutating in an environment enriched by the bindings of the arguments, in the case that the CAR was a λ expression.

Regretably, such a simple approach is impossible in the case of MODLISP. The easiest way of exemplifying this is to see that we cannot evaluate the operator until we know what the modes[1] of the operands are: e.g. the TIMES function for integers is different from that for polynomials or for matrices. Furthermore, there is the additional complication of heterogeneous argument lists; e.g. multiplying a polynomial by an integer.

Fortunately things are not as bleak as all that – we can still evaluate as far as the name of the operator (assuming that evaluating the operator will yield a name) without knowing the types of (or indeed anything else about) the operands. The difficulty comes in the step that corresponds to a conventional LISP 1.5 system accessing the EXPR (or whatever) property of the operator name. The MODLISP system needs to apply one of a variety of functions at this point, depending on the modes of the operands, rather than one specific function.

Rather than associate an EXPR property with an atom, therefore, MODLISP associates a MODEMAP property with it, whose corresponding value is a *modemap list*[2] These modemap lists are to MODLISP what *subrs* (i.e. BPIs) are to conventional LISPs: they are the applicable objects, as defined by the evaluator. The evaluator looks through the modemap list, which is a list of maps and associated functions, until it finds a map which matches the arguments. This causes somewhat of a "chicken and egg" situation, since the map cannot be matched until we know what the modes of the arguments are, and we cannot evaluate the arguments until we know what mode to evaluate them in, which the map will tell us.

Hence the evaluator (in fact function EVFORM, as described in Appendix A) runs down the list of arguments, evaluating them in the mode given by the corresponding entry of the map. An example of a case where this complexity is needed is given by the (admittedly artificial, but realistic examples are more complex) construct:

```
(TIMES (ZERO) (RATIONAL 1 2))
```

where RATIONAL is a function which takes two Integer arguments and returns a Rational result. ZERO is a function which returns the zero element of whatever type is required (and, of course, the Integer zero may well be different from the Rational zero or the Matrix zero). TIMES might have a (simplified) modemap-list such as:

```
( ((Integer Integer Integer) IntegerTimes)
  ((Rational Rational Rational) RationalTimes) )
```

[1] We will later define precisely what we mean by the term 'mode'. For the time being, it may help the reader to regard it as the equivalent of 'type' in conventional typed languages.

[2] We imply here that a modemap list is a list of (map, function) pairs. The actual structure is not significantly more complicated (as mentioned in the Appendix A) and is a list of (map, cond-part) pairs, where a cond-part is a list of (predicate, function) pairs. The function is only used if the predicate evaluates to T. While this additional level is not required in the system as described so far, several of the features of MODLISP described later in the paper require it. It is mentioned here to avoid giving a false impression.

Then (assuming for the sake of simplicity that there is no particular mode for the result) the evaluator would first try to apply the first modemap (which states that TIMES can yield an Integer result from two Integer arguments) and would evaluate (ZERO) in mode Integer, which would work, and then would evaluate (RATIONAL 1 2) in mode Integer, which would not (i.e. eval returns NIL). Hence we infer that that map is not applicable, and we attempt to apply the second modemap. This causes (ZERO) and (RATIONAL 1 2) to be evaluated in mode Rational, which works, and hence the function (in the untyped sense – i.e. that definition which was asserted to be valid for the case of multiplying two Rationals) RationalTimes is applied.

The modemap for TIMES quoted above looks as if it is the solution to the problem of *polymorphic operators*, i.e. those which correspond to different functions for different data types, but in a computer algebra setting it is not a complete answer. One way of seeing this is to ask "How many definitions of TIMES are there?", and the answer is that there are an unlimited number of them. There are definitions of TIMES for polynomials over the integers, matrices over the rational numbers, polynomials over matrices over the integers, polynomials over the Gaussian integers, polynomials over the field of fractions of the polynomials over the integers modulo 7 etc. Searching such a modemap would be extremely inefficient, but that is by no means the worst of the problem – how are we to know that we have placed enough definitions on the modemap. It should now be obvious that a modemap which lists simple domains and the associated functions is insufficient for our purposes – and that is why we need the concept of a *mode*.

0.1.3 What is a MODE?

In order to discuss MODLISP, we need to define what is meant by the term 'mode'. The terminology in this area is sufficiently confused, with different authors using the same word to mean many different things, that we hope the reader will bear with us while we define many terms in the way we intend to use them.

The first term we wish to define is 'domain', (short for 'domain of computation') whose meaning approximates that of 'data type' in traditional programming languages. The traditional view of such 'data types' is best summarised by Jensen & Wirth [Jens75, p. 12] – "A data type defines the set of values a variable may assume". More recent work has lead to the view-point [Morris, 1973] that a type is a set of values complete with a list of operations that can be performed on those values.

The simplest example of such a domain is that of a basic type, e.g. Integer or Boolean. The basic types built into MODLISP are Integer, Boolean, Identifier and String, though the user can (and probably will) define others.

The next level of complexity is provided by the structured types, of which MODLISP provides two: Union and Struct. Familiar examples of these are Sexpr, defined as Union(Pair,Atom) and Pair, defined as Struct(Car:Sexpr, Cdr:Sexpr).

We can move up from this to *parameterised data types* or just *types*, also known as 'type generators', as defined by ADJ [Ehri80a]. The basic MODLISP system provides two parameterised data types for efficiency (List and Vector), but there are facilities for defining one's own, and it is expected that a large amount of MODLISP programming will be done via these user-defined types. General examples of parameterised data types include Stack and Alist, while computer algebra is full of such types: Polynomial, Matrix, Algebraic Extension, Rational Function etc.

ADJ only considered parameterised data types parameterised by one or more domains (in their terminology 'data types'), whereas we may also wish to parameterise them by actual values, or a mixture of the two. An exmple of this is "3-dimensional vectors over the rational numbers", which is parameterised both by the number 3 and by the type "rational number".

The previous definitions allow us to define a *domain* (recursively) as being either a basic type, or the result of supplying actual values and domains for all the parameters of a (parameterised data) type. Hence "Vectors of Integers, of length 3", is a domain, since "Integers" is a basic type, 3 is an actual value for the length of the vectors, and "Vector" is a type parameterised by a positive integer and a type. Conversely, "Vectors of Integers" is not a domain, for we have not specified the length.

We can now answer the question posed by the heading of this section: "What is a mode?". The answer is deceptively simple: a mode is *any* class of domains. While the MODLISP system places no limitation on the definition of this class, some classes are more useful than others. We shall see below (under the heading 'Compilation for Arbitrary Domains') that it is possible to define a mode consisting of all the domains on which certain operations (satisfying certain axioms) are defined – this kind of mode will be called a 'category'.

Some particularly useful modes are those constructed by substituting actual values for some parameters of a type, hence we can speak about the mode of "Vectors of length 3" or the mode of "Stacks". These modes appear naturally in the modemap for many operations, as we see in the next paragraph. One mode that crops up particularly in interactive computer algebra, but is useful elsewhere, is NilMode, denoting the class of all domains.

With parameterised data types, one wants only one routine, and one entry inthe modemap for a particular operator, for all the various parameterisations of one type, e.g. one entry for Vector, and not one for Vector(3,Rational), another for Vector(2,Complex) etc. This is made possible by the use of pattern-matching variables (denoted by *1, *2, ...) in modemaps. For example, the map for 'scalar product of vectors' might look like[3]

```
(*2 (Vector *1 *2) (Vector *1 *2))
```

which could be applied with *1=3 and *2=Rational, or with *1=2 and *2=Complex. If pattern matching alone is unable to express the precise nature of an operation, then the conditional-part of the modemap (mentioned in the prior footnote) can be used to add extra restrictions to the applicability of a particular function definitions. For example we could have a map of (*2 *3 *3) and a conditional expression of the form

```
(EQUAL *3 '(VECTOR *1 *2))
```

which would have the same overall effects as the previous map.

0.1.4 Compilation

While such a system may appear laudable, it will not make progress in computer algebra unless efficient compilation is available. Many of the problems of computer algebra strain the resources of even the largest computers, and any extra overhead due to interpretation cannot be afforded (especially as MODLISP interpretation is inherently more expensive than LISP interpretation due to the necessity to analyse modemaps). This gives rise to the obvious

[3] The internal representation is actually somewhat different, since we distinguish between parameters which are types, and those which are expressions (e.g. between "Rational" and "3"), and there is also space to record whether or not that part of the map is obligatory or optional (see below under "Partial Declarations").

question: "How do we compile modemap selection?".

It is reasonably clear that, in the case where all the arguments of a λ expression are of known definite type (e.g. Integer), then all the analysis can indeed be done, and we are in a situation similar to standard LISP compilation. The only difficulty might be that both the compiler and the evaluator (i.e. interpreter) would perform modemap analysis, and it would be possible for these two to get out of step, thus giving compiled code a different semantics from interpreted code. This difficulty is met by adopting the technique of *partial compilation* [Ersh77], in which the evaluator also acts as the compiler, producing values where it can, and compiled code where it cannot[4] As an example of this, let us consider the compilation of (PLUS X Y) in three cases:

1. X is 2 of mode Integer, Y is 3 of mode Integer. Then the modemap for PLUS tells us to apply the function IntegerPlus, and the evaluator does so, returning the answer 5.
2. X and Y are known to be of mode Integer, but their values are not known. For example, they might be the formal parameters of a function. In this case, we can still select a function from the modemap for PLUS, since we do know the modes of X and Y. We cannot apply the function IntegerPlus that is produced, but we can compile code to produce it, so the compiled code[5] is (IntegerPlus X Y).
3. Nothing is known about X and Y. In this case, we have no idea how to apply the modemap for PLUS, and we must compile a call to eval.

These additions to the evaluator, to allow it to function as a compiler as well, while requiring significant changes to the treatment of some of the special forms, do not require a major rewrite of the actual evaluator. Having compilation performed by the partial compilation feature of the MODLISP evaluator has an additional advantage – any expressions which can be evaluated at compile time are, and so, for example, in (PLUS X (TIMES 1 2)) the multiplication would be replaced by 2. While one could argue that such code should not be written, one case where it is hard not to do so is conversions. If X is Rational, then in (TIMES X 2), the number 2 must be converted from Integer to Rational for the multiplication to be performed. This conversion would automatically be done at compile time rather than run time.

0.1.5 Compilation for Arbitrary Domains

A significant problem is how to compile code (not just calls to the evaluator) for parameterised data types, so that, for example, the code for Polynomial can be compiled, and can then be used for Polynomials of Integers, Polynomials of Matrices, Polynomials of Polynomials etc. or so that we can write (and compile) *polymorphic functions*, such as a sort function that will sort any set on which a comparison predicate has been defined. Note that this is a problem not addressed by, say, the Mode Reduce system [Hear74], where a separate version of the module is compiled for each underlying domain. Some of the mechanism required for this exists already – as we saw above we can declare a modemap for the scalar-product of two vectors which is independent of the dimension of the vectors, or of the ground field. Hence calls to this routine can be compiled even when these parameters of Vector are not known.

[4] Note that we are not considering delayed (or lazy) evaluation [Hend76] here. The 'rule' feature of MODLISP [Jenk79] is based on such a concept, but, as it is independent of the ideas considered in this paper, we shall not discuss it further.

[5] The current MODLISP implementation compiles into LISP370[IBMx78], which is compiled into machine code by the LISP370 compiler.

The answer adopted is to declare some properties of the parameterising types, so that, just as the first argument to Vector, as discussed above, is declared to be an integer, so the second is declared to be a Field, i.e. a type on which certain operators are valid.

We require a theoretical basis for "declaring some properties of the parameterising types", and this is provided in the work of ADJ. [Gogu76, Ehri80a] We do not intend to discuss the whole of their approach here, and the reader is referred to the papers quoted for further details, but it is sufficient for present purposes to say that they consider a *data type* (which corresponds to our *domain*) to be a (many-sorted) *algebra* and a list of operations between the carriers. This agrees with Morris[Morr73] in interpreting a type as a set (or sets) with operations. These operations must not only exist but are also required to satisfy certain *axioms*.

We might wish to define a stack of integers this way (this example is based on a rigorous discussion by ADJ [Ehri80a, p. 11]) by saying that there are two carriers I and S (informally the integers themselves, and stacks of them)[6], and operations:

$$\begin{array}{ll} \texttt{Create}: & () \to S(\texttt{noarguments}), \\ \texttt{Push}: & (I, S) \to S, \\ \texttt{Pop}: & (S) \to S^{7} \\ \texttt{Top}: & (S) \to I, \\ \texttt{Empty}: & (S) \to \mathit{Boolean} \end{array}$$

Of course, what we want to do is to define stack as an abstract idea, implemented by one program, rather than just to define stack-of-integers, and later define stack-of-rationals, etc. by separate programs.

We now define a *category* of algebras (i.e. types) to be the class of all algebras with carriers indexed by the same sets, and with equivalent operators which satisfy a certain list of axioms (this definition is deliberately vague and intuitive – precise definitions are given by ADJ[Ehri80a]). As an example we can define the category of all stacks to be the class of all algebras with two carriers (indexed by I and S)[8] with operators

$$\begin{array}{ll} \texttt{Create}: & () \to C_S(\texttt{noarguments}), \\ \texttt{Push}: & (C_I, C_S) \to C_S, \\ \texttt{Pop}: & (C_S) \to C_S \\ \texttt{Top}: & (C_S) \to C_I, \\ \texttt{Empty}: & (C_S) \to \mathit{Boolean} \end{array}$$

Here C_S is the carrier indexed by S and C_i is the carrier indexed by I. These operations must satify certain axioms, such as

```
Top(Push(x,s)) = x;
Pop(Push(x,s)) = s;
Empty(Create()) = True;
Empty(Push(x,s)) = False;
```

We can also abstractly define other structures, corresponding not to one concrete implementation (such as Stack), but to whole families. One example is Collection, which consists of

[6] Actually the mathematics demands a third carrier, that of the Booleans, since the function Empty takes values in the Booleans (i.e. True or False). Computationally, however, the Booleans are a constant, and we do not need to declare them specially.

[8] As remarked in a previous footnote, there is a technical requirement for a third carrier Boolean. Since this is intended to be a constant, we shall ignore this requirement. We can view what we write as a shorthand for a formal language in which these constant carriers were always fully defined, and with suitable extra axioms to ensure that all instances of the constant Boolean were isomorphic.

two carriers I and S (or, more precisely, each element of the category Collection consists of two carriers indexed by I and S) with the following operations:

```
New:    () -> S;
Add:    (I,S) -> S;
Empty:  (S) -> Boolean;
Member: (I,S) -> Boolean
```

and the following axioms:

```
Empty(New())       = True;
Empty(Add(x,s))    = False;
Member(x,New())    = False;
Member(x,Add(y,s)) = if x=y then True else Member(x,s)
```

This represents any collection to which we can add things, and can test whether or not an item is in the collection. The concrete representation of this abstract category includes lists, lists without duplicates, hash tables, balanced trees etc.

The general problem of compilation is then typified by the question: "How do we compile something to run over any of these domains, without any modemap resolution at run time?".

We discussed above, when describing modemaps for parameterised routines, the possibility of using pattern matching variables in modemaps. Using these, we can describe the modemap for a routine (such as BelongsList, which, given a list of members of I, and a collection, returns a list of those members of the list that lie in the collection) by a map like ((List *1) (List *1) *2) with a conditional part expressing the fact that *2 has to be a collection of *1.

The remaining problem is to place something in the modemap which the compiler can access and place into the compiled code, even though it does not know what, in this case, either I or S actually is – I could be the Integers and S a hash table, or I could be a set of polynomials and S a linked list. However we do know, from our formalism that the only functions that can be accessed are New, Add, Empty and Member, so that if we know of a vector which will contain the four functions, in a fixed order, at run time, then the compiler can just compile accesses into this vector. For example, New might be element 5 of the vector[9] and then a call to New would compile into code to select the fifth element and apply it.

This is in fact the scheme adopted – each description of a category (such as Collection above, or the many categories Ring, Group, Field etc. are needed in computer algebra) defines the layout of a vector which is to represent the operations defined on any domain lying in that category. The modemaps for these operations are updated to include appropriate entries, for example PLUS has an entry of

```
((*1 *1 *1) ( (OF *1 Monoid) (Elt *1 5)))
```

meaning that PLUS is a function valid in all Monoids (the clause starting OF is the conditional part of the modemap) and that its definition is to be found as element 5 of the vector representing the operations on the monoid.

Hence the only extra overhead at run-time due to the existence of parameterised types (i.e. not knowing at compile time what the types of the operands were) is the cost of the element extraction – typically two machine instructions.

[9] In the system, "user" functions start at the fifth element – elements 0, 1 and 2 are reserved for bookkeeping, 3 contains the equality-testing function, and 4 contains the printing function.

0.1.6 Partial Declaration

We have seen in the previous section that, if we can declare, for example, that X and Y belong to some unspecified domain G which is a Group (for example) then we will be able to compile code which will operate over all groups. This is not difficult to do (though the algebra system being implemented in MODLISP has a high-level language preprocessor to make it easier), and gives rise to a very powerful system.

In fact we can do more – this system opens up the possibility of "partial declaration", which can be performed in either interpreted or compiled code. We can say, for example, that F is to be a polynomial in X, but we do not care over what the polynomial is taken (one example of this would be in integration, where we wish to express the function to be integrated in terms of the variable of integration, and all other variables etc. are secondary). Code to manipulate F can still be compiled, because we know that polynomials always lie in rings, and the precise nature of the underlying coefficients is irrelevant since all the necessary information (functions to manipulate them, etc.) will be contained in the vector implementing the particular polynomial domain.

0.1.7 Status of the Project

The MODLISP system is currently implemented above LISP/370[IBMx78], and is being used to develop an experimental computer algebra system which is based on abstract data types, in the sense that it is possible to define "Polynomial" as an operation which takes a ring, and yields another one (i.e. a vector of operations which can be accessed by a MODLISP program compiled knowing that its arguments came from a ring).

0.1.8 Acknowledgments

We are grateful to J.W. Thatcher and E.G. Wagner for many useful discussions about abstract parameterised data types, and to D.R. Barton, J.D. Cohen and D.Y.Y. Yun for many fruitful discussions on MODLISP. Earlier drafts of this paper were read by F.W. Blair, R.W. Ryniker II and J.W. Thatcher, and we are grateful to them for many helpful comments and criticisms.

0.1.9 Appendix A: Description of MODLISP Evaluator

This appendix first gives a formal description of a simplified MODLISP evaluator, one which does not provide for partial compilation or partial declarations. The overall interpretive structure is more easily seen from this simpler description. We then discuss how this evaluator can be extended to the complete MODLISP evaluator, which provides both partial compilation and partial declaration, and for the interaction of the two.

The following summarizes the dialect of LISP used below to describe the evaluator. (QUOTE X) is abbreviated by "X. (LET X .) is used to introduce a local variable X. The notation <X M E>is used for an atomic datatype called *triple* consisting of an expression X, a mode M, and an environment E. An argument of the form (: A B) to a λ expression means "A has the form of B", and has the side-effect of binding symbols appearing in B to the indicated parts of the structure. The (REPEAT (..) X) repeats X as given by the iterators (..) and has the value NIL. (COLLECT (..) X) does the same but its value is a list of the successive values

of X (thus, (COLLECT ((IN U X)) (F U)) is similar to (MAPCAR X (FUNCTION F))). COLLECT and REPEAT do not generate PROGs, so that (RETURN X) always causes an exit from the function containing it with value X.

```
(EVAL (LAMBDA ((: T <X M E>)) (COND
  ((ATOM X) (PROGN
    (LET TP (COND
      ((TRIPLEP X) ((GETEVALFUN M E) X))
      ((SYMBOLP X) (EVSYMBOL X E))
      ("T <X (MODE X) E>) ))
    (EVCONVERT TP M) ))
  ((ATOM (LET OP (CAR X))) (PROGN
    ((LET FN (GET OP "SPECIAL E)) (FN T))
    ((REPEAT ((IN MAP (GET OP "MODEMAP E)))
      (COND ((LET Z (EVFORM T MAP))
               (RETURN Z)))))
    ("T NIL) ))
  ((EQ (CAR OP) "LAMBDA)
    (EVLAPPLY (CADR OP) (CADDR OP) (CDR X) M E))
  ("T (EVAL <(S:EXPR (EVAL <OP "SEXPR E>))
             M
             E>)) )))

(EVSYMBOL (LAMBDA (S E) (COND
  ((LET V (GET S "VALUE E)) V)
  ((GET S "MODE E) NIL)
  ("T <S "SYMBOL E>) )))

(EVCONVERT (LAMBDA (T M) (COERCE T M)))

(EVFORM (LAMBDA ((: T <(OP . ARGL) TM E>)
                 (: MAP (SIG . CEXPR))) (PROGN
  (LET MAP (SUBLIS (OR (MATCH TM (CAR SIG))
                       (RETURN NIL)) MAP))
  (LET L (COLLECT ((IN A ARGL) (IN M (CADR MAP)))
    (S:EXPR (LET T (OR (EVAL <A M
                                (LET E (S:ENV T))>)
                         (RETURN NIL)))) ))
  (REPEAT ((IN X L))
    (COND ((EQUAL "True
             (S:EXPR (EVAL <(CAR X) "Boolean E>)))
          (RETURN <(LISPAPPLY (CADR X) L) MD E>))))
  (RETURN NIL) )))

(EVLAPPLY (LAMBDA (VL B ARGL M E) (PROGN
  (LET XE (EXTEND E))
  (REPEAT ((IN A ARGL) (IN X VL))
    (LET XE (BIND (CAR X) XE (LIST
             (CONS "MODE (CDR X))
             (CONS "VALUE (EVAL <A (CDR X) E>))))))
  (EVAL <B M XE>) )))
```

0.1.10 Explanation

EVAL takes a triple T consisting of an expression X, a "target" mode M, and an environment E, and returns either a triple or else NIL indicating failure to produce such a triple. If X is an atom, a triple TP is created, then EVCONVERTed to M. There are three ways that triples are produced from atomic X. If X is already a triple (remember, triples are atomic), then an evaluation function associated with M is applied to evaluate X in the new environment E (for example, if M is SEXPR then that evaluation function is EVAL). If X is a symbol, then TP is produced by EVSYMBOL. Otherwise, the atom is a basic object for which the function MODE can produce the triple directly.

If X is not an atom but its first element OP is, then there are three cases. First, symbol OP may be a special MODLISP operator such as EXIT, IF, LAMBDA, QUOTE, RETURN, SETQ, in which case the name FN stored under property SPECIAL is applied to the list of arguments of X. If OP has a MODEMAP property, then the value stored under that property is a list of modemaps. Each modemap is applied in turn by EVFORM attempting to evaluate form X successfully to produce a triple Z. If both of the above two cases fail, EVAL returns NIL.

If neither X nor its CAR is atomic, then two cases remain. If the CAR of OP is "LAMBDA, then EVLAPPLY is called. Otherwise, the OP is evaluated with target mode SEXPR to produce a new OP which is consed onto the list of arguments and passed recursively to EVAL.

EVSYMBOL takes a symbol S and an environment E, and returns either a triple or NIL. If S has a VALUE property[10] then a triple stored under that property is returned. If S has a MODE property but not a VALUE, then EVSYMBOL fails. Otherwise, a triple with mode SYMBOL is created directly for S.

EVCONVERT takes a triple T and a mode MP, and produces either another triple or NIL. In a system without partial declarations (for example, all conventional "typed" languages), EVCONVERT is equivalent to the conversion function COERCE described below.

EVFORM takes two arguments: a triple T=<X TM E>, where X consists of a (generic) OPerator consed onto a list ARGL of arguments, and, a Modemap MAP consisting of a "signature pattern" SIG consed onto a conditional expression CEXPR. The SIG is a list of patterns whose CAR gives a "mode-pattern" of the result and whose CDR gives the mode-patterns for the arguments in left-to-right order. First, an attempt is made to MATCH SIG's result mode to the mode TM of the triple. If this match fails, then EVFORM returns NIL. The substitution list resulting from a successful match is SUBLISted into the signature pattern to produce a signature MAP containing no free variables. Next, each argument in ARGL is evaluated in turn with the corresponding mode from the new signature; if all evaluations are successful, a list L of evaluated arguments is produced. The CDR of the new MAP is a conditional expression list, each element of which is a list (Boolean expression, function descriptor). If the Boolean expression can be successfully EVALed with mode Boolean to yield "true, then the function descriptor is consed onto L and LISPEVALed.

EVLAPPLY takes a triple T consisting of a LAMBDA-expression with symbol list VL and a body B consed onto a list of arguments ARGL as the expression part, a mode, and an environment. Here, VL is a list of pairs each consisting of a formal parameter consed with a mode. First, XE is created by extending E with a new (empty) local environment. Next, XE is incrementally extended by adding a binding for each successive member of VL. Each

[10] As set by an assignment (the SETQ special form) or by a binding (function EVLAPPLY).

binding in MODLISP is a property list consisting of a MODE m together with a VALUE consisting of a triple created by evaling the corresponding member of ARGL using m as target mode, and the original E as the environment. Having extended XE to include bindings for all the formal parameters of VL, a triple is formed by evaluating the body of the LAMBDA-expression in that extended environment.

The following auxillary functions are used:

(BIND S E PL) adds a property list PL as a binding for symbol S in the current local environment of E and returns the updated E.

(COERCE T M) converts triple T to one with mode M, or else returns NIL if that conversion is not possible.

(EXTEND E) extends E to contain a new empty local environment.

(GET S P E) returns value stored under property P for symbol S in environment E.

(GETEVALFUN M) returns the evaluator function for mode M.

(MATCH E P) returns a list of substitution pairs ((A . B) ..) if pattern P (containing "free variables" A,..) matches expression E (that is, P=E when all A's in P are replaced by corresponding B's)

(LISPAPPLY FN L) applies MODLISP function FN to a list L of evaluated arguments by a LISP evaluation of (CONS (CAR FN) (APPEND L (LIST (CDR FN)))). In MODLISP, all functions are represented as pairs whose CAR is a code pointer and whose CDR is a run-time environment (as defined by the modemap) which will be supplied to FN as its last argument when FN is called.

(MODE X) returns the mode of X, assumed to be a built-in type

(S:EXPR T) returns the expression part of triple T.

(S:MODE T) returns the mode part of triple T.

(S:ENV T) returns the environment part of triple T.

(SYMBOLP X) tests that X is a symbol.

(TRIPLEP X) tests that X is a triple.

(COMPILE FN ARGS) compiles a call to the function FN, with argumets ARGS.

(COMPILEDP X) test if X represents a piece of compiled code, rather than a value.

0.1.11 On Extensions for Partial Compilation

In order to cope with partial compilation, several modifications to the evaluator must be made. The first, and easiest, is that the fourth line of EVSYMBOL should be changed to return a compiled reference to X rather than returning NIL. The other changes are to EVFORM. The loop in lines 5-8 should be modified to set the flag EVALFG if some (S:MODE

T) is "NilMode, COMPFG, if some (S:EXPR T) represents compiled code. In addition, the call to LISPAPPLY on the last line should be replaced by a conditional expression (COND (EVALFG "CALLEVAL) (COMPFG "COMPILE) ("T "LISPAPPLY)). These changes to EVFORM discriminate (in reverse order) between the 3 cases described in the section "Compilation" in the body of the paper. To recap, if any of the arguments has an unknown mode (indicated by a return mode of "NilMode), then all we can do is call the evaluator at runtime, when the modes are known. If the modes are known, but some of the values are not (i.e. the expression part of at least one triple represents a piece of compiled code, as determined by COMPILEDP), then we can compile a call to the function determined from the modemap. Finally, if all the values are known, then we can proceed by calling LISPEVAL, just as in the case of the basic evaluator without the additions for partial compilation.

0.1.12 On Extensions for Partial Declarations

In the general case where partial declarations are allowed [Cohe80], the definition of EVCONVERT must be replaced by:

```
(EVCONVERT (LAMBDA ((: T <X M E>) MP)
  (COERCE T (OR (RESOLVE M MP) (RETURN NIL))) ))
```

Here, the mode M of T is RESOLVEd with MP to produce a third mode M' (if no such mode can be found, EVCONVERT returns NIL immediately). The expression X of T is then COERCEd to M' producing a final triple, or NIL if that is not possible.

In addition, the description of EVFORM becomes somewhat more complicated. First of all, it is not necessary for the result mode of SIG to match successfully the target mode TM of the triple. As a result SIG will generally contain free variables throughout the process of left-to-right argument evaluation. This process has the following alternative code replacing the loop in lines 5-8 of EVFORM:

```
(LET LT (COLLECT ((IN A ARGL) (IN P (CDR SIG)))
  (PROGN
    (LET MD (PATTERN_TO_MODE (SUBLIS LS P)))
    (LET T (OR (EVAL <A MD E>) (RETURN NIL)))
    (LET E (S:ENV T))
    (LET LS (NCONC (MATCH (S:MODE T) P) LS))
    T)))

(LET L (COLLECT ((IN T TL)
                 (IN MD (SUBLIS LS (CDR SIG))))
  (S:EXPR (OR (COERCE T MD) (RETURN NIL)))))
```

This is described as follows. First, a list LT of triples is produced. The first argument A of ARGL is evaluated in the initial environment with a target mode obtained by SUBLISting LS (initially NIL[11]) into P and converting the result to a mode by PATTERN_TO_MODE. If evaluation successfully produces a triple T, the initial environment is replaced by the environment of T, the substitution list LS is updated to contain new bindings which result from MATCHing P to the mode of T. A second argument is then evaluated with a target

[11] This initial setting determines a *bottom-up* evaluation strategy, where the mode of a result is determined purely by the modes of its arguments. This clearly creates difficulties for functions without arguments. The strategy actually used in MODLISP is first to try a *top-down* evaluation, and then to use a bottom-up strategy if the top-down one fails. The top-down strategy is implemented by setting the initial value of LS to be the result of MATCHing the mode-pattern of the result (CAR SIG) with the target mode TM.

mode determined from the evaluation of the first argument, etc. until a triple is either produced for each argument or else NIL is returned along the way.

The list LT now consists of triples containing values whose modes are specific domains. All relevant free variables in the final LS are paired with domains, with later pairings (those which appear left-most in LS) taking precedence. A final signature is determined therefore by SUBLISting LS into (CDAR MAP). A pass over TL is then made to COERCE all expressions to the appropriate domain.

0.1.13 Appendix B: Sample Compilation by the MODLISP Evaluator

The following illustrates a sample source language function definition and corresponding compilation by the current MODLISP evaluator into LISP code. The example describes a function to raise an expression to a power using a repeated squaring method. The domain of the expression, r, is assumed to be in the category Ring, the power assumed to be a NonNegativeInteger. The example is taken from the context of the definition of the category Ring in our experimental computer algebra system.

```
Source language:
  given
    x: r
    n: NonNegativeInteger
  define
    x ** n ->
      if n = 0 then 1
        else if n = 1 then x
  else if oddp(n) then x*((x*x)**((n-1)/2))
    else (x*x)**(n/2).
```

Compilation into LISP:

```
(LAMBDA (x n r) ((LAMBDA (G4 G1 G2 G0 G3)
  (COND
    ((EQUAL n 0) (ELT r 10))
    ((EQUAL n 1) x)
    ((oddp n)
      ((CAR (SETQ G2 (ELT r 9)))
        x
((CAR (SETQ G1 (ELT r 11)))
  ((CAR (SETQ G0 (ELT r 9))) x x (QCDR G0))
  (QUOTIENT (DIFFERENCE n 1) 2) (QCDR G1))
(QCDR G2)) )
    ("T
      ((CAR (SETQ G4 (ELT r 11)))
        ((CAR (SETQ G3 (ELT r 9))) x x (QCDR G3))
        (QUOTIENT n 2)
        (QCDR G4))) ))
  NIL NIL NIL NIL NIL ))
```

Explanation. A representation of the ring r is passed as a third argument to the function. Rings are represented by vectors of length 11 (or more, if they are of a more restrictive category). Elements 5 through 11 are allocated as follows:

```
(5) the zero element of the ring
```

```
(6) dyadic +
(7) monadic -
(8) dyadic -
(9) dyadic *
(10) the multiplicative identity (1)
(11) dyadic **
```

Each of the slots 6-9 and 11 are pairs consisting of a code pointer in its CAR and a pointer to the vector representing r in its CDR. In particular, the CAR of the contents of slot 11 in the vector is a pointer to the compiled code of the above function!

The function QCDR is a version of CDR that does not check to see if its argument is a pair first, and hence it can compile into 1 machine instruction. Note that this function is only used in circumstances where the CAR has already been taken, so we know that the argument really is a pair. (ELT v n) extracts the n-th element (counting from 0) of the vector v. The symbols G0-G4 represent "gensyms" generated by MODLISP.

We note that there are several optimizations which could be performed on the code to improve performance and to reduce the size of the compiled code, such as replacing EQUAL by EQ, G2 and G3 by G0, G4 by G1, the inner (CAR (SETQ G0..)) by (QCAR (SETQ G0..)), all (ELT r 9) by (QELT r 9). These optimisations which are planned, are global in nature. The evaluator already performs several local optimisations, such as the use of QCDR.

0.2 Tedious Maintainer Tasks

0.2.1 Maintaining the credits list

Asiom tried hard to maintain a list of credits for people who contribute to the effort. The contributions range from people who developed the original code to the administrative assistant who maintained the newsletter to people who set up a vising scholar position to people who wrote algebra code. In short, the list of people who contributed is varied. Credit should be wildely shared.

The credits list is maintained in several places for several reasons.

1) books/bookheader.tex is the prefix to every book. Every book has the credits list.

2) books/bookvol5 (interpreter) has the credits list embedding in the source code.

3) bookvol10.4 has the credits list as a unit test

4) src/input/unittest1.input tests the)credits command

0.3 What is the purpose of the HACKPI domain?

HACKPI is a hack provided for the benefit of the axiom interpreter. As a mathematical type, it is the simple transcendental extension `Q(\pi)` of the rational numbers. This type allows interactive users to use the name `'%pi'` without a type both where a numerical value is expected [as in `draw(sin x,x=-%pi..%pi)`] or when the exact symbolic value is meant. The interpreter defaults a typeless `%pi` to HACKPI and then uses the various conversions to cast it further as required by the context.

One could argue that it is unfair to single `%pi` out from other constants, but it occurs frequently enough in school examples (specially for graphs) so it was worth a special hack. In a non-interactive environment (library), HACKPI would not exist.

(Manuel Bronstein)

0.4 How Axiom Builds

0.4.1 The environment variables

Axiom uses a tree of Makefiles to build the system. Each Makefile is created from the literate file (Makefile.pamphlet) and then executed.

In order to have a complete set of variables we create an "environment" that contains all of the shell variables (except the AXIOM variable).

These can be changed on the command line at the time of the top level "make" command. One common usage pattern is to override the NOISE variable. This variable controls whether we see the full output or just the echo of each individual step. Sometimes a build fails at a step and we would like to know the details. By default they are written to `$TMP/trace` but we can watch every detail with the command line:

```
make NOISE=
```

This overrides the output file and writes everything to the console.

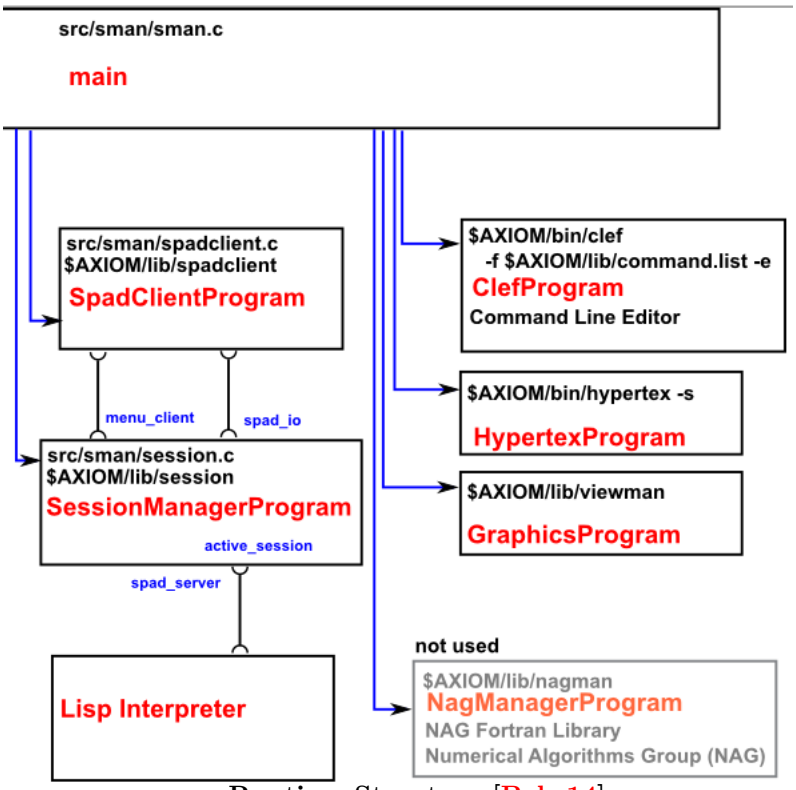
Another common usage pattern is to override the tests that are run. By default, all tests are run. This can be very time consuming. A particular subset can be run or, using the option "notests", none will be run:

```
make TESTSET=notests
```

```
AWK=gawk
BOOKS=/research/test/books
BYE=bye
CC=gcc
CCF=-O2 -fno-strength-reduce -Wall -D_GNU_SOURCE -DLINUXplatform
    -I/usr/X11/include
COMMAND=/usr/local/axiom/mnt/ubuntu/bin/axiom
DAASE=/research/test/src/share
DESTDIR=/usr/local/axiom
DOCUMENT=/research/test/mnt/ubuntu/bin/document
GCLDIR=/research/test/lsp/gcl-2.6.8pre4
GCLOPTS=--enable-vssize=65536*2 --enable-locbfd --disable-dynsysbfd
        --disable-statsysbfd --enable-maxpage=512*1024 --disable-xgcl
        --disable-tkconfig
GCLVERSION=gcl-2.6.8pre4
INC=/research/test/src/include
INT=/research/test/int
LDF= -L/usr/X11R6/lib -L/usr/lib -lXpm
LISP=lsp
LSP=/research/test/lsp
MNT=/research/test/mnt
NOISE=-o /research/test/obj/tmp/trace
O=o
```

```
OBJ=/research/test/obj
PART=cprogs
PATCH=patch
PLF=LINUXplatform
RANLIB=ranlib
RUNTYPE=serial
SPAD=/research/test/mnt/
SPADBIN=/research/test/mnt/ubuntu/bin
SPD=/research/test
SRC=/research/test/src
SRCDIRS=interpdir sharedir algebradir etcdir clefdir docdir graphdir
        smandir hyperdir browserdir inputdir
SUBPART=everything
SYS=ubuntu
TANGLE=/research/test/mnt/ubuntu/bin/lib/notangle
TAR=tar
TESTSET=none
TMP=/research/test/obj/tmp
TOUCH=touch
UNCOMPRESS=gunzip
VERSION=Axiom (May 2010)
WEAVE=/research/test/mnt/ubuntu/bin/lib/noweave
XLIB=/usr/X11R6/lib
ZIPS=/research/test/zips
```

0.5 The runtime structure of Axiom



Runtime Structure [Bake14]

0.5.1 The build step

This shows the steps taken to build Axiom in the sequence they happen. Each level of indentation is another level of Makefile being executed.

```
Makefile
  1 noweb
  2 copy SRC/scripts to AXIOM/bin
  3 extract Makefile.SYS from Makefile.pamphlet
  4 latex SRC/input/*.input.pamphlet
  5 extract SRC/algebra/Makefile.help from SRC/algebra/Makefile.pamphlet
    5a make SRC/algebra/Makefile.help parallelhelp
       5a1 extract syntax help from BOOKS/bookvol5
       5a2 extract help files from BOOKS/bookvol10.*
  6 extract BOOKS/Makefile from BOOKS/Makefile.pamphlet
    6a make BOOKS/Makefile
       6a1 copy SRC/scripts/tex/axiom.sty to AXIOM/doc
       6a2 create AXIOM/doc/*.pdf
           6a2a copy book*.pamphlet to AXIOM/doc
           6a2b extract latex for each book
           6a2c latex each book
           6a2d dvipdfm each dvi file
```

```
      6a3 make AXIOM/doc/toc.pdf
 7 extract AXIOM/doc/hypertex/Makefile1 from BOOKS/bookvol11
   7a make AXIOM/doc/hypertex/Makefile1
      7a1 extract all xhtml pages to AXIOM/doc/hypertex
      7a2 extract axiom1.bitmap from BOOK/bookvol11
      7a3 extract rcm3720.input from BOOK/bookvol11
      7a4 extract strang.input from BOOK/bookvol11
      7a5 extract signatures.txt from BOOK/bookvol11
      7a6 copy BOOKS/ps/doctitle.png to AXIOM/doc/hypertex
      7a7 copy BOOKS/ps/lightbayou.png to AXIOM/doc/hypertex
 8 make Makefile.SYS
   8a create the root directories
   8b create noweb if needed
   8c extract SRC/Makefile from SRC/Makefile.pamphlet
   8d make SRC/Makefile setup
      8d1 extract SRC/scripts/Makefile from SRC/scripts/Makefile.pamphlet
      8d2 make SRC/scripts/Makefile
         8d1a copy all scripts to AXIOM/bin
      8d3 extract SRC/lib/Makefile from SRC/lib/Makefile.pamphlet
      8d4 make SRC/lib/Makefile
         8d4a compile INT/lib/bsdsignal.c
         8d4b compile INT/lib/cursor.c
         8d4c compile INT/lib/edin.c
         8d4d compile INT/lib/fnct-key.c
         8d4e compile INT/lib/halloc.c
         8d4f compile INT/lib/openpty.c
         8d4g compile INT/lib/pixmap.c
         8d4h compile INT/lib/prt.c
         8d4i compile INT/lib/sockio-c.c
         8d4j compile INT/lib/spadcolors.c
         8d4k compile INT/lib/util.c
         8d4l compile INT/lib/wct.c
         8d4m compile INT/lib/xdither.c
         8d4n compile INT/lib/xshade.c
         8d4o compile INT/lib/xspadfill.c
         8d4p create libspad.a
         8d4q compile INT/lib/cfuns-c.c
         8d4r compile INT/lib/hash.c
         8d4s latex all files to INT/doc/src/lib
   8e extract LSP/Makefile from LSP/Makefile.pamphlet
   8f make LSP/Makefile gcldir
      8f1 untar ZIPS/gcl
      8f2 apply Axiom patches to gcl
      8f3 copy gcl_collectfn.lsp to OBJ/SYS/lsp
      8f4 copy sys-proclaim.lisp to OBJ/SYS/lsp
      8f5 make LSP/GCLVERSION/Makefile
      8f6 add BOOKS/tangle.lsp to gcl to create INT/SYS/lisp
   8g make SRC/Makefile
      8g1 make stanzas from SRCDIRS
         8g1a interpdir
           8g1a1 copy bookvol5 to src/interp
           8g1a2 copy bookvol9 to src/interp
           8g1a3 copy bookvol10.5 to src/interp
           8g1a4 extract util.ht from BOOKS/bookvol7.1 to AXIOM/doc
```

```
  8g1a5 make SRC/interp/Makefile
    8g1a5a build SAVESYS=OBJ/SYS/bin/interpsys
      8g1a5a1 build DEPSYS=OBJ/SYS/bin/depsys
      8g1a5a2 compile all interp files
      8g1a5a3 call build-interpsys to make SAVESYS
      8g1a5a4 build warm.data
      8g1a5a5 build SAVESYS
      8g1a5a6 copy SAVESYS to AXIOMSYS=AXIOM/bin/AXIOMsys
8g1b sharedir
  8g1b1 make share/Makefile
    8g1b1a copy SRC/share/algebra/command.list to AXIOM/lib
8g1c algebradir
  8g1c1 extract algebra/Makefile from SRC/algebra/Makefile.pamphlet
  8g1c2 copy bookvol10.2 to SRC/algebra
  8g1c3 copy bookvol10.3 to SRC/algebra
  8g1c4 copy bookvol10.4 to SRC/algebra
  8g1c5 copy bookvol10.5 to SRC/algebra
  8g1c6 extract 'findAlgebraFiles'
          from SRC/algebra/Makefile.pamphlet
  8g1c7 execute findAlgebraFiles and append output
          to SRC/algebra/Makefile
  8g1c8 make SRC/algebra/Makefile
    8g1c8a build INT/algebra nrlibs
    8g1c8b copy SRC/algebra/libdb.text to AXIOM/algebra
    8g1c8c construct AXIOM/bin/index.html
    8g1c8d copy SRC/share/algebra/gloss.text AXIOM/algebra
    8g1c8e copy SRC/share/algebra/glossdef.text AXIOM/algebra
    8g1c8f copy SRC/share/algebra/glosskey.text AXIOM/algebra
8g1d etcdir
  8g1d1 extract SRC/etc/Makefile from SRC/etc/Makefile.pamphlet
  8g1d2 make etc/Makefile
    8g1d2a copy SRC/doc/gloss.text INT/algebra
    8g1d2b copy SRC/doc/topics.data INT/algebra
    8g1d2c call make-databases
    8g1d2b copy INT/algebra/*.daase AXIOM/algebra
    8g1d2e compile asq.c
    8g1d2f copy OBJ/SYS/etc/asq AXIOM/bin
    8g1d2g copy SRC/etc/summary AXIOM/lib
    8g1d2h copy SRC/etc/copyright AXIOM/lib
8g1e clefdir
  8g1e1 extract SRC/clef/Makefile from SRC/clef/Makefile.pamphlet
  8g1e2 make clef/Makefile
    8g1e2a extract edible.c to OBJ/SYS/clef
    8g1e2b compile OBJ/SYS/clef/edible.c
    8g1e2c link edible, fnct-key, edin, bsdsignal, prt, wct,
               openpty, cursor into AXIOM/bin/clef
8g1f docdir
  8g1f1 extract SRC/doc/Makefile from SRC/doc/Makefile.pamphlet
  8g1f2 make SRC/doc/Makefile
    8g1f2a extract SRC/doc/axiom.bib to INT/doc
    8g1f2b extract SRC/doc/axiom.sty to AXIOM/bin/tex
    8g1f2c extract SRC/doc/refcard.dvi to AXIOM/doc
    8g1f2d extract SRC/doc/endpaper.dvi to AXIOM/doc
    8g1f2e copy SRC/doc/ps/* to AXIOM/doc/ps
```

```
8g1k2a copy SRC/input/*.input INT/input
8g1k2b lisp tangle input files from SRC/input/*.input.pamphlet
8g1k2c extract INT/input/Makefile
         from SRC/input/Makefile.pamphlet
8g1k2d make INT/input/Makefile TESTSET
  8g1k2d1 run regresstests
  8g1k2d2 run catstests
  8g1k2d3 run richtests
  8g1k2d4 run regression tests
  8g1k2d5 extract INT/input/Makefile.algebra
            from SRC/algebra/Makefile.pamphlet
  8g1k2d6 make INT/input/Makefile.algebra
```

0.5.2 Where each output file is created

Here we show which step in the above set of actions creates the file that ends up in the final ship directory. We break it down by subdirectory in the final image.

AXIOM/algebra

```
in AXIOM/algebra:
  *.o
  browse.daase
  category.daase
  compress.daase
  dependents.daase
  interp.daase
  operation.daase
  users.daase
```

AXIOM/autoload

```
in AXIOM/autoload:
  ax.o
  bc-matrix.o
  br-con.o
  ht-util.o
  mark.fn
  mark.o
  nag-c02.o
  nag-c05.o
  nag-c06.o
  nag-d01.o
  nag-d02.o
  nag-d03.o
  nag-e01.o
  nag-e02.o
  nag-e04.o
  nag-f01.o
  nag-f02.o
```

```
  nag-f04.o
  nag-f07.o
  nag-s.o
  nspadaux.o
  pspad1.fn
  pspad1.o
  pspad2.fn
  pspad2.o
  topics.o
  wi1.fn
  wi1.o
  wi2.fn
  wi2.o
```

AXIOM/bin

```
in AXIOM/bin:
  asq            8g1d2f  copy OBJ/SYS/etc/asq AXIOM/bin
  axiom          8g1h2k  chmod axiom shell script to be executable
  axiom.sty      6a1     copy SRC/scripts/tex/axiom.sty to AXIOM/doc
  AXIOMsys       8g1a5a6 copy SAVESYS to AXIOMSYS=AXIOM/bin/AXIOMsys
  booklet        8g1f2i  copy booklet to AXIOM/bin
  boxhead        2       copy SRC/scripts to AXIOM/bin
  boxtail        2       copy SRC/scripts to AXIOM/bin
  boxup          2       copy SRC/scripts to AXIOM/bin
  clef           8g1e2c  link edible, fnct-key, edin, bsdsignal, prt, wct,
                            openpty, cursor into AXIOM/bin/clef
  document       2       copy SRC/scripts to AXIOM/bin
  htadd          8g1i2c  extract and compile AXIOM/bin/htadd
  htsearch       8g1i2e  extract and compile AXIOM/bin/htsearch
  hypertex       8g1i2g  extract and compile AXIOM/bin/hypertex
  index.html     8g1c8c  construct AXIOM/bin/index.html
  lib            1       noweb
    btdefn
    cpif
    emptydefn
    finduses
    h2a
    htmldoc
    markup
    mnt
    nodefs
    noidx
    noindex
    noroff
    noroots
    notangle
    nountangle
    noweave
    noweb
    nt
    nuweb2noweb
```

```
    numtime
    pipedocs
    tmac.w
    toascii
    tohtml
    toroff
    totex
    unmarkup
  Makefile.pamphlet
  man
    man1
      cpif.1
      htmltoc.1
      nodefs.1
      noindex.1
      noroff.1
      noroots.1
      notangle.1
      nountangle.1
      noweave.1
      noweb.1
      nuweb2noweb.1
      sl2h.1
    man7
      nowebfilters.7
      nowebstyle.7
  showdvi            2       copy SRC/scripts to AXIOM/bin
? sman               8g1h2i  link OBJ/SYS/sman/sman.o to AXIOM/lib/sman
  SPADEDIT           2       copy SRC/scripts to AXIOM/bin
  tex                2       copy SRC/scripts to AXIOM/bin
    axiom.sty        8g1f2b  extract SRC/doc/axiom.sty to AXIOM/bin/tex
                     2       copy SRC/scripts to AXIOM/bin
    noweb.sty        1       noweb
    nwmac.tex        1       noweb
? viewalone          8g1g2d  compile and link AXIOM/lib/viewalone
```

AXIOM/doc

```
AXIOM/doc:
  axbook
    *.xhtml
  axiom.sty
  bigbayou.png
  bitmaps
    *.bitmap
  bookvol0.out
  bookvol0.pdf
  bookvol0.toc
  bookvol10.1.out
  bookvol10.1.pdf
  bookvol10.1.toc
  bookvol10.2.out
```

```
  ht.db
  hypertex
    *.xhtml
  msgs
    s2-us.msgs
  ps
    *.ps
  refcard.dvi
  rosetta.dvi
  spadhelp
    *.help
  src
?    algebra
    algebra.Makefile.dvi
    books.Makefile.dvi
    clef
      axiom.sty
      edible.c.dvi
    clef.Makefile.dvi
    doc.Makefile.dvi
    etc.Makefile.dvi
?   hyper
    input
      *.input.dvi
    input.Makefile.dvi
?   interp
    interp.Makefile.dvi
    lib
      *.c.dvi
    lib.Makefile.dvi
    Makefile.dvi
    root.Makefile.dvi
    scripts.Makefile.dvi
    share.Makefile.dvi
?   sman
    src.Makefile.dvi
  toc.pdf
  util.ht
  viewports
    *.view
      data
      graph0
      image.bm
      image.xpm
```

AXIOM/graph

```
AXIOM/graph
  parabola.view:     8g1g2e extract AXIOM/graph/parabola.view from bookvol8
    data
    graph0
```

AXIOM/input

```
AXIOM/input:
  *.input files
```

AXIOM/lib

```
AXIOM/lib:
  command.list       8g1b1a copy SRC/share/algebra/command.list to AXIOM/lib
  copyright          8g1d2h copy SRC/etc/copyright AXIOM/lib
  ex2ht              8g1i2b extract and compile AXIOM/lib/ex2ht
  graph              8g1g2f extract psfiles from bookvol8 to AXIOM/lib/graph
    colorpoly.ps
    colorwol.ps
    drawarc.ps
    drawcolor.ps
    drawIstr.ps
    drawline.ps
    drawlines.ps
    drawpoint.ps
    draw.ps
    drawrect.ps
    drawstr.ps
    drwfilled.ps
    end.ps
    fillarc.ps
    fillpoly.ps
    fillwol.ps
    header.ps
    setup.ps
  hthits         8g1i2d extract and compile AXIOM/lib/hthits
  presea         8g1i2f extract and compile AXIOM/lib/presea
  session        8g1h2c link OBJ/SYS/sman/session.o to AXIOM/lib/session
  spadbuf        8g1i2a extract and compile AXIOM/lib/spadbuf
  spadclient     8g1h2f link OBJ/SYS/sman/spadclient.o AXIOM/lib/spadclient
  SPADEDIT
  summary        8g1d2g copy SRC/etc/summary AXIOM/lib
  view2d         8g1g2b compile and link AXIOM/lib/view2d
  view3d         8g1g2c compile and link AXIOM/lib/view3d
  viewman        8g1g2a compile and link AXIOM/lib/viewman
```

AXIOM/src

```
AXIOM/src:
? algebra
```

AXIOM/timestamp

```
AXIOM/timestamp
```

0.6 How Axiom Works

0.6.1 Input and Type Selection

First we change the default setting for autoload messages to turn off the noise of file loading from the library:

```
(1) -> )set mes auto off
```

Next we tell the interpreter to show us the modemaps used to classify input and select types. This is known as "bottomup" messages. We can watch the interpreter ponder the input.

```
(1) -> )set mes bot on
```

Now we give it something nontrivial to ponder.

```
(1) -> f:=1/(a*x+b)
```

After parsing the input Axiom begins to figure out the type of the expression. In this case it starts with the multiply operator in the denominator.

Axiom has determined that "a" is of type VARIABLE and "x" is of type VARIABLE. It is looking for function of the form

```
    VARIABLE * VARIABLE
```

so it looks in the domain of the left argument "a" which is VARIABLE and does not find the required function. Similarly it looks in the domain of the right argument "x" which is VARIABLE and, not surprisingly, does not find the required function.

It tried to promote each VARIABLE to SYMBOL and looks for a way to mulitply VARIABLES and SYMBOLS or SYMBOLS and SYMBOLS. Neither succeeds.

```
 Function Selection for *
      Arguments: (VARIABLE a,VARIABLE x)
   -> no appropriate * found in Variable a
   -> no appropriate * found in Variable x
   -> no appropriate * found in Symbol
   -> no appropriate * found in Variable a
   -> no appropriate * found in Variable x
   -> no appropriate * found in Symbol

 Modemaps from Associated Packages
   no modemaps
```

Since it cannot find a specific modemap that uses the exact types it now expands the search to look for the general modemaps. It searches these modemaps in order to try to find one that fits.

```
 Remaining General Modemaps
   [1] (D,D1) -> D from D
            if D has XFALG(D2,D1) and D2 has ORDSET and D1 has RING
```

The first match will fail because Symbol does not have RING. We can determine this by asking the interpreter:

```
SYMBOL has RING

   (1)  false
                                                    Type: Boolean
```

The following modemaps will fail for various similar reasons:

```
[2] (D1,D) -> D from D
        if D has XFALG(D1,D2) and D1 has ORDSET and D2 has RING
[3] (Integer,D) -> D from D
        if D has VECTCAT D2 and D2 has TYPE and D2 has ABELGRP
[4] (D1,D) -> D from D
        if D has VECTCAT D1 and D1 has TYPE and D1 has MONOID
[5] (D,D1) -> D from D
        if D has VECTCAT D1 and D1 has TYPE and D1 has MONOID
[6] (D,D1) -> D1 from D
        if D has SMATCAT(D2,D3,D4,D1) and D3 has RING and D4 has
        DIRPCAT(D2,D3) and D1 has DIRPCAT(D2,D3)
[7] (D1,D) -> D1 from D
        if D has SMATCAT(D2,D3,D1,D4) and D3 has RING and D1 has
        DIRPCAT(D2,D3) and D4 has DIRPCAT(D2,D3)
[8] (D,D) -> D from D if D has SGROUP
[9] (D,D1) -> D from D if D has RMODULE D1 and D1 has RNG
[10] (D,D) -> D from D if D has MONAD
[11] (D,D) -> D from D
        if D has MATCAT(D1,D2,D3) and D1 has RING and D2 has FLAGG
        D1 and D3 has FLAGG D1
[12] (D1,D) -> D from D
        if D has MATCAT(D1,D2,D3) and D1 has RING and D2 has FLAGG
        D1 and D3 has FLAGG D1
[13] (D,D1) -> D from D
        if D has MATCAT(D1,D2,D3) and D1 has RING and D2 has FLAGG
        D1 and D3 has FLAGG D1
[14] (Integer,D) -> D from D
        if D has MATCAT(D2,D3,D4) and D2 has RING and D3 has FLAGG
        D2 and D4 has FLAGG D2
[15] (D,D1) -> D1 from D
        if D has MATCAT(D2,D3,D1) and D2 has RING and D3 has FLAGG
        D2 and D1 has FLAGG D2
[16] (D1,D) -> D1 from D
        if D has MATCAT(D2,D1,D3) and D2 has RING and D1 has FLAGG
        D2 and D3 has FLAGG D2
[17] ((D5 -> D6),(D4 -> D5)) -> (D4 -> D6) from MappingPackage3(D4,
        D5,D6)
        if D4 has SETCAT and D5 has SETCAT and D6 has SETCAT
[18] (D1,D) -> D from D if D has LMODULE D1 and D1 has RNG
[19] (PolynomialIdeals(D1,D2,D3,D4),PolynomialIdeals(D1,D2,D3,D4))
         -> PolynomialIdeals(D1,D2,D3,D4)
        from PolynomialIdeals(D1,D2,D3,D4)
        if D1 has FIELD and D2 has OAMONS and D3 has ORDSET and D4
        has POLYCAT(D1,D2,D3)
[20] (D1,D) -> D from D
        if D has GRMOD(D1,D2) and D1 has COMRING and D2 has ABELMON

[21] (D,D1) -> D from D
        if D has GRMOD(D1,D2) and D1 has COMRING and D2 has ABELMON

[22] (D1,D2) -> D from D
        if D has FMCAT(D1,D2) and D1 has RING and D2 has SETCAT
[23] (D1,D2) -> D from D
```

```
            if D has FAMONC(D2,D1) and D2 has SETCAT and D1 has CABMON

   [24] (Equation D1,D1) -> Equation D1 from Equation D1
            if D1 has SGROUP and D1 has TYPE
   [25] (D1,Equation D1) -> Equation D1 from Equation D1
            if D1 has SGROUP and D1 has TYPE
   [26] (D,D1) -> D from D
            if D has DIRPCAT(D2,D1) and D1 has TYPE and D1 has MONOID

   [27] (D1,D) -> D from D
            if D has DIRPCAT(D2,D1) and D1 has TYPE and D1 has MONOID

   [28] (DenavitHartenbergMatrix D2,Point D2) -> Point D2
            from DenavitHartenbergMatrix D2
            if D2 has Join(Field,TranscendentalFunctionCategory)
   [29] (PositiveInteger,Color) -> Color from Color
   [30] (DoubleFloat,Color) -> Color from Color
   [31] (CartesianTensor(D1,D2,D3),CartesianTensor(D1,D2,D3)) ->
            CartesianTensor(D1,D2,D3)
            from CartesianTensor(D1,D2,D3)
            if D1: INT and D2: NNI and D3 has COMRING
   [32] (PositiveInteger,D) -> D from D if D has ABELSG
   [33] (NonNegativeInteger,D) -> D from D if D has ABELMON
   [34] (Integer,D) -> D from D if D has ABELGRP
```

Eventually the interpreter decides that it can coerce Symbol to Polynomial(Integer). We can do this in the interpreter also:

```
a::Symbol::POLY(INT)

   (1)  a
                                                    Type: Polynomial Integer
```

And the interpreter can find multiply in POLY(INT):

```
 [1]  signature:   (POLY INT,POLY INT) -> POLY INT
      implemented: slot $$$ from POLY INT
 [2]  signature:   (POLY INT,POLY INT) -> POLY INT
      implemented: slot $$$ from POLY INT
```

We can see this signature exists by asking the interpreter to show us the domain POLY(INT) (truncated here for brevity):

```
)show POLY(INT)
 Polynomial Integer is a domain constructor.
 Abbreviation for Polynomial is POLY
 This constructor is exposed in this frame.
 Issue )edit src/algebra/POLY.spad to see algebra source code for POLY

------------------------------ Operations ------------------------------

 ?*? : (Fraction Integer,%) -> %       ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?*? : (%,Fraction Integer) -> %
 ?*? : (%,Integer) -> %                ?*? : (%,%) -> %
```

Having found multipy the interpreter now starts a search for the operation

```
    (POLY(INT)) + (VARIABLE)
```

It cannot find this modemap

```
Function Selection for +
     Arguments: (POLY INT,VARIABLE b)
  -> no appropriate + found in Polynomial Integer
  -> no appropriate + found in Variable b
  -> no appropriate + found in Variable b
```

so it promotes VARIABLE to POLY(INT) and finds the operation:

```
  (POLY(INT)) + (POLY(INT))

[1]  signature:   (POLY INT,POLY INT) -> POLY INT
     implemented: slot $$$ from POLY INT
```

Next it tackles the division operation where the numerator is PI (PositiveInteger) and the denominator is POLY(INT). It tries to find

```
   (PI) / (POLY(INT))
```

in PositiveInteger, Polynomial Integer and Integer. All attempts fail.

```
Function Selection for /
     Arguments: (PI,POLY INT)
  -> no appropriate / found in PositiveInteger
  -> no appropriate / found in Polynomial Integer
  -> no appropriate / found in Integer
  -> no appropriate / found in PositiveInteger
  -> no appropriate / found in Polynomial Integer
  -> no appropriate / found in Integer

Modemaps from Associated Packages
  no modemaps
```

So now it turns to the general modemaps:

```
Remaining General Modemaps
  [1] (D,D1) -> D from D if D has VSPACE D1 and D1 has FIELD
  [2] (D,D1) -> D from D
           if D has RMATCAT(D2,D3,D1,D4,D5) and D1 has RING and D4 has
           DIRPCAT(D3,D1) and D5 has DIRPCAT(D2,D1) and D1 has FIELD

  [3] (D1,D1) -> D from D if D has QFCAT D1 and D1 has INTDOM
  [4] (D,D1) -> D from D
           if D has MATCAT(D1,D2,D3) and D1 has RING and D2 has FLAGG
           D1 and D3 has FLAGG D1 and D1 has FIELD
  [5] (D,D1) -> D from D
           if D has LIECAT D1 and D1 has COMRING and D1 has FIELD
  [6] (D,D) -> D from D if D has GROUP
  [7] (SparseMultivariatePolynomial(D2,Kernel D),
           SparseMultivariatePolynomial(D2,Kernel D)) -> D
           from D if D2 has INTDOM and D2 has ORDSET and D has FS D2

  [8] (Float,Integer) -> Float from Float
  [9] (D,D) -> D from D if D has FIELD
```

```
[10] (D,D) -> D from D
         if D = EQ D1 and D1 has FIELD and D1 has TYPE or D = EQ D1
         and D1 has GROUP and D1 has TYPE
[11] (DoubleFloat,Integer) -> DoubleFloat from DoubleFloat
[12] (D,D1) -> D from D
         if D has AMR(D1,D2) and D1 has RING and D2 has OAMON and D1
         has FIELD
```

it eventually promotes PI to FRAC(POLY(INT)) and POLY(INT) to FRAC(POLY(INT)) and finds the match:

```
(FRAC(POLY(INT))) / (FRAC(POLY(INT)))
```

We can ask the intepreter to show us this operation (again, the output is truncated for brevity):

```
)show FRAC(POLY(INT))
Fraction Polynomial Integer is a domain constructor.
Abbreviation for Fraction is FRAC
This constructor is exposed in this frame.
Issue )edit src/algebra/FRAC.spad to see algebra source code for FRAC

------------------------------ Operations --------------------------------

?*? : (Fraction Integer,%) -> %       ?*? : (Integer,%) -> %
?*? : (PositiveInteger,%) -> %        ?*? : (%,Fraction Integer) -> %
?*? : (%,%) -> %                      ?**? : (%,Integer) -> %
?**? : (%,PositiveInteger) -> %       ?+? : (%,%) -> %
?-? : (%,%) -> %                      -? : % -> %
?/? : (%,%) -> %                      ?<? : (%,%) -> Boolean

[1]  signature:   (FRAC POLY INT,FRAC POLY INT) -> FRAC POLY INT
     implemented: slot $$$ from FRAC POLY INT
```

At this point the interpreter has succeeded in finding a type for the expression and eventually returns the result badged with the appropriate type:

```
          1
   (1)  -------
        a x + b
                                          Type: Fraction Polynomial Integer
```

0.6.2 A simple integral

Now we will show an integration with successive levels of expansion of explanation. We will use the expression above:

```
(1) -> f:=1/(a*x+b)

          1
   (1)  -------
        a x + b
                                          Type: Fraction Polynomial Integer
(2) -> integrate(f,x)
```

```
      log(a x + b)
(2)   ------------
            a
                                        Type: Union(Expression Integer,...)
```

0.6.3 A simple integral, expansion 1 interpreter

```
(2) -> integrate(f,x)
```

Here we assume the previous discussion of modemap handling for the expression f and we only look at the modemap handling for the integrate function. We are looking for a modemap of the form:

```
integrate(FRAC(POLY(INT)),VARIABLE x)
```

So first we look in the domains of the arguments, that is, in Fraction Polynomial Integer, and Variable. Neither one succeeds:

```
Function Selection for integrate
     Arguments: (FRAC POLY INT,VARIABLE x)
  -> no appropriate integrate found in Fraction Polynomial Integer
  -> no appropriate integrate found in Variable x
  -> no appropriate integrate found in Fraction Polynomial Integer
  -> no appropriate integrate found in Variable x

Modemaps from Associated Packages
  no modemaps
```

Next we look at the general modemaps to find one that might work:

```
Remaining General Modemaps
  [1] (D,D1) -> D from D
           if D1 = SYMBOL and D has UTSCAT D2 and D2 has RING and D2
           has ACFS INT and D2 has PRIMCAT and D2 has TRANFUN and D2
           has ALGEBRA FRAC INT or D1 = SYMBOL and D has UTSCAT D2 and
           D2 has RING and D2 has variables: D2 -> List D1 and D2 has
           integrate: (D2,D1) -> D2 and D2 has ALGEBRA FRAC INT
  [2] (D,D1) -> D from D
           if D1 = SYMBOL and D has UPXSCAT D2 and D2 has RING and D2
           has ACFS INT and D2 has PRIMCAT and D2 has TRANFUN and D2
           has ALGEBRA FRAC INT or D1 = SYMBOL and D has UPXSCAT D2
           and D2 has RING and D2 has variables: D2 -> List D1 and D2
           has integrate: (D2,D1) -> D2 and D2 has ALGEBRA FRAC INT

  [3] (D,D1) -> D from D
           if D1 = SYMBOL and D has ULSCAT D2 and D2 has RING and D2
           has ACFS INT and D2 has PRIMCAT and D2 has TRANFUN and D2
           has ALGEBRA FRAC INT or D1 = SYMBOL and D has ULSCAT D2 and
           D2 has RING and D2 has variables: D2 -> List D1 and D2 has
           integrate: (D2,D1) -> D2 and D2 has ALGEBRA FRAC INT
  [4] (Polynomial D2,Symbol) -> Polynomial D2 from Polynomial D2
           if D2 has ALGEBRA FRAC INT and D2 has RING
  [5] (D,D1) -> D from D
```

```
          if D has MTSCAT(D2,D1) and D2 has RING and D1 has ORDSET
          and D2 has ALGEBRA FRAC INT
   [6] (Fraction Polynomial D4,Symbol) -> Union(Expression D4,List
          Expression D4)
          from IntegrationResultRFToFunction D4
          if D4 has CHARZ and D4 has Join(GcdDomain,RetractableTo
          Integer,OrderedSet,LinearlyExplicitRingOver Integer)
   [7] (Expression Float,List Segment OrderedCompletion Float) ->
          Result
          from AnnaNumericalIntegrationPackage
   [8] (Expression Float,Segment OrderedCompletion Float) -> Result
          from AnnaNumericalIntegrationPackage
   [9] (GeneralUnivariatePowerSeries(D2,D3,D4),Variable D3) ->
          GeneralUnivariatePowerSeries(D2,D3,D4)
          from GeneralUnivariatePowerSeries(D2,D3,D4)
          if D3: SYMBOL and D2 has ALGEBRA FRAC INT and D2 has RING
          and D4: D2
   [10] (D2,Symbol) -> Union(D2,List D2) from FunctionSpaceIntegration(
          D4,D2)
          if D4 has Join(EuclideanDomain,OrderedSet,
          CharacteristicZero,RetractableTo Integer,
          LinearlyExplicitRingOver Integer) and D2 has Join(
          TranscendentalFunctionCategory,PrimitiveFunctionCategory,
          AlgebraicallyClosedFunctionSpace D4)
   [11] (Fraction Polynomial D4,SegmentBinding OrderedCompletion
          Fraction Polynomial D4) -> Union(f1: OrderedCompletion Expression
          D4,f2: List OrderedCompletion Expression D4,fail: failed,
          pole: potentialPole)
          from RationalFunctionDefiniteIntegration D4
          if D4 has Join(EuclideanDomain,OrderedSet,
          CharacteristicZero,RetractableTo Integer,
          LinearlyExplicitRingOver Integer)
   [12] (Fraction Polynomial D4,SegmentBinding OrderedCompletion
          Expression D4) -> Union(f1: OrderedCompletion Expression D4,f2:
          List OrderedCompletion Expression D4,fail: failed,pole:
          potentialPole)
          from RationalFunctionDefiniteIntegration D4
          if D4 has Join(EuclideanDomain,OrderedSet,
          CharacteristicZero,RetractableTo Integer,
          LinearlyExplicitRingOver Integer)
   [13] (D2,SegmentBinding OrderedCompletion D2) -> Union(f1:
          OrderedCompletion D2,f2: List OrderedCompletion D2,fail: failed,
          pole: potentialPole)
          from ElementaryFunctionDefiniteIntegration(D4,D2)
          if D2 has Join(TranscendentalFunctionCategory,
          PrimitiveFunctionCategory,AlgebraicallyClosedFunctionSpace
          D4) and D4 has Join(EuclideanDomain,OrderedSet,
          CharacteristicZero,RetractableTo Integer,
          LinearlyExplicitRingOver Integer)
```

Modemap [6] wins because we can construct the first argument by matching

```
Fraction Polynomial Integer
```

to

```
  Fraction Polynomial D4
```

so we can infer that D4 == Integer

```
   [6] (Fraction Polynomial D4,Symbol) -> Union(Expression D4,List
            Expression D4)
            from IntegrationResultRFToFunction D4
            if D4 has CHARZ and D4 has Join(GcdDomain,RetractableTo
            Integer,OrderedSet,LinearlyExplicitRingOver Integer)
```

Given that match we have two requirements on Integer, both of which we can check with the interpreter:

```
INT has CHARZ

   (3)  true
                                                        Type: Boolean
(4) -> INT has Join(GcdDomain,RetractableTo Integer,OrderedSet,_
                    LinearlyExplicitRingOver Integer)

   (4)  true
                                                        Type: Boolean
```

So we have a match

```
 [1]  signature:   (FRAC POLY INT,SYMBOL) -> Union(EXPR INT,LIST EXPR INT)
      implemented: slot (Union (Expression (Integer))
                        (List (Expression (Integer))))
                        (Fraction (Polynomial (Integer)))(Symbol)
                   from IRRF2F INT
 [2]  signature:   (EXPR INT,SYMBOL) -> Union(EXPR INT,LIST EXPR INT)
      implemented: slot (Union (Expression (Integer))
                        (List (Expression (Integer))))
                        (Expression (Integer))(Symbol)
                   from FSINT(INT,EXPR INT)
```

Now we invoke

```
    integrate(FRAC(POLY(INT)),SYMBOL) -> Union(EXPR INT,LIST EXPR INT)
      from IRRF2F(INT)

    integrate(1/(a*x+b),x)
```

can print the result:

```
        log(a x + b)
   (2)  ------------
              a
                                          Type: Union(Expression Integer,...)
```

0.6.4 A simple integral, expansion 2 integrate

Now that we know how the interpreter has matched the input and called the function we need to follow the first level call into the function.

Axiom provides a trace tool that will allow us to walk into the function invocation and watch what happens. We will follow this same invocation path many times, each time we will descend another layer, repeating the information as we do.

For now, we look at the domain IRRF2F from irexpand.spad. The categorical definition of this domain reads (we remove parts of the definition for brevity):

```
IntegrationResultRFToFunction(R): Exports == Implementation where
  R: Join(GcdDomain, RetractableTo Integer, OrderedSet,
          LinearlyExplicitRingOver Integer)

  RF  ==> Fraction Polynomial R
  F   ==> Expression R
  IR  ==> IntegrationResult RF
  OF  ==> OutputForm

  Exports ==> with
    expand          : IR -> List F
       ++ expand(i) returns the list of possible real functions
       ++ corresponding to i.
    if R has CharacteristicZero then
      integrate       : (RF, Symbol) -> Union(F, List F)
        ++ integrate(f, x) returns the integral of \spad{f(x)dx}
        ++ where x is viewed as a real variable..

  Implementation ==> add
    import IntegrationTools(R, F)
    import TrigonometricManipulations(R, F)
    import IntegrationResultToFunction(R, F)

    toEF: IR -> IntegrationResult F

    toEF i          == map(#1::F, i)$IntegrationResultFunctions2(RF, F)
    expand i        == expand toEF i
    complexExpand i == complexExpand toEF i

    if R has CharacteristicZero then
      import RationalFunctionIntegration(R)

      if R has imaginary: () -> R then
        integrate(f, x) == complexIntegrate(f, x)
      else
        integrate(f, x) ==
          l := [mkPrim(real g, x) for g in expand internalIntegrate(f, x)]
          empty? rest l => first l
          l

@
```

We can see that this domain constructor takes one argument which, in this case, is Integer. We've already determined that Integer has the required Joins:

```
(4) -> INT has Join(GcdDomain,RetractableTo Integer,OrderedSet,_
                    LinearlyExplicitRingOver Integer)

   (4)  true
```

```
                                                          Type: Boolean
```

and we can see that:

```
(5) -> INT has CharacteristicZero

   (5)  true
                                                                Type: Boolean
```

so we can match the signature of integrate:

```
  integrate(Fraction Polynomial Integer, Symbol) ->
       Union(Expression Integer, List Expression Integer)
```

We can trace this domain and ask to see the output in math form:

```
(6) -> )trace IRRF2F )math

   Packages traced:
      IntegrationResultRFToFunction Integer
   Parameterized constructors traced:
      IRRF2F
```

and now, when we do the integration, we see the output of the trace:

```
integrate(1/(a*x+b),x)
1<enter IntegrationResultRFToFunction.integrate,32 :
          1
 arg1= -------
       a x + b
 arg2= x
 1<enter IntegrationResultRFToFunction.expand,18 :
        1     a x + b
  arg1= - log(-------)
        a        a
 1>exit  IntegrationResultRFToFunction.expand,18 :
       a x + b
   log(-------)
         a
  [------------]
        a
1>exit  IntegrationResultRFToFunction.integrate,32 :
 log(a x + b)
 ------------
       a

        log(a x + b)
   (6)  ------------
              a
                                          Type: Union(Expression Integer,...)
```

From this we learn that the arguments to integrate are exactly the arguments we supplied and we know the exact types of the arguments because they have to match the signature of the function:

```
1<enter IntegrationResultRFToFunction.integrate,32 :
  integrate(, Symbol) ->
          1
 arg1= -------    <== Fraction Polynomial Integer
```

```
       a x + b
 arg2= x          <== Symbol
```

and returns the result

```
1>exit  IntegrationResultRFToFunction.integrate,32 :
 log(a x + b)
 ------------     <== Union(Expression Integer, List Expression Integer)
       a
```

0.6.5 A simple integral, expansion 2 internalIntegrate

If we look at the function definition for integrate:

```
integrate(f, x) ==
  l := [mkPrim(real g, x) for g in expand internalIntegrate(f, x)]
  empty? rest l => first l
  l
```

we can see that there is a call to the function

```
internalIntegrate(f, x)
```

and we can compute the types of the arguments since they are exactly the types of the integrate function itself:

```
internalIntegrate(Fraction Polynomial Integer, Symbol)
```

and since the return value will be fed to the expand function we can look at the signature of expand:

```
expand: IntegrationResult Fraction Polynomial Integer ->
           List Expression Integer
```

and we can get the full signature for internalIntegrate:

```
internalIntegrate(Fraction Polynomial Integer, Symbol) ->
  IntegrationResult Fraction Polynomial Integer
```

This comes from the domain

```
RationalFunctionIntegration(F): Exports == Implementation where
  F: Join(IntegralDomain, RetractableTo Integer, CharacteristicZero)
```

where F is Integer.

```
SE  ==> Symbol
P   ==> Polynomial F
Q   ==> Fraction P
UP  ==> SparseUnivariatePolynomial Q
QF  ==> Fraction UP
LGQ ==> List Record(coeff:Q, logand:Q)
UQ  ==> Union(Record(ratpart:Q, coeff:Q), "failed")
ULQ ==> Union(Record(mainpart:Q, limitedlogs:LGQ), "failed")

Exports ==> with
  internalIntegrate: (Q, SE) -> IntegrationResult Q
     ++ internalIntegrate(f, x) returns g such that \spad{dg/dx = f}.
Implementation ==> add
  import RationalIntegration(Q, UP)
```

```
    import IntegrationResultFunctions2(QF, Q)
    import PolynomialCategoryQuotientFunctions(IndexedExponents SE,
                                                        SE, F, P, Q)
    internalIntegrate(f, x) ==
      map(multivariate(#1, x), integrate univariate(f, x))
```

If we look the signature for internalIntegrate and expand it we see:

```
    internalIntegrate: (Q, SE) -> IntegrationResult Q

    internalIntegrate: ( Fraction Polynomial Integer, Symbol) ->
      IntegrationResult Fraction Polynomial Integer
```

which is exactly what we need. When we look at the function we see:

```
    internalIntegrate(f, x) ==
      map(multivariate(#1, x), integrate univariate(f, x))
```

We can watch the function call by tracing INTRF:

```
(7) -> )trace INTRF )math

   Packages traced:
      IntegrationResultRFToFunction Integer,
            RationalFunctionIntegration Integer
   Parameterized constructors traced:
      IRRF2F, INTRF
```

and we see:

```
(7) -> integrate(1/(a*x+b),x)
1<enter IntegrationResultRFToFunction.integrate,32 :
          1
 arg1= -------
       a x + b
 arg2= x
 1<enter RationalFunctionIntegration.internalIntegrate,25 :
           1
  arg1= -------
        a x + b
  arg2= x
 1>exit  RationalFunctionIntegration.internalIntegrate,25 :
  1     a x + b
  - log(-------)
  a        a
 1<enter IntegrationResultRFToFunction.expand,18 :
        1     a x + b
  arg1= - log(-------)
        a        a
 1>exit  IntegrationResultRFToFunction.expand,18 :
       a x + b
   log(-------)
         a
  [------------]
         a
1>exit  IntegrationResultRFToFunction.integrate,32 :
```

```
 log(a x + b)
 ------------
       a

        log(a x + b)
   (7)  ------------
              a
                                          Type: Union(Expression Integer,...)
```

Now we see that internalIntegrate was called with the arguments

```
 1<enter RationalFunctionIntegration.internalIntegrate,25 :
           1
  arg1= -------      <== Fraction Polynomial Integer
        a x + b
  arg2= x            <== Symbol
```

and returned the values:

```
 1>exit  RationalFunctionIntegration.internalIntegrate,25 :
 1     a x + b
 - log(-------)     <== IntegrationResult Fraction Polynomial Integer
 a        a
```

0.6.6 A simple integral, expansion 3 univariate

But the internalIntegrate function does its work by calling yet other functions, the deepest of which is univariate:

```
internalIntegrate(f, x) ==
  map(multivariate(#1, x), integrate univariate(f, x))
```

Since univariate uses the arguments to the internalIntegrate function which has the signature:

```
internalIntegrate: ( Fraction Polynomial Integer, Symbol) ->
```

we can determine that we need a univariate function with the signature:

```
univariate: ( Fraction Polynomial Integer, Symbol) ->
```

This function is found in PolynomialCategoryQuotientFunctions, POLYCATQ which has the form:

```
PolynomialCategoryQuotientFunctions(E, V, R, P, F):
 Exports == Implementation where
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: Ring
  P: PolynomialCategory(R, E, V)
  F: Field with
    coerce: P -> %
    numer : % -> P
    denom : % -> P

  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP

  Exports ==> with
    variables   : F -> List V
```

```
      ++ variables(f) returns the list of variables appearing
      ++ in the numerator or the denominator of f.
    mainVariable: F -> Union(V, "failed")
      ++ mainVariable(f) returns the highest variable appearing
      ++ in the numerator or the denominator of f, "failed" if
      ++ f has no variables.
    univariate  : (F, V) -> RF
      ++ univariate(f, v) returns f viewed as a univariate
      ++ rational function in v.
  Implementation ==> add
    P2UP: (P, V) -> UP

    univariate(f, x) == P2UP(numer f, x) / P2UP(denom f, x)

    P2UP(p, x) ==
      map(#1::F,
          univariate(p, x))$SparseUnivariatePolynomialFunctions2(P, F)
```

So we are calling the function:

```
 univariate: ( Fraction Polynomial Integer, Symbol) ->
  Fraction SparseUnivariatePolynomial Field with
   coerce: PolynomialCategory(Ring, OrderedAbelianMonoidSup, OrderedSet) -> %
   numer: % -> PolynomialCategory(Ring, OrderedAbelianMonoidSup, OrderedSet)
   denom: % -> PolynomialCategory(Ring, OrderedAbelianMonoidSup, OrderedSet)
```

which we can see by tracing that domain:

```
(8) -> )trace POLYCATQ )math

   Packages traced:
      IntegrationResultRFToFunction Integer,
            RationalFunctionIntegration Integer,
            PolynomialCategoryQuotientFunctions(IndexedExponents
            Kernel Expression Integer,Kernel Expression Integer,
            Integer,SparseMultivariatePolynomial(Integer,Kernel
            Expression Integer),Expression Integer),
            PolynomialCategoryQuotientFunctions(IndexedExponents
            Symbol,Symbol,Integer,Polynomial Integer,Fraction
            Polynomial Integer)
   Parameterized constructors traced:
      IRRF2F, INTRF, POLYCATQ
```

which gives the input:

```
  1<enter PolynomialCategoryQuotientFunctions.univariate,16 :
            1
   arg1= -------  <== Fraction Polynomial Integer
         a x + b
   arg2= x        <== Symbol
```

and the output

```
  1>exit  PolynomialCategoryQuotientFunctions.univariate,16 :
     1
     -
     a
   -----   <==   Fraction SparseUnivariatePolynomial Field with
```

```
      b                  coerce: P -> %
  ? + -                  numer:  % -> P
      a                  denom:  % -> P
```

It should be clear that univariate divided the numerator and denominator by the leading coefficient of the polynomial in the denominator. It also replaced "x" with the variable "?".

0.6.7 A simple integral, expansion 4 integrate

When univariate returns, the results are fed to another integrate, this time from RationalIntegration (INTRAT). This domain looks like:

```
RationalIntegration(F, UP): Exports == Implementation where
  F : Join(Field, CharacteristicZero, RetractableTo Integer)
  UP: UnivariatePolynomialCategory F

  RF  ==> Fraction UP
  IR  ==> IntegrationResult RF
  LLG ==> List Record(coeff:RF, logand:RF)
  URF ==> Union(Record(ratpart:RF, coeff:RF), "failed")
  U   ==> Union(Record(mainpart:RF, limitedlogs:LLG), "failed")
  OF  ==> OutputForm

  Exports ==> with
    integrate  : RF -> IR
      ++ integrate(f) returns g such that \spad{g' = f}.

  Implementation ==> add
    import TranscendentalIntegration(F, UP)

    integrate f ==
      rec := monomialIntegrate(f, differentiate)
      integrate(rec.polypart)::RF::IR + rec.ir
```

This domain was constructed and "brought into scope" in RationalFunctionIntegration(F) with the statement

```
import RationalIntegration(Fraction Polynomial Integer,
        SparseUnivariatePolynomial Fraction Polynomial Integer)
```

and the function has the signature

```
integrate:
  Fraction SparseUnivariatePolynomial Fraction Polynomial Integer ->
   IntegrationResult Fraction
                     Fraction Polynomial Integer
```

```
 1<enter RationalIntegration.integrate,32 :
          1
          -
          a
  arg1= -----  <== Fraction SparseUnivariatePolynomial
            b                Fraction Polynomial Integer
        ? + -
            a
 1>exit  RationalIntegration.integrate,32 :
```

```
   1         b
   - log(? + -) <== IntegrationResult Fraction SparseUnivariatePolynomial
   a         a                 Fraction Polynomial Integer
```

0.6.8 A simple integral, expansion 5 monomialIntegrate

The integrate function is defined as:

```
 integrate f ==
   print(outputForm("tpdhere INTRAT 1")@OF)$OF
   rec := monomialIntegrate(f, differentiate)
   integrate(rec.polypart)::RF::IR + rec.ir
```

Notice that while "f" is an argument to integrate, the "differentiate" function is a free variable. The Axiom compiler will look at all of the symbols "in scope" to find its meaning. This code does an import:

```
  import TranscendentalIntegration(Fraction Polynomial Integer,
          SparseUnivariatePolynomial Fraction Polynomial Integer)
```

which exports monomialIntegrate

```
TranscendentalIntegration(F, UP): Exports == Implementation where
  F  : Field
  UP : UnivariatePolynomialCategory F

  RF  ==> Fraction UP
  FF  ==> Record(ratpart:F,  coeff:F)
  UF  ==> Union(FF,  "failed")
  IR  ==> IntegrationResult RF
  REC ==> Record(ir:IR, specpart:RF, polypart:UP)

  Exports ==> with
    monomialIntegrate : (RF, UP -> UP) -> REC
      ++ monomialIntegrate(f, ') returns \spad{[ir, s, p]} such that
      ++ \spad{f = ir' + s + p} and all the squarefree factors of the
      ++ denominator of s are special w.r.t the derivation '.

 Implementation ==> add
  import SubResultantPackage(UP, UP2)
  import MonomialExtensionTools(F, UP)
  import TranscendentalHermiteIntegration(F, UP)
  import CommuteUnivariatePolynomialCategory(F, UP, UP2)

  monomialIntegrate(f, derivation) ==
    zero? f => [0, 0, 0]
    r := HermiteIntegrate(f, derivation)
    zero?(inum := numer(r.logpart)) =>
       [r.answer::IR, r.specpart, r.polypart]
    iden  := denom(r.logpart)
    x := monomial(1, 1)$UP
    resultvec := subresultantVector(UP2UP2 inum -
                    (x::UP2) * UP2UP2 derivation iden, UP2UP2 iden)
    respoly := primitivePart leadingCoefficient resultvec 0
    rec := splitSquarefree(respoly, kappa(#1, derivation))
    logs:List(LOG) := [
```

```
        [1, UP2UPR(term.factor),
         UP22UPR swap primitivePart(resultvec(term.exponent),term.factor)]
               for term in factors(rec.special)]
    dlog :=
         ((derivation x) = 1) => r.logpart
         differentiate(mkAnswer(0, logs, empty()),
                       differentiate(#1, derivation))
    (u := retractIfCan(p := r.logpart - dlog)@Union(UP, "failed")) case UP =>
      [mkAnswer(r.answer, logs, empty), r.specpart, r.polypart + u::UP]
    [mkAnswer(r.answer, logs, [[p, dummy]]), r.specpart, r.polypart]
```

which expands into the type signature:

```
  monomialIntegrate:
  (Fraction SparseUnivariatePolynomial Fraction Polynomial Integer,
     SparseUnivariatePolynomial Fraction Polynomial Integer ->
          SparseUnivariatePolynomial Fraction Polynomial Integer) ->
    Record(ir: IntegrationResult Fraction
                    SparseUnivariatePolynomial Fraction Polynomial Integer,
           specpart: Fraction
                    SparseUnivariatePolynomial Fraction Polynomial Integer,
           polypart: SparseUnivariatePolynomial Fraction Polynomial Integer)
      ++ monomialIntegrate(f, ') returns \spad{[ir, s, p]} such that
      ++ \spad{f = ir' + s + p} and all the squarefree factors of the
      ++ denominator of s are special w.r.t the derivation '.
```

we can watch this happen:

```
)trace INTTR )math

   Function traced: UnivariatePolynomialCategory
   Packages traced:
      IntegrationResultRFToFunction Integer,
            RationalFunctionIntegration Integer,
            RationalIntegration(Fraction Polynomial Integer,
            SparseUnivariatePolynomial Fraction Polynomial
            Integer), PolynomialCategoryQuotientFunctions(
            IndexedExponents Kernel Expression Integer,Kernel
            Expression Integer,Integer,
            SparseMultivariatePolynomial(Integer,Kernel
            Expression Integer),Expression Integer),
            PolynomialCategoryQuotientFunctions(IndexedExponents
            Symbol,Symbol,Integer,Polynomial Integer,Fraction
            Polynomial Integer), TranscendentalIntegration(
            Fraction Polynomial Integer,
            SparseUnivariatePolynomial Fraction Polynomial
            Integer)
   Parameterized constructors traced:
      IRRF2F, INTRF, INTRAT, POLYCATQ, INTTR
```

and we can watch the monomialIntegrate function call

```
(34) -> integrate(1/(a*x+b),x)
1<enter IntegrationResultRFToFunction.integrate,32 :
          1
 arg1= -------
```

```
      a x + b
 arg2= x
   "tpdhere IRRF2F 1"
 1<enter RationalFunctionIntegration.internalIntegrate,25 :
          1
  arg1= -------
       a x + b
  arg2= x
  1<enter PolynomialCategoryQuotientFunctions.univariate,16 :
           1
   arg1= -------
        a x + b
   arg2= x
  1>exit  PolynomialCategoryQuotientFunctions.univariate,16 :
     1
     -
     a
   -----
       b
   ? + -
       a
  1<enter RationalIntegration.integrate,32 :
           1
           -
           a
   arg1= -----  <== Fraction SparseUnivariatePolynomial
             b                 Fraction Polynomial Integer
         ? + -
             a
   1<enter TranscendentalIntegration.monomialIntegrate,81 :
            1
            -
            a
    arg1= -----  <== Fraction SparseUnivariatePolynomial
              b                 Fraction Polynomial Integer
          ? + -
              a
    arg2= theMap(UPOLYC-;differentiate;2S;37,873)
   1>exit  TranscendentalIntegration.monomialIntegrate,81 :
         1         b
    [ir= - log(? + -),specpart= 0,polypart= 0]
         a         a
  1>exit  RationalIntegration.integrate,32 :
   1         b
   - log(? + -)
   a         a
 1>exit  RationalFunctionIntegration.internalIntegrate,25 :
  1     a x + b
  - log(-------)
  a        a
1>exit  IntegrationResultRFToFunction.integrate,32 :
 log(a x + b)
 ------------
       a
```

```
         log(a x + b)
   (34)  ------------
               a
                                        Type: Union(Expression Integer,...)
(35) ->
```

0.6.9 A simple integral, expansion 6 HermiteIntegrate

Since "f" is not zero we invoke HermiteIntegrate from the domain TranscendentalHermiteIntegration which looks like:

```
TranscendentalHermiteIntegration(F, UP): Exports == Implementation where
  F  : Field
  UP : UnivariatePolynomialCategory F

  N   ==> NonNegativeInteger
  RF  ==> Fraction UP
  REC ==> Record(answer:RF, lognum:UP, logden:UP)
  HER ==> Record(answer:RF, logpart:RF, specpart:RF, polypart:UP)

  Exports ==> with
    HermiteIntegrate: (RF, UP -> UP) -> HER
         ++ HermiteIntegrate(f, D) returns \spad{[g, h, s, p]}
         ++ such that \spad{f = Dg + h + s + p},
         ++ h has a squarefree denominator normal w.r.t. D,
         ++ and all the squarefree factors of the denominator of s are
         ++ special w.r.t. D. Furthermore, h and s have no polynomial parts.
         ++ D is the derivation to use on \spadtype{UP}.

  Implementation ==> add
    import MonomialExtensionTools(F, UP)

    HermiteIntegrate(f, derivation) ==
      rec := decompose(f, derivation)
      hi  := normalHermiteIntegrate(rec.normal, derivation)
      qr  := divide(hi.lognum, hi.logden)
      [hi.answer, qr.remainder / hi.logden, rec.special, qr.quotient + rec.poly]
```

The function has the same input signature as monomialIntegrate but a different return signature.

```
  HermiteIntegrate:
  (Fraction SparseUnivariatePolynomial Fraction Polynomial Integer,
     SparseUnivariatePolynomial Fraction Polynomial Integer ->
          SparseUnivariatePolynomial Fraction Polynomial Integer) ->
   Record(answer:Fraction SparseUnivariatePolynomial
                                              Fraction Polynomial Integer,
          logpart:Fraction SparseUnivariatePolynomial
                                              Fraction Polynomial Integer,
          specpart:Fraction SparseUnivariatePolynomial
                                              Fraction Polynomial Integer,
          polypart:SparseUnivariatePolynomial Fraction Polynomial Integer)
```

so we trace this domain

```
(37) -> )trace INTHERTR )math

   Function traced: UnivariatePolynomialCategory
   Packages traced:
      IntegrationResultRFToFunction Integer,
            RationalFunctionIntegration Integer,
            RationalIntegration(Fraction Polynomial Integer,
            SparseUnivariatePolynomial Fraction Polynomial
            Integer), PolynomialCategoryQuotientFunctions(
            IndexedExponents Kernel Expression Integer,Kernel
            Expression Integer,Integer,
            SparseMultivariatePolynomial(Integer,Kernel
            Expression Integer),Expression Integer),
            PolynomialCategoryQuotientFunctions(IndexedExponents
            Symbol,Symbol,Integer,Polynomial Integer,Fraction
            Polynomial Integer), TranscendentalIntegration(
            Fraction Polynomial Integer,
            SparseUnivariatePolynomial Fraction Polynomial
            Integer), TranscendentalHermiteIntegration(Fraction
            Polynomial Integer,SparseUnivariatePolynomial
            Fraction Polynomial Integer)
   Parameterized constructors traced:
      IRRF2F, INTRF, INTRAT, POLYCATQ, INTTR, INTHERTR
```

and now we see

```
(38) -> integrate(1/(a*x+b),x)
1<enter IntegrationResultRFToFunction.integrate,32 :
         1
 arg1= -------
       a x + b
 arg2= x
   "tpdhere IRRF2F 1"
 1<enter RationalFunctionIntegration.internalIntegrate,25 :
          1
  arg1= -------
        a x + b
  arg2= x
  1<enter RationalIntegration.integrate,32 :
           1
           -
           a
   arg1= -----
             b
         ? + -
             a
   "tpdhere INTRAT 1"
   1<enter TranscendentalIntegration.monomialIntegrate,81 :
            1
            -
            a
    arg1= -----
              b
          ? + -
              a
```

```
    arg2= theMap(UPOLYC-;differentiate;2S;37,873)
    1<enter TranscendentalHermiteIntegration.HermiteIntegrate,18 :
             1
             -
             a
     arg1= -----
               b
           ? + -
               a
     arg2= theMap(UPOLYC-;differentiate;2S;37,873)
    1>exit  TranscendentalHermiteIntegration.HermiteIntegrate,18 :
                            1
                            -
                            a
      [answer= 0,logpart= -----,specpart= 0,polypart= 0]
                              b
                         ? + -
                              a
   1>exit  TranscendentalIntegration.monomialIntegrate,81 :
         1         b
    [ir= - log(? + -),specpart= 0,polypart= 0]
         a         a
   "tpdhere UPOLYC 1"
  1>exit  RationalIntegration.integrate,32 :
   1         b
   - log(? + -)
   a         a
 1>exit  RationalFunctionIntegration.internalIntegrate,25 :
  1     a x + b
  - log(-------)
  a        a
 1<enter IntegrationResultRFToFunction.expand,18 :
        1     a x + b
  arg1= - log(-------)
        a        a
 1>exit  IntegrationResultRFToFunction.expand,18 :
       a x + b
   log(-------)
          a
  [------------]
          a
1>exit  IntegrationResultRFToFunction.integrate,32 :
 log(a x + b)
 ------------
       a

         log(a x + b)
   (38)  ------------
               a
                                          Type: Union(Expression Integer,...)
```

so HermiteIntegrate did nothing to the input. Next we call normalHermiteIntegrate which is a local function

0.7 Tools

0.7.1 svn

SVN is a source control system on all platforms. Axiom 'silver' is maintained in an SVN archive on sourceforge. This can be pulled from:

```
svn co https://axiom.svn.sf.net/svnroot/axiom/trunk/axiom axiom
```

0.7.2 git

Git is a unix-based source code control system. Axiom 'silver' is maintained in a git archive. This can be pulled from:

```
git-clone ssh://git@axiom-developer.org/home/git/silver
```

the password for the userid git is linus.

0.7.3 cvs

This assumes that you have set up ssh on the Savannah site. CVS does not use a password. You have to log onto the Savannah site and set up a public key. This requires you to:

- set up a local public key: ssh-keygen -b 1024 -t rsa1
- open a browser
- nagivate to the savannah page that has your personal keys
- open .ssh/identity.pub
- cut .ssh/identity.pub
- paste it into your personal key list on savannah
- go have a beer (the page takes an hour or two to update)

Once you have a working key you can do the cvs login. If it prompts you for a password then the key is not working. If it prompts you to "Enter the passphrase for RSA key" then cvs login will work.

I maintain a directory where I work (call this WORK)

```
/home/axiomgnu/new
```

and a directory for CVS (call this GOLD)

```
/axiom
```

When I want to export a set of changes I do the following steps:

0) MAKE SURE THE /.ssh/config FILE IS CORRECT:

```
    (you should only need to do this once.
     you need to change the User= field)

Host *.gnu.org
  Protocol=1
  Compression=yes
  CompressionLevel=3
  User=axiom
```

```
  StrictHostKeyChecking=no
  PreferredAuthentications=publickey,password
  NumberOfPasswordPrompts=2
```

1) MAKE SURE THE SHELL VARIABLES ARE OK:

```
    (normally set in .bashrc)

export CVS_RSH=ssh
export CVSROOT=:pserver:axiom@subversions.gnu.org:/cvsroot/axiom
                        ^^^^^
                        change this to your id
```

2) MAKE SURE YOU'RE LOGGED IN:

```
 (I keep a session open all the time but it doesn't seem to care
  if you login again. i'm not sure what login does, actually)

cvs login
```

3) GET A FRESH COPY FOR THE FIRST TIME OR AT ANY TIME:

```
 (you only need to do this the first time but you can erase
  your whole axiom subtree and refresh it again doing this.

  note that i work as root so i can update /.  Most rational
  people are smarter than me and work as a regular user so
  you have to change the instructions for cd. But you knew that)

cd /
cvs co axiom
```

4) MAKE SURE THAT GOLD, MY LOCAL CVS COPY, IS UP TO DATE:

```
 (I maintain an exact copy of the CVS repository and only make
  changes to it when i want to export the changes. that way I
  won't export my working tree by accident. my working tree is
  normally badly broken.

  The update command makes sure that you have all of the changes
  other people might have made and checked in. you have to merge
  your changes so you don't step on other people's work.
  So be sure to run update BEFORE you copy files to GOLD)

cd /axiom
cvs update
```

5) COPY CHANGED FILES FROM WORK TO THE GOLD TREE:

```
 (This is an example for updating the *.daase files.
  You basically are changing your GOLD tree to reflect the
  way you want CVS to look once you check in all of the files.)

cd /home/axiomgnu/new
cp src/share/algebra/*.daase /axiom/src/share/algebra
```

6) IF A FILE IS NEW (e.g. src/interp/foo.lisp.pamphlet) THEN:

```
 (If you create a file you need to "put it under CVS control"
  CVS only cares about files you explicitly add or delete.
  If you make a new file and copy it to GOLD you need to do this.
```

```
  Don't do the "cvs add" in your WORK directory. The cvs add
  command updates the files in the CVS directory and you won't
  have them in your WORK directory.

  Notice that you do the "cvs add" in the directory where the
  file was added (hence, the cd commands).

cd /axiom/src/interp
cvs add -m"some pithy comment" foo.lisp.pamphlet
cd /axiom
```

7) IF A FILE IS DELETED (e.g. src/interp/foo.lisp.pamphlet) THEN:

```
 (you have to delete the file from the GOLD directory BEFORE you
  do a "cvs remove". The "cvs remove" will update the files in
  the CVS directory

  Notice that you do the "cvs remove" in the directory where the
  file was deleted (hence, the cd commands).

cd /axiom/src/interp
rm foo.lisp.pamphlet
cvs remove foo.lisp.pamphlet
cd /axiom
```

8) IF A DIRECTORY IS NEW (e.g. foodir) THEN:

```
 (this will put "foodir" under CVS control. It will also create
  foodir/CVS as a directory with a bunch of control files in the
  foodir/CVS directory. Don't mess with the control files.

 (there are a bunch of special rules about directories.
  empty directories are not downloaded by update.)

 (NOTE: THERE IS NO WAY TO DELETE A DIRECTORY)

cd /axiom/src
mkdir foodir
cvs add -m "pithy comment" foodir
cd /axiom
```

9) EDIT CHANGELOG:

```
  changelog is already under CVS control so it will get uploaded
  when you do the checkin.)

cd /axiom
emacs -nw changelog
 (add a date, initials, and pithy comment, save it, and exit)
```

10) CHECK IN THE CHANGES

```
 (This will actually change the savannah CVS repository.

  The "cvs ci" command will recurse thru all of the lower
  subdirectories and look for changed files. It will change
  the host versions of those files to agree with your copy.
```

```
  If somebody else has changed a file while you were busy
  developing code then the checkin MAY complain (if it can't
  merge the changes)

cd /axiom
cvs ci -m"pithy comment"
```

Congrats. You've now done your first change to the production image. Please be very careful as this is a world readable copy. We don't want to ship nonsense. Test everything. Even trivial changes before you upload.

0.8 Common Lisps

0.8.1 GCL

Axiom was ported to run under AKCL which was a common lisp developed by Bill Schelter. He started with KCL (Kyoto Common Lisp) and, since he lived and worked in Austin, Texas, named his version AKCL (Austin-Kyoto Common Lisp). Bill worked under contract to the Scratchpad group at IBM Research. I was the primary developer for system internals so Bill and I worked closely together on a lot of issues. After Axiom was sold to NAG Bill continued to develop AKCL and it eventually became GCL (Gnu Common Lisp).

In order to port Axiom to run on GCL we need to do several things. First, we need to apply a few patches. These patches enlarge the default stack size, remove the startup banner, link with Axiom's socket library, and rename collectfn.

The issue with the stack size is probably bogus. At one point the system was running out of stack space but the problem was due to a recursive expansion of a macro and no amount of stack space would be sufficient. This patch remains at the moment but should probably be removed and tested.

The startup banner is an issue because we plan to run under various frontend programs like Texmacs and the Magnus ZLC. We need to just output a single prompt.

Axiom has a socket library because at the time it was developed under AKCL there was no socket code in Lisp. There is still not a standard common lisp socket library but I believe all common lisps have a way to manipulate sockets. This code should be rewritten in lisp and `#+` for each common lisp.

The collectfn file is a major optimization under GCL. When collectfn is loaded and the lisp compiler is run then collectfn will output a .fn file. The second time the compiler is invoked the .fn file is consulted to determine the actual types of arguments used. Function calling is highly optimized using this type information so that fast function calling occurs. Axiom should be built one time to create the int/*/*.fn files. It should then be rebuilt using the cached .fn files. I will automate this process into the Makefiles in the future.

GCL implementation will have a major porting problem to brand new platforms. The compiler strategy is to output C code, compile it using GCC, and dynamically link the machine code to the running image. This requires deep knowledge of the symbol tables used by the native linker for each system. In general this is a hard problem that requires a lot of expertise. Bill Schelter and I spent a lot of time and effort making this work for each port. The magic knowledge is not written down anywhere and I no longer remember the details.

0.8.2 CCL

When Axiom was sold to NAG it was ported to CCL (Codemist Common Lisp) which is not, strictly speaking, a common lisp implementation. It contains just enough common lisp to support Axiom and, as I'm a great believer in simple code, it only needed a small subset of a full common lisp.

CCL can be considered the best way to get Axiom running on a new architecture as the porting issues are minimal.

CCL is a byte-interpreter implementation and has both the positive and negative aspects of that design choice. The positive aspect is that porting the code to run on new architectures is very simple. Once the CCL byte-code interpreter is running then Axiom is running. The saved-system image is pure byte-codes and is completely system independent.

The negative aspects are that it is slow and the garbage collector appears broken. Compiling the Axiom library files on an file-by-file basis takes about 1 hour on GCL and about 12 hours on CCL. Compiling all of the Axiom library files in the same image (as opposed to starting a new image per file) still takes about 1 hour on GCL. It never finishes in CCL. Indeed it stops doing useful work after about the 40th file (out of several hundred).

When Axiom became open source I moved the system back to GCL because I could not understand how to build a CCL system. I plan to revisit this in the future and document the process so others can follow it as well as build Makefiles to automate it.

0.8.3 CMU CL

CMU CL grew out of the Carnegie-Mellon University SPICE project. That project studied the issues involved in building an optimizing compiler for common lisp. Axiom, back when it was Scratchpad at IBM, ran on CMU CL. Indeed, a lot of the lisp-level optimizations are due to use of the CMU CL compiler and the disassemble function.

0.8.4 Franz Lisp

Axiom, as Scratchpad, ran on Franz Lisp.

0.8.5 Lucid Common Lisp

Axiom, as Scratchpad, ran on Lucid Common Lisp.

0.8.6 Symbolics Common Lisp

Axiom, as Scratchpad, ran on Symbolics Common Lisp.

0.8.7 Golden Common Lisp

Axiom, as Scratchpad, ran on Golden Common Lisp. This was a PC version of Common Lisp which appears to have died.

0.8.8 VM/LISP 370

Axiom, as Scratchpad, ran on VM/Lisp 370. This was an IBM version of lisp and was not a common lisp. The .daase random access file format is an artifact of running on this lisp.

0.8.9 Maclisp

Axiom, as Scratchpad, ran on Maclisp. This was an early MIT version of lisp and is not common lisp. Many of the funny function names that have slightly different semantics than their common lisp counterparts still exist in the system as macros due to this lisp.

0.9 Changing GCL versions

Axiom lives on GNU Common Lisp. Axiom adds C code to the lisp image. Axiom caches versions to ensure that nothing breaks. Changing GCL versions has introduced subtle bugs at various times. The steps necessary to introduce a new version are

1. Add the latest GCL sources to Axiom
2. Update the patches to the new version
3. create diff -Naur patches to the gcl sources
4. update lsp/Makefile.pamphlet to apply the patch at build
5. add a new chunk to lsp/Makefile.pamphlet to build gcl-2.6.10
6. Change the GCLVERSION to point at the new sources
7. change the Makefile to match Makefile.pamphlet

We assume in the following that `$AXHOME` is the home directory and that Axiom lives in the silver subdirectory.

In more detail these steps are:

1. Add the lateset GCL sources to Axiom
   (a) Download the latest GCL from gnu.org
   For these instructions assume the file is
   ```
   gcl-Version_2_6_10.tar.gz
   ```
   (b) move the tar file into /tmp
   We are going to make changes to the distribution via patches
   (c) untar the file
   ```
   tar -zxf gcl-Version_2_6_10.tar.gz
   ```
   (d) cd to the untarred directory
   ```
   cd gcl-Version_2_6_10
   ```
   (e) rename the gcl directory
   Camm follows a convention that the top level directory in the tar file is called gcl. Since we maintain several past versions we need to rename this and re-tar it
   ```
   mv gcl gcl-2.6.10
   ```
   (f) rename gcl to use our naming convention

```
tar -zcf gcl-2.6.10.tgz gcl-2.6.10
```

(g) We move the original, renamed, retarred file to the zip directory

```
mv gcl-2.6.10.tgz $AXHOME/silver/zips
```

(h) We have to make sure to include the new file in the git commit

```
cd $AXHOME/silver
```

(i) Tell git we care about the file

```
git add zips/gcl-2.6.10.tgz
```

2. Update the patches to the new version
   (a) find the previous patches

```
ls $AXHOME/silver/zips/gcl-2.6.8pre7*patch
```

   (b) for each patch do (Step 3 ; Step 4)
3. create diff -Naur patches to the gcl sources
   (a) assume we are looking at gcl-2.6.8pre7.h.linux.defs.patch
   The name tells us what file to patch. From the above we can see that when Axiom builds GCL it will

```
cd lsp/gcl-2.6.8pre7
```

   because GCL is built by the lsp/Makefile.pamphlet. That Makefile will do a

```
cd h
patch <gcl-2.6.8pre7.h.linux.defs.patch
```

   which will apply the patch .... So we need to make a patch
   (b) move to the subdirectory containing the file

```
cd /tmp/gcl-Version_2_6_10/gcl-2.6.10/h
```

   (c) edit the 'linux.defs' file to create the proper patch
   (d) save the changed file as linux.defs.tpd
   (e) in a shell, create a diff -Naur patch by

```
diff -Naur linux.defs linux.defs.tpd >gcl-2.6.10.h.linux.defs.patch
```

   (f) move it to the zips directory

```
mv gcl-2.6.10.h.linux.defs.patch $AXHOME/silver/zips
cd $AXHOME/silver
git add zips/gcl-2.6.10.h.linux.defs.patch
```

4. update lsp/Makefile.pamphlet to apply the patch at build
   (a) edit lsp/Makefile.pamphlet
   (b) search for chunk `gcl-2.6.8pre7.h.linux.defs.patch`
   (c) copy the chunk and name the new chunk `gcl-2.6.10.h.linux.defs.patch`
5. add a new chunk to lsp/Makefile.pamphlet to build gcl-2.6.10
   (a) find the subsection "The GCL-2.6.8pre7 stanza"
   (b) make a copy named "The GCL-2.6.10 stanza"
   (c) add the new patches
   (d) tell git we care

```
cd $AXHOME/silver
git add lsp/Makefile.pamphlet
```

6. Change the GCLVERSION to point at the new sources
   (a) emacs `$AXHOME/silver/Makefile.pamphlet`
   (b) search for `#GCLVERSION`, a Makefile comment line
   (c) the last line is uncommented. Assume it reads GCLVERSION=gcl-2.6.8pre7
   gcl-2.6.8pre7 was is the name of the current version we are replacing. We will use this name in the next step
   (d) put a # in front of the GCLVERSION variable to comment it out
   We maintain the list of old, working patches. We also remember the names of the prior GCLVERSIONS in case we have to back up
   (e) Add a new line reading:

   ```
   GCLVERSION=gcl-2.6.10
   ```

   This will cause Axiom to untar this tgz file to get the sources and apply the corresponding patches
7. change the Makefile to match Makefile.pamphlet
   - compile the tangle program

     ```
     ( cd books ; gcc -o tangle tangle.c )
     ```

   - use books/tangle to extract the new Makefile

     ```
     books/tangle Makefile.pamphlet >Makefile
     ```

0.10 Literate Programming

The Axiom source code was originally developed at IBM Research. It was sold to The Numerical Algorithms Group (NAG) and was on the market as a commercial competitor to Mathematica and Maple.

Axiom was withdrawn from the market in 2000 and released as free and open source software in 2001. When the Axiom project was started on savannah, the GNU Free Software Foundation site the source code had been rewritten into "pamphlet" files. The reasons for this are twofold.

0.10.1 Pamphlet files

When the Axiom code was released it contained few comments. That made it very difficult to understand what the code actually did. Unlike commercial software there would be no group of individuals who would work on the project for its lifetime. Thus there needed to be a way to capture the expertise and understanding underlying ongoing development.

Unlike any other piece of free and open source software Axiom will still give useful answers 30 years from now. Thus it is important, and worthwhile, to invest a large amount of effort into documenting how these answers are arrived at and why the algorithms are written the way they are.

The pamphlet file format follows Knuth's idea of literate programming. Knuth made the

observation that a program should be a work designed to be read by humans. Making the program readable by machine was a secondary consideration. Making documentation primary and code secondary was a dramatic shift for a programmer.

Knuth created a file format that combined documentation and code. He created a tool called "Web" which had two basic command, tangle and weave. The tangle command would be run against a literate document and extract the source code, the weave command would be run against the literate document and extract the TeX.

0.10.2 noweb

Knuth's Web tool was specifically designed to use Pascal code. The "tangle" operation would prettyprint the output according to the style rules of Pascal.

Axiom was written in a variety of languages, such as C and Lisp, and used tools such as Makefiles which have their own syntax. Thus Web could not be used directly.

Axiom defines a new latex environment called chunk. This chunk environment makes the pamphlet file a pure latex file. This eliminates the need for the weave operation. The tangle operation only needs to occur while manipulating code, either during system build or end user interaction. At both of these times the tangle operation can be built into the system and hidden.

To support extracting chunks from pamphlet files Axiom now has a new top level command. At the top level one can write:

```
)tangle filename
```

This will look for "filename.pamphlet" and extract the top level chunk which has the name "*".

The latest changeset introduces two related changes, gclweb and axiom.sty. Together these changes allow optional syntactic changes to pamphlets. These changes will completely eliminate the need to weave files since now a pamphlet file can be a valid latex file. Tangle is the only remaining command and it will eventually be an option on)compile, etc.

The src/interp/gclweb.lisp file introduces the ability to extract code from pamphlet files while inside Axiom. The short description is that gclweb will now automatically distinguish the type of chunk style (latex or noweb) based on the chunk name. It is a first step to a native understanding of pamphlet files. Future work involves integrating it into commands like)compile and adding commands like)tangle.

Tangle can also be called directly from lisp on a file from within Axiom:

```
)lisp (tangle "filename.pamphlet" "chunkname")
)lisp (tangle "filename.pamphlet" "chunkname" "filename.spad")
```

gclweb distinguishes the input syntax by looking at the first character of the chunkname. If it is a '<' then noweb is used, otherwise latex.

The src/doc/axiom.sty.pamphlet introduces the new chunk environment. This is a completely compatible change and has no impact on existing pamphlets. The new syntax makes pamphlet files = tex files so there is no need to use weave. The gclweb change has a compatible tangle function which can be invoked from inside Axiom.

```
\begin{chunk}{chunkname}
  your code goes here
\end{chunk}
```

One feature of the latex chunk style is that latex commands work within the chunk. To get typeset mathematics use \(and \)

```
-- This will typeset in a chunk \( x^2+\epsilon \)
-- And you can format things {\bf bold}
```

0.11 Databases

0.11.1 libcheck

The databases are built from the .kaf files in the .nrlib directories. (.kaf files are random access files).

interp.exposed is a file that names all of the CDPs (Category, Domain, and Packages) and classifies them. Only some CDPs are exposed because most are used to implement algebra and are not intended to be user level functions. Exposing all of the functions causes much ambiguity.

There is a function called libcheck (see src/interp/util.lisp.pamphlet) that will check nrlibs vs interp.exposed. This is only partially functional as I see that changes were made to the system which broke this function.

The libcheck function requires an absolute pathname to the int directory so call it thus:

```
 --> )lisp (libcheck "/axiom/int/algebra")
```

The main reason this function is broken is that the system now gets exposure information from src/algebra/exposed.lsp.pamphlet. It appears that interp.exposed.pamphlet is no longer used (although I made sure that both files have the same information). I'm going to modify libcheck to use exposed.lsp in the future and eliminate all references in the system to interp.exposed.

For the moment, however, the libcheck function is quite useful. It used to be run during system build because I frequently ran into database problems and this function would alert me to that fact. I'll add it back into the Makefile once I elide interp.exposed.

0.11.2 asq

Axiom has several databases which contain information about domains, categories, and packages. The databases are in a compressed format and are organized as random-access files using numeric index values so it is hard to get at the stored information. However, there is a command-line query function called asq (pronounced ask) that knows the format of the files and can be used for stand-alone queries. For instance, if you know the abbreviation for a domain but want to know what source file generated that domain you can say:

```
asq -so FOOBAR
```

and it will tell you the name of the algebra source file that defines FOOBAR.

0.12 Axiom internal representations

```
PRIMITIVE REPRESENTATIONS OF AXIOM OBJECTS
```

```
There are several primitive representations in axiom. These are:

boolean
 this is represented as a lisp boolean

integer
 this is represented as a lisp integer

small integer
 this is represented as a lisp integer

small float
 this is represented as a lisp float

list
 this is represented as a lisp list

vector
 this is represented as a lisp vector

record
 there are 3 cases:
  records of 1 element are a pair (element . nil)
  records of 2 element are a pair (element1 . element2)
  records of 3 or more are a vectors #<a b c...>

mapping
 mappings are a spadcall objects. they are represented as a pair
  (lispfn . env)
 where the env is usually a type object. A spadcall rips this
 pair open and applies the lispfn to its args with env as the
 last arg.

union
 there are 2 cases
  if the object can be determined by a lisp predicate
  (eg integer) then the union is just the object (eg 3)
  itself since we can use lisp to decide which branch of
  the union the object belongs to. that is, 3 is of the
  integer branch in union(list,integer)

  if the object cannot be determined then the object is
  wrapped into a pair where the car of the pair is the
  union branch name and the cdr of the pair is the object.
  that is, given union(a:SUP,b:POLY(INT)) x might be (a . x)

  note: if no tags are given in the union the system uses
  consecutive integers, thus union(SUP,POLY(INT)) will give
  a pair of (1 . x) or (2 . x) depending on the type of x

other types are built up of compositions of these primitive
types. a sparse univariate polynomial (SUP) over the integers
```

```
  x**2+1

is represented as

Term := Record(k:NonNegativeInteger,c:R)
Rep  := List Term

that is, the representation is a list of terms where each term
is a record whose first field is a nonnegative integer (the
exponent) and the second field is a member of the coefficient
ring. since this is a record of length 2 it is represented as
a pair. thus, the internal form of this polynomial is:

 ((2 . 1) (0 . 1))

a more complex object (recursively defined) is POLY(INT). given

 x**2+1

as a POLY(INT) we look at its representation and see:

D := SparseUnivariatePolynomial($)
VPoly := Record(v:VarSet,ts:D)
Rep := Union(R,VPoly)

so first we find that we are a member of the second form of the
union and since this is an untagged union the system uses 2 as
the tag. thus the first level of internal representation is:

 ( 2 . <a VPoly object> )

next we need to define the VPoly object. VPolys are records of
length 2 so we know they are represented by a pair. the car of
the pair is a VarSet. the cdr is a D which is a
SparseUnivariatePolynomial. Thus we consider this to be a poly
in x (at the top level) and we get:

 ( 2 . ( x . <an SUP>))

the SUP is over the SparseMultivariatePolynomials (SMP) so the
representation is recursive. Since an SUP is represented as a
list of

 (non-negative int . coefficient)

one per term and we have 2 terms we know the next level of
structure is:

 ( 2 . ( x . (( 2 . <an SMP> ) ( 0 . <an SMP> ))))

the SMP is just the integers so it fits into the first branch
of the union and each SMP looks like:
```

```
 ( uniontag . value )
```

in this case, being the first branch we get

```
 ( 2 . ( x . (( 2 . ( 1 . 1 )) ( 0 . ( 1 . 1 )))))
```

as the internal representation of

```
 x**2 + 1
```

what could be easier?

0.13 Spad to internal function calling

0.13.1 getdatabse output

```
GETDATABASE('Permutation, 'OPERATIONALIST)$Lisp
```

generates the output

```
((|$unique|)
 (~= (((Boolean) $ $) () T ELT))
 (sort (((List $) (List $)) 76 T ELT))
 (sign (((Integer) $) 59 T ELT))
 (sample (($) () T CONST))
 (recip (((Union $ "failed") $) () T ELT))
 (order (((NonNegativeInteger) $) 57 T ELT))
 (orbit (((Set #1) $ #1) 48 T ELT))
 (one? (((Boolean) $) () T ELT))
 (odd? (((Boolean) $) 62 T ELT))
 (numberOfCycles (((NonNegativeInteger) $) 60 T ELT))
 (movedPoints (((Set #1) $) 41 T ELT))
 (min (($ $ $) () (OR (has #1 (Finite)) (has #1 (OrderedSet))) ELT))
 (max (($ $ $) () (OR (has #1 (Finite)) (has #1 (OrderedSet))) ELT))
 (listRepresentation
   (((Record (: preimage (List #1)) (: image (List #1))) $) 35 T ELT))
 (latex (((String) $) () T ELT))
 (inv (($ $) 92 T ELT))
 (hash (((SingleInteger) $) () T ELT))
 (fixedPoints (((Set #1) $) 98 (has #1 (Finite)) ELT))
 (even? (((Boolean) $) 58 T ELT))
 (eval ((#1 $ #1) 46 T ELT))
 (elt ((#1 $ #1) 93 T ELT))
 (degree (((NonNegativeInteger) $) 43 T ELT))
 (cycles (($ (List (List #1))) 84 T ELT))
 (cyclePartition (((Partition) $) 52 T ELT))
 (cycle (($ (List #1)) 21 T ELT))
 (conjugate (($ $ $) () T ELT))
 (commutator (($ $ $) () T ELT))
 (coercePreimagesImages (($ (List (List #1))) 38 T ELT))
 (coerceListOfPairs (($ (List (List #1))) 87 T ELT))
 (coerceImages (($ (List #1)) 95 T ELT))
 (coerce  (((OutputForm) $) 83 T ELT)  (($ (List (List #1))) 65 T ELT)
```

```
      (($ (List #1)) 66 T ELT))
    (^  (($ $ (PositiveInteger)) () T ELT)
      (($ $ (NonNegativeInteger)) () T ELT)  (($ $ (Integer)) () T ELT))
    (One (($) 16 T CONST))
    (>= (((Boolean) $ $) () (OR (has #1 (Finite)) (has #1 (OrderedSet))) ELT))
    (> (((Boolean) $ $) () (OR (has #1 (Finite)) (has #1 (OrderedSet))) ELT))
    (= (((Boolean) $ $) 44 T ELT))
    (<= (((Boolean) $ $) () (OR (has #1 (Finite)) (has #1 (OrderedSet))) ELT))
    (< (((Boolean) $ $) 64 T ELT))
    (/ (($ $ $) () T ELT))
    (**  (($ $ (PositiveInteger)) () T ELT)
      (($ $ (NonNegativeInteger)) () T ELT)  (($ $ (Integer)) () T ELT))
    (* (($ $ $) 22 T ELT)))
```

Sometimes in a getdatabase expression you will see:

```
(~= (((Boolean) $ $) () T ELT))
---------------------^^
```

and in other places there is a number

```
      (sign (((Integer) $) 59 T ELT))
--------------------------^^
```

In general, when a large number appears it is a byte index into the compress.daase file.

Axiom would not fit on a laptop. We needed smaller databases. The solution to the problem was to scan the datatases for common substrings, write the substring to compress.daase, and replace the substring by the byte offset.

When reading the database these numbers would be replaced by the substring from compress.daase using random access seeks based on the byte offset.

See book volume 5 for an explanation of the database file formats.

HOWEVER, in this case, the number has a different meaning which I will talk about below.

In summary, this shows what the following incantation means:

```
  (sign (((Integer) $) () (has #1 (OrderedIntegralDomain))))

INTEGER inherits sign from OINTDOM (OrderedIntegralDomain)
OINTDOM inherits sign from ORDRING (OrderedRing)
ORDRING implements sign
   since ORDRING is a category, the actual code lives in
   ORDRING-.nrlib/code.lsp
```

The code for sign in ORDRING-.nrlib/code.lsp has the signature:

```
   (DEFUN |ORDRING-;sign;SI;3| (|x| $) ....)
```

We can "decode" the meaning of the function name as

- **ORDRING-** the implementing file
- **sign** the function name
- **SI** returns SingleInteger (an old domain name)
- **3** the third function in the file (unique, to distinguish multiple functions with the same name)

It takes 2 arguments,

- `|x|` which should be a SingleInteger
- `$` which is the current domain (ORDRING-)

So it looks like I have the following structure

```
(NAME ((TARGETTYPE SOURCETYPE) ?1 CONDITION ?2))
```

but we are looking up 'sign' in INTEGER so there is a condition on sign

```
Integer has OrderedRing  ==> true
```

so that explains the condition field.

Here we show how Axiom finds the function implementation, looks up the function "in the domain", and calls it.

```
(sign (((Integer) $) 59 T ELT))
```

Now you've asked for 'sign' from domain Permutation

```
(sign (((Integer) $) 59 T ELT))
```

The implementation for 'sign' is in PERM.nrlib/code.lsp. It reads:

```
(defun |PERM;sign;$I;17| (|p| $)
 (cond
  ((spadcall |p| (qrefelt $ 58)) 1)
  ('t -1)))
```

which you would read as

```
if (calling function in position 58 of myself) is true
  then return 1
  else return -1
```

How does Axiom find the function? It is in the infovec which is the "information vector" containing information about the domain.

First we must make sure that PERM has the necessary domain information loaded (the 'infovec').

```
-> [1,2,3]::PERM(INT)
```

Now, back to the 'sign' function. You saw this:

```
(sign (((Integer) $) 59 T ELT))  (sample (($) () T CONST))
```

which is asking you to look up element 59 from the domain (`$`)

Note that `$` *actually* means the infovec. So we are asking:

```
(elt (elt (getf (symbol-plist '|Permutation|) '|infovec|) 0) 59)
```

which results in:

```
|PERM;sign;$I;17|
```

so we "looked up" the function sign in the domain PERM.

Explaining in more detail, from the inside out by walking the runtime data structures we see

```
(symbol-plist '|Permutation|)
```

returns the property list on the symbol **Permutation** which is where Axiom caches domain information. Almost everything of interest about a domain resides on the property list, shown here in all its glory.

```
(LOADED "/research/test/mnt/ubuntu/algebra/PERM.o"
 SYSTEM:DEBUG (#:G1567 #:G1568)
 |infovec| (
  #(NIL NIL NIL NIL NIL NIL
    (|local| |#1|)        (QUOTE |Rep|)           (|Boolean|)
    (0 . <)               (|PositiveInteger|)     (6 . |lookup|)
    (|Integer|)           (|List| 6)              (11 . |maxIndex|)
    (16 . |elt|)
    (CONS IDENTITY (FUNCALL (|dispatchFunction|   |PERM;One;$;29|) $))
    (|NonNegativeInteger|) (22 . |last|)          (28 . |first|)
    (34 . |concat|)        |PERM;cycle;L$;26|     |PERM;*;3$;28|
    (40 . =)               (46 . =)               (52 . |elt|)
    (58 . |list|)          (63 . |position|)      (69 . |delete|)
    (|Mapping| 8 13 13)    (|List| 13)            (75 . |sort|)
    (81 . |copy|)          (86 . |member?|)
    (|Record| (|:| |preimage| 13) (|:| |image| 13))
    |PERM;listRepresentation;$R;9|                (92 . |elt|)
    (98 . ~=)              |PERM;coercePreimagesImages;L$;10|
    (|Set| 6)              (104 . |construct|)    |PERM;movedPoints;$S;11|
    (109 . |#|)            |PERM;degree;$Nni;12|  |PERM;=;2$B;13|
    (114 . |brace|)        |PERM;eval;$2S;31|     (119 . |insert!|)
    |PERM;orbit;$SS;14|    (|List| 12)            (|Partition|)
    (125 . |partition|)    |PERM;cyclePartition;$P;15|
    (130 . |convert|)      (135 . |removeDuplicates|)
    (|List| $)             (140 . |lcm|)          |PERM;order;$Nni;16|
    |PERM;even?;$B;18|     |PERM;sign;$I;17|      |PERM;numberOfCycles;$Nni;33|
    (145 . |even?|)        |PERM;odd?;$B;19|      (150 . |maxIndex|)
    |PERM;<;2$B;20|        |PERM;coerce;L$;21|    |PERM;coerce;L$;22|
    (|Record| (|:| |cycl| 30) (|:| |permut| $$))
    (|List| 67)            (155 . |cons|)         (|Mapping| 8 67 67)
    (161 . |sort|)         (|List| $$)            (167 . |nil|)
    (171 . |cons|)         (177 . |reverse|)      |PERM;sort;2L;23|
    (|OutputForm|)         (182 . |coerce|)       (187 . |blankSeparate|)
    (192 . |paren|)        (197 . |outputForm|)   (202 . |hconcat|)
    |PERM;coerce;$Of;24|   |PERM;cycles;L$;25|    (207 . |second|)
    (212 . =)              |PERM;coerceListOfPairs;L$;27|
    (|Vector| 6)           (218 . |construct|)    (223 . |elt|)
    (229 . |new|)          |PERM;inv;2$;30|       |PERM;elt;$2S;32|
    (235 . |coerce|)       (240 . |coerceImages|) (245 . |index|)
    (250 . |complement|)   (255 . |fixedPoints|)  (260 . |conjugate|)
    (265 . +)              (|Union| $ (QUOTE "failed"))
    (|SingleInteger|)      (|String|))
  #(~= 271                     |sort| 277               |sign| 282
    |sample| 287               |recip| 291             |order| 296
    |orbit| 301                |one?| 307              |odd?| 312
    |numberOfCycles| 317       |movedPoints| 322       |min| 327
    |max| 333                  |listRepresentation| 339 |latex| 344
    |inv| 349                  |hash| 354              |fixedPoints| 359
    |even?| 364                |eval| 369              |elt| 375
    |degree| 381               |cycles| 386           |cyclePartition| 391
    |cycle| 396                |conjugate| 401        |commutator| 407
    |coercePreimagesImages| 413 |coerceListOfPairs| 418 |coerceImages| 423
    |coerce| 428               ^ 443                  |One| 461
     >= 465                    > 471                  = 477
```

```
   <= 483                    < 489                    / 495
   ** 501                    * 519)
 ((|unitsKnown| . 0))
 (#(0 0 0 0 3 0 0 0)
  #(NIL
    |Group&|
    |Monoid&|
    |SemiGroup&|
    |OrderedSet&|
    |SetCategory&|
    |BasicType&|
    NIL)
  #((|PermutationCategory| 6)
    (|Group|)
    (|Monoid|)
    (|SemiGroup|)
    (|OrderedSet|)
    (|SetCategory|)
    (|BasicType|)
    (|CoercibleTo| 77))
 .
  #( 2  6  8  0  0  9  1  6 10  0 11  1 13 12  0 14  2 13  6  0 12
    15  2 13  0  0 17 18  2 13  0  0 17 19  2 13  0  0  0 20  2  6
     8  0  0 23  2 13  8  0  0 24  2  7 13  0 12 25  1 13  0  6 26
     2 13 12  6  0 27  2 13  0  0 12 28  2 30  0 29  0 31  1 13  0
     0 32  2 13  8  6  0 33  2 30 13  0 12 36  2  6  8  0  0 37  1
    39  0 13 40  1 39 17  0 42  1 39  0 13 45  2 39  0  6  0 47  1
    50  0 49 51  1 50 49  0 53  1 49  0  0 54  1 12  0 55 56  1 12
     8  0 61  1 30 12  0 63  2 68  0 67  0 69  2 68  0 70  0 71  0
    72  0 73  2 72  0  2  0 74  1 72  0  0 75  1  6 77  0 78  1 77
     0 55 79  1 77  0  0 80  1 77  0 12 81  1 77  0 55 82  1 13  6
     0 85  2 39  8  0  0 86  1 88  0 13 89  2 88  6  0 12 90  2  7
     0 17 13 91  1  6  0 12 94  1  0  0 13 95  1  6  0 10 96  1 39
     0  0 97  1  0 39  0 98  1 50  0  0 99  2 50  0  0  0 100 2  0
     8  0  0  1  1  0 55 55 76  1  0 12  0 59  0  0  0  1  1  0 101
     0  1  1  0 17  0 57  2  0 39  0  6 48  1  0  8  0  1  1  0  8
     0 62  1  0 17  0 60  1  0 39  0 41  2  3  0  0  0  1  2  3  0
     0  0  1  1  0 34  0 35  1  0 103 0  1  1  0  0  0 92  1  0 102
     0  1  1  1 39  0 98  1  0  8  0 58  2  0  6  0  6 46  2  0  6
     0  6 93  1  0 17  0 43  1  0  0 30 84  1  0 50  0 52  1  0  0
    13 21  2  0  0  0  0  1  2  0  0  0  0  1  1  0  0 30 38  1  0
     0 30 87  1  0  0 13 95  1  0  0 13 66  1  0  0 30 65  1  0 77
     0 83  2  0  0  0 12  1  2  0  0  0 17  1  2  0  0  0 10  1  0
     0  0 16  2  3  8  0  0  1  2  3  8  0  0  1  2  0  8  0  0 44
     2  3  8  0  0  1  2  0  8  0  0 64  2  0  0  0  0  1  2  0  0
     0 12  1  2  0  0  0 17  1  2  0  0  0 10  1  2  0  0  0  0 22))
 |lookupComplete|)
   PNAME "Permutation"
   DATABASE
    #S(DATABASE
       ABBREVIATION PERM
       ANCESTORS NIL
       CONSTRUCTOR NIL
       CONSTRUCTORCATEGORY 2444459
```

```
CONSTRUCTORKIND |domain|
CONSTRUCTORMODEMAP
 ((((|Permutation| |#1|)
   (|Join|
    (|PermutationCategory| |#1|)
    (CATEGORY |domain|
     (SIGNATURE |listRepresentation|
       ((|Record| (|:| |preimage| #) (|:| |image| #)) $))
     (SIGNATURE |coercePreimagesImages| ($ (|List| (|List| |#1|))))
     (SIGNATURE |coerce| ($ (|List| (|List| |#1|))))
     (SIGNATURE |coerce| ($ (|List| |#1|)))
     (SIGNATURE |coerceListOfPairs| ($ (|List| (|List| |#1|))))
     (SIGNATURE |degree| ((|NonNegativeInteger|) $))
     (SIGNATURE |movedPoints| ((|Set| |#1|) $))
     (SIGNATURE |cyclePartition| ((|Partition|) $))
     (SIGNATURE |order| ((|NonNegativeInteger|) $))
     (SIGNATURE |numberOfCycles| ((|NonNegativeInteger|) $))
     (SIGNATURE |sign| ((|Integer|) $))
     (SIGNATURE |even?| ((|Boolean|) $))
     (SIGNATURE |odd?| ((|Boolean|) $))
     (SIGNATURE |sort| ((|List| $) (|List| $)))
     (IF (|has| |#1| (|Finite|))
       (SIGNATURE |fixedPoints| ((|Set| |#1|) $)) |noBranch|)
     (IF (|has| |#1| (|IntegerNumberSystem|))
       (SIGNATURE |coerceImages| ($ (|List| |#1|)))
       (IF (|has| |#1| (|Finite|))
         (SIGNATURE |coerceImages| ($ #)) |noBranch|))))
   (|SetCategory|))
  (T |Permutation|))
COSIG (NIL T)
DEFAULTDOMAIN NIL
MODEMAPS 2443154
NILADIC NIL
OBJECT "PERM"
OPERATIONALIST
 ((|$unique|)
  (~= ((((|Boolean|) $ $) NIL . #0=(T . #1=(ELT))))
  (|sort| ((((|List| $) (|List| $)) 76 . #0#))
  (|sign| ((((|Integer|) $) 59 . #0#))
  (|sample| (($) NIL T CONST))
  (|recip| ((((|Union| $ "failed") $) NIL . #0#))
  (|order| ((((|NonNegativeInteger|) $) 57 . #0#))
  (|orbit| ((((|Set| |#1|) $ |#1|) 48 . #0#))
  (|one?| ((((|Boolean|) $) NIL . #0#))
  (|odd?| ((((|Boolean|) $) 62 . #0#))
  (|numberOfCycles| ((((|NonNegativeInteger|) $) 60 . #0#))
  (|movedPoints| ((((|Set| |#1|) $) 41 . #0#))
  (|min| (($ $ $) NIL
    (OR (|has| |#1| (|Finite|)) (|has| |#1| (|OrderedSet|))) . #1#))
  (|max| (($ $ $) NIL
    (OR (|has| |#1| (|Finite|)) (|has| |#1| (|OrderedSet|))) . #1#))
  (|listRepresentation|
    ((((|Record| (|:| |preimage| (|List| |#1|))
                 (|:| |image| (|List| |#1|))) $) 35 . #0#))
```

```
(|latex| (((|String|) $) NIL . #0#))
(|inv| (($ $) 92 . #0#))
(|hash| (((|SingleInteger|) $) NIL . #0#))
(|fixedPoints| (((|Set| |#1|) $) 98 (|has| |#1| (|Finite|)) . #1#))
(|even?| (((|Boolean|) $) 58 . #0#))
(|eval| ((|#1| $ |#1|) 46 . #0#))
(|elt| ((|#1| $ |#1|) 93 . #0#))
(|degree| (((|NonNegativeInteger|) $) 43 . #0#))
(|cycles| (($ (|List| (|List| |#1|))) 84 . #0#))
(|cyclePartition| (((|Partition|) $) 52 . #0#))
(|cycle| (($ (|List| |#1|)) 21 . #0#))
(|conjugate| (($ $ $) NIL . #0#))
(|commutator| (($ $ $) NIL . #0#))
(|coercePreimagesImages| (($ (|List| (|List| |#1|))) 38 . #0#))
(|coerceListOfPairs| (($ (|List| (|List| |#1|))) 87 . #0#))
(|coerceImages| (($ (|List| |#1|)) 95 . #0#))
(|coerce|
  (((|OutputForm|) $) 83 . #0#)
  (($ (|List| (|List| |#1|))) 65 . #0#)
  (($ (|List| |#1|)) 66 . #0#))
(^ (($ $ (|PositiveInteger|)) NIL . #0#)
   (($ $ (|NonNegativeInteger|)) NIL . #0#)
   (($ $ (|Integer|)) NIL . #0#))
(|One| (($) 16 T CONST))
(>= (((|Boolean|) $ $) NIL
   (OR (|has| |#1| (|Finite|)) (|has| |#1| (|OrderedSet|))) . #1#))
(> (((|Boolean|) $ $) NIL
   (OR (|has| |#1| (|Finite|)) (|has| |#1| (|OrderedSet|))) . #1#))
(= (((|Boolean|) $ $) 44 . #0#))
(<= (((|Boolean|) $ $) NIL
   (OR (|has| |#1| (|Finite|)) (|has| |#1| (|OrderedSet|))) . #1#))
(< (((|Boolean|) $ $) 64 . #0#))
(/ (($ $ $) NIL . #0#))
(** (($ $ (|PositiveInteger|)) NIL . #0#)
    (($ $ (|NonNegativeInteger|)) NIL . #0#)
    (($ $ (|Integer|)) NIL . #0#))
(* (($ $ $) 22 . #0#)))
DOCUMENTATION 1609893
CONSTRUCTORFORM 1609883
ATTRIBUTES 1614391
PREDICATES 1614406
SOURCEFILE "bookvol10.3.pamphlet"
PARENTS NIL
USERS NIL
DEPENDENTS NIL
SPARE NIL))
```

There are many things on the property list which looks like

```
(symbol1 thing1 symbol2 thing2 ... symboln thingn)
```

In the PERM case we see

```
(LOADED "/research/silver/mnt/algebra/PERM.o"
 |infovec| (#<vector> #<vector>...)
 ....)
```

We can get the `|infovec|` off the property list with the call

(getf (symbol-plist '—Permutation—) '—infovec—)

is a request to search the property list for the symbol —infovec— and return the value, which is the domain "information vector".

You can see this vector if you look in PERM.nrlib/code.lsp. At the bottom of that file you'll see:

```
(SETF (GET (QUOTE |Permutation|) (QUOTE |infovec|) ....)
```

which uses SETF to put the infovec on the property list of PERM. This information vector contains information for function lookup. This vector gets created when we "instantiate" PERM.

The **infovec** is a list with the structure

```
(#<vector 08ea516c>
 #<vector 08ea5150>
 ((|unitsKnown| . 0))
 (#<vector 08ea50fc>
  #<vector 08ea5134>
  #<vector 08ea5118> . #<vector 08ea50e0>)
|lookupComplete|)
```

So, now that we have the infovec, back to the game...

```
 (elt (getf (symbol-plist '|Permutation|) '|infovec|) 0)
```

This gets the 0th element out of the infovec list which is a vector of the name of every function Permutation implements. We look up function names in this list, in particular, 59:

```
(elt (elt (getf (symbol-plist '|Permutation|) '|infovec|) 0) 59)
```

looks into this vector of names at the 59th element which returns

```
|PERM;sign;$I;17|
```

The SPAD form of this function reads:

```
sign(p) ==
  even? p => 1
  -1
```

The lisp form (see PERM.nrlib/code.lsp) reads:

```
(defun |PERM;sign;$I;17| (|p| $)
 (cond
  ((spadcall |p| (qrefelt $ 58)) 1)
  ('t -1)))
```

We call the `|PERM;sign;$I;17|` which takes 2 arguments

The first of which is the permutation and the second is the infovec for the PERM domain.

The `(qrefelt $ 58)` uses the above dance to look up a function in the infovec at the 58th position... which returns

```
|PERM;even?;$B;18|
```

The spadcall calls `|PERM;even?;$B;18|` with the value of `|p|`.

If we look in the domain Permutation for the implementation of even?

```
even?(p) == even?(#(p.1) - numberOfCycles p)
```

which in PERM.nrlib/code.lsp we see

```
(defun |PERM;even?;$b;18|
 (spadcall
  (- (length (spadcall |p| 1 (qrefelt $ 25)))
     (spadcall |p| (qrefelt $ 60)))
  (qrefelt % 61)))
```

where

```
(qrefelt $ 25) ==> (52 . |elt|)
(qrefelt $ 60) ==> |PERM;numberOfCycles;$Nni;33|
(qrefelt $ 61) ==> (145. |even?|)
```

So, to summarize, the small magic numbers you see in the results are indexes into the infovec, which is where Axiom stores things it needs to look up at runtime, usually function references.

If there is () rather than a number than there is no need to do a function lookup.

Axiom execution is an alternating series of function lookups in the infovec followed by a call of that function which results in a function lookup in the infovec followed by a call of that function which results in .....

spadcall is a wrapper macro which takes the arguments and a function to call. qrefelt does the infovec lookup.

0.14 axiom command

The axiom command will eventually be a shell script. At the moment it is just a copy of the interpsys image. However the whole Axiom system consists of several processes and the axiom command starts these processes. The shell script will transparently replace the axiom executable image which will be renamed to spadsys.

0.15 help command documentation

Axiom supports a)help command that takes a single argument. This argument is interpreted as the name of a flat ascii file which should live in $AXIOM/doc/src/spadhelp.

0.15.1 help documentation for algebra

The help documentation for algebra files lives within the algebra pamphlet. The help chunk contains the name of the domain, thus:

```
\begin{chunk}{thisdomain.help}
=============================================================================
thisdomain examples
=============================================================================

   (documentation for this domain)
```

```
    examplefunction foo
     output
                  Type: thetype

  See Also:
  o )show thisdomain
  o $AXIOM/bin/src/doc/algebra/thisfile.spad.dvi

  \end{chunk}
```

The documentation starts off with the domain enclosed in two lines of equal signs. The documentation is free format. Generally the functions are indented two spaces, the output is indented 3 spaces, and the Type field has been moved toward the center of the line.

The "See Also:" section lists the domain with the "show" command and the path to the source file in dvi format.

0.15.2 Adding help documentation in Makefile

There is a section in the src/algebra/Makefile.pamphlet that reads:

```
SPADHELP=\
 ${HELP}/AssociationList.help  ${HELP}/BalancedBinaryTree.help \
```

which is essentially a list of all of the algebra help files. Each item in this list refers to a stanza that looks like:

```
${HELP}/AssociationList.help: ${BOOKS}/bookvol10.3.pamphlet
@echo 7000 create AssociationList.help from \
           ${BOOKS}/bookvol10.3.pamphlet
@${TANGLE} -R"AssociationList.help" ${BOOKS}/bookvol10.3.pamphlet \
            >${HELP}/AssociationList.help
@cp ${HELP}/AssociationList.help ${HELP}/ALIST.help
@${TANGLE} -R"AssociationList.input" ${BOOKS}/bookvol10.3.pamphlet \
            >${INPUT}/AssociationList.input
@echo "AssociationList (ALIST)" >>${HELPFILE}
```

Notice that the first line has an connection between the help file and the spad file that contains it.

The second line gives debugging output containing a unique number for console debugging purposes of failed builds.

The third line extracts the help file. Help files are part of the algebra books (bookvol10.2, bookvol10.3, and bookvol10.4). The chunkname is the same as the Category, Domain, or Package.

The fourth line copies the file with the long name of the domain to a file with the abbreviation of the domain so the user can query the domain with either form using help.

The fifth line creates a regression test file for the help file. In the algebra each help file has an associated regression test file to test all of the function calls shown in the help page. These files are copied to the intermediate directory for regression testing.

The sixth line adds a line to the HELPFILE (see the variable in the src/algebra/Makefile). This HELPFILE is concatenated onto the final help.help file in the MNT/doc/spadhelp directory. Thus, when a user types)help with no argument they see a list of domains which

contain help information.

0.15.3 Using help documentation for regression testing

The fifth line extracts an input test file for the algebra. In general each help file is used to create an input test file for regression testing.

There is a Makefile variable called REGRESS in the algebra Makefile:

```
REGRESS=\
 AssociationList.regress  BalancedBinaryTree.regress \
```

This is part of a Makefile that structure within the algebra Makefile. This Makefile gets extracted by the Makefile in the input subdirectory. Thus there is a connection between the two Makefiles (algebra and input). This algebra regression Makefile goes by the chunk name **algebra.regress**. It contains a list of regression files and a single stanza:

```
%.regress: %.input
@ echo algebra regression testing $*
@ rm -f $*.output
@ echo ')read $*.input' | ${TESTSYS}
@ echo ')lisp (regress "$*.output")' | ${TESTSYS} \
                | egrep -v '(Timestamp|Version)' | tee $*.regress
```

The input Makefile extracts **algebra.regress** and then calls make to process this file.

This keeps the regression test list in the algebra Makefile.

0.15.4 help documentation as algebra test files

0.16 debugsys

The "debugsys" executable is the "interpsys" image but it is built using the interpreted lisp code rather than using compiled lisp code. This will make it slower but may, in certain cases, give much better feedback in case of errors. If you find you need to use debugsys you're really doing deep debugging. It isn't useful for much else. It can be started by typing:

```
export AXIOM=/home/axiomgnu/new/mnt/linux
/home/axiomgnu/new/obj/linux/bin/debugpsys
```

Notice that this image lives in the "obj" subtree. It is not shipped with the "final" system image as only developers could find it useful.

0.16.1 debugging hyperdoc

Hyperdoc will sometimes exit and also kill the AXIOMsys image with no error message. One way to get around this is to replace the AXIOMsys image with the debugsys image:

1. mv $AXIOM/bin/AXIOMsys $AXIOM/bin/AXIOMsys.backup
   This keeps the failing axiomsys image around for later restoration.
2. cp obj/sys/bin/debugsys $AXIOM/bin/AXIOMsys
   This puts an interpreted version of axiom in place of the compiled form
3. axiom

Now we are running a fully interpreted form and the error messages are much more informative.

0.17 Understanding a compiled function

Suppose we stop a program at a function call to some low level lisp function, say ONEP. We can do that by entering

```
)trace ONEP )break
```

at the Axiom command prompt. Or at the lisp prompt:

```
(trace (ONEP :entry (break)))
```

Next we execute some function that will eventually call ONEP thus:

```
p := numeric %pi

Break: onep
Broken at ONEP.  Type :H for Help.
BOOT>>
```

We have stopped and entered a lisp command prompt. We can enter any lisp expression here and there are commands that begin with a ":" character. ":b" requests a backtrace of the call stack, thus:

```
BOOT>>:b
Backtrace: funcall > system:top-level > restart > /read >
      |upLET| > eval > |Pi| > |newGoGet| > |newGoGet| > ONEP
```

Here we see that the function ONEP was called by the function newGoGet. Notice that the name is surrounded by vertical bars. Vertical bars are a common lisp escape sequence used to allow non-standard characters to occur in symbol names. Common lisp is not case sensitive. Boot code is case sensitive. Thus symbol names that were written in Boot tend to have escape sequence characters around the name.

Now that we see the simple backtrace we can ask for a more complex one. The command is ":bt". It shows more detail about each level of call on the invocation history stack (ihs) including the function name, its arguments and the depth of the invocation history stack ([ihs=13]):

```
BOOT>>:bt

#0   ONEP {1=nil,} [ihs=13]
#1   newGoGet {g3629=("0" (#<vector 08b34bb4> 45 . |char|)),
       loc1=#<compiled-function |CHAR;cha...} [ihs=12]
#2   newGoGet {g3629=("%pi" (#<vector 08b34bec> 0 . |coerce|)),
       loc1=(#<vector 08b34bec> 0 . |c...} [ihs=11]
#3   Pi {g109299=nil,loc1=nil,loc2=#<hash-table 082992f4>,
       loc3=|Pi|,loc4=15,loc5=#<vecto...} [ihs=10]
#4   EVAL {loc0=nil,loc1=nil,loc2=nil,
       loc3=#<compiled-function |Pi|>} [ihs=9]
#5   upLET {t=(#<vector 08b34d04> #<vector 08b34ce8>
       (#<vector 08b34ccc> (#<vector 08b34c08...} [ihs=8]
#6   /READ {loc0=#p"/home/axiomgnu/new/src/input/algbrbf.input",
       loc1=nil,loc2=nil,loc3=nil,...} [ihs=7]
#7   RESTART {loc0=((|read|
```

```
      |/home/axiomgnu/new/src/input/algbrbf.input|)),
      loc1=|/home/axiomg...} [ihs=6]
#8   TOP-LEVEL {loc0=nil,loc1=0,loc2=0,loc3=nil,loc4=nil,
      loc5=nil,loc6=nil,loc7=nil,loc8=nil,lo...} [ihs=5]
#9   FUNCALL {loc0=#<compiled-function system:top-level>} [ihs=4]
BOOT>>:bl
>> (LAMBDA-BLOCK ONEP (&REST X) ...)():
X        : (1)
NIL
```

We can ask to see the local variables that are used at the current level of the invocation history stack. The command is ":bl" thus:

```
BOOT>>:bl
>> (LAMBDA-BLOCK ONEP (&REST X) ...)():
X        : (1)
NIL
```

We can move up the stack one level at a time looking at the function that called the current function (the previous function) using ":p" thus:

```
BOOT>>:p
Broken at |NEWGOGET|.
```

And again, we can look at the variables that can be accessed locally:

```
BOOT>>:bl
>> newGoGet():
Local0(G3629): (0 (#<vector 08b34bb4> 45 . char))
Local(1): #<compiled-function CHAR;char;S$;20>
Local(2): 0
Local(3): #<vector 08b233f0>
Local(4): 1
NIL
```

Here we see that the function newGoGet is calling CHAR;char;S$;20 which is a mangled form of the name of the original spad function. To decode this name we can see that the CHAR portion is used to identify the domain where the function lives. This domain, CHAR, comes from the source file "string.spad" which lives in "src/algebra/string.spad.pamphlet". To discover this we use the Axiom "asq" command with the "-so" (sourcefile) option at a standard shell prompt (NOT in the lisp prompt) thus:

```
asq -so CHAR
string.spad
```

If we look at the code in the string.spad.pamphlet file we find the following code signature:

```
 char: String   -> %
  ++ char(s) provides a character from a string s of length one.
```

and it's implementation code:

```
 char(s:String) ==
  (#s) = 1 => s(minIndex s) pretend %
  error "String is not a single character"
```

The string.spad file can be compiled at the command prompt. In particular, we can compile only the CHAR domain out of this file thus:

```
)co string.spad )con CHAR
```

This will produce a directory called CHAR.NRLIB containing 3 files:

```
ls CHAR.NRLIB
code.lsp index.kaf info
```

The info file contains information used by the spad compiler. We can ignore it for now.

The index.kaf file contains information that will go into the various Axiom database (.daase) files. The kaf file format is a random access file. The first entry is an integer that will be an index into the file that can be used in an operating system call to seek. In this case it will be an index which is the last used byte in the file. Go to the last expression in the file and we find:

```
(
 ("slot1Info" 0 11302)
 ("documentation" 0 9179)
 ("ancestors" 0 9036)
 ("parents" 0 9010)
 ("abbreviation" 0 9005)
 ("predicates" 0 NIL)
 ("attributes" 0 NIL)
 ("signaturesAndLocals" 0 8156)
 ("superDomain" 0 NIL)
 ("operationAlist" 0 7207)
 ("modemaps" 0 6037)
 ("sourceFile" 0 5994)
 ("constructorCategory" 0 5434)
 ("constructorModemap" 0 4840)
 ("constructorKind" 0 4831)
 ("constructorForm" 0 4817)
 ("NILADIC" 0 4768)
 ("compilerInfo" 0 2093)
 ("loadTimeStuff" 0 20))
```

This is a list of triples. Each triple has two interesting parts, the name of the data and the seek index of the data in the index.kaf file. So, for instance, if you want to know what source file contains this domain you can start at the top of the index.kaf file, move ahead 5994 bytes and you will be at the start of the string:

```
"/usr/local/axiom/src/algebra/string.spad"
```

The information in the index.kaf files are collected into the special databases (the .daase files). The stand-alone "asq" function can query these databases and answer questions. The kind of questions you can ask are the names in the list above.

The third file in the CHAR.NRLIB directory is the code.lsp file. This is the actual common lisp code that will be executed as a result of calling the various spad functions. The spad code from the char command was:

```
char(s:String) ==
  (#s) = 1 => s(minIndex s) pretend %
  error "String is not a single character"
```

which got compiled into the common lisp code:

```
(DEFUN |CHAR;char;S$;20| (|s| |$|)
 (COND
  ((EQL (QCSIZE |s|) 1)
    (SPADCALL |s|
      (SPADCALL |s| (QREFELT |$| 47))
```

```
      (QREFELT |$| 48)))
  ((QUOTE T)
    (|error| "String is not a single character"))))
```

To understand what is going on here we need to understand the low level details of Axiom's interface to Common Lisp. The "Q" functions are strongly typed (Quick) versions of standard common lisp functions. QCSIZE is defined in src/interp/vmlisp.lisp.pamphlet thus:

```
(defmacro qcsize (x)
 `(the fixnum (length (the simple-string ,x))))
```

This macro will compute the length of a string.

QREFELT is defined in the same file as:

```
(defmacro qrefelt (vec ind)
 `(svref ,vec ,ind))
```

This macro will return the element of a vector.

SPADCALL is defined in src/interp/macros.lisp.pamphlet as:

```
(defmacro SPADCALL (&rest L)
  (let ((args (butlast l)) (fn (car (last l))) (gi (gensym)))
   `(let ((,gi ,fn))
     (the (values t) (funcall (car ,gi) ,@args (cdr ,gi))))
   ))
```

This macro will call the last value of the argument list as a function and give it everything but the last argument as arguments to the function. There are confusing historical reasons for this I won't go into here.

So you can see that these are simply macros that will expand into highly optimizable (the optimizations depend on the abilities of the common lisp compiler) common lisp code.

The common lisp code computes the length of the string s. If the length is 1 then we call the minIndex function from string on s. The minIndex function is found by looking "in the domain". The compiler changes the minIndex function call into a reference into a vector. The 47th element of the vector contains the function minIndex.

```
      (SPADCALL |s| (QREFELT |$| 47))
```

This code is equivalent (ignoring the gensyms) to the call

```
(minIndex s)
```

The $ symbol refers to the domain. At runtime this amounts to a lookup of the "infovec". The compile-time infovec shown here:

```
(setf (get
 (QUOTE |Character|)
 (QUOTE |infovec|))
 (LIST
  (QUOTE
   #(NIL
     NIL
     NIL
     NIL
     NIL
     NIL
     (QUOTE |Rep|)
```

```
  (|List| 28)
  (|PrimitiveArray| 28)
  (0 . |construct|)
  (QUOTE |OutChars|)
  (QUOTE |minChar|)
  (|Boolean|)
  |CHAR;=;2$B;1|
  |CHAR;<;2$B;2|
  (|NonNegativeInteger|)
  |CHAR;size;Nni;3|
  (|Integer|)
  |CHAR;char;I$;6|
  (|PositiveInteger|)
  |CHAR;index;Pi$;4|
  |CHAR;ord;$I;7|
  |CHAR;lookup;$Pi;5|
  (5 . |coerce|)
  |CHAR;random;$;8|
  |CHAR;space;$;9|
  |CHAR;quote;$;10|
  |CHAR;escape;$;11|
  (|OutputForm|)
  |CHAR;coerce;$Of;12|
  (|CharacterClass|)
  (10 . |digit|)
  (|Character|)
  (14 . |member?|)
  |CHAR;digit?;$B;13|
  (20 . |hexDigit|)
  |CHAR;hexDigit?;$B;14|
  (24 . |upperCase|)
  |CHAR;upperCase?;$B;15|
  (28 . |lowerCase|)
  |CHAR;lowerCase?;$B;16|
  (32 . |alphabetic|)
  |CHAR;alphabetic?;$B;17|
  (36 . |alphanumeric|)
  |CHAR;alphanumeric?;$B;18|
  (|String|)
  |CHAR;latex;$S;19|
  (40 . |minIndex|)
  (45 . |elt|)
  |CHAR;char;S$;20|
  |CHAR;upperCase;2$;21|
  |CHAR;lowerCase;2$;22|
  (|SingleInteger|)))
(QUOTE
  #(|~=| 51 |upperCase?| 57 |upperCase| 62 |space| 67
    |size| 71 |random| 75 |quote| 79 |ord| 83 |min| 88
    |max| 94 |lowerCase?| 100 |lowerCase| 105 |lookup| 110
    |latex| 115 |index| 120 |hexDigit?| 125 |hash| 130
    |escape| 135 |digit?| 139 |coerce| 144 |char| 149
    |alphanumeric?| 159 |alphabetic?| 164 |>=| 169 |>| 175
    |=| 181 |<=| 187 |<| 193))
```

```
   (QUOTE NIL)
   (CONS
    (|makeByteWordVec2| 1 (QUOTE (0 0 0 0 0 0)))
    (CONS
      (QUOTE #(NIL |OrderedSet&| NIL |SetCategory&|
               |BasicType&| NIL))
      (CONS
        (QUOTE
         #((|OrderedFinite|)
           (|OrderedSet|)
           (|Finite|)
           (|SetCategory|)
           (|BasicType|)
           (|CoercibleTo| 28)))
        (|makeByteWordVec2| 52
          (QUOTE
            (1 8 0 7 9 1 6 0 17 23 0 30 0 31 2 30 12 32 0 33
             0 30 0 35 0 30 0 37 0 30 0 39 0 30 0 41 0 30 0
             43 1 45 17 0 47 2 45 32 0 17 48 2 0 12 0 0 1 1
             0 12 0 38 1 0 0 0 50 0 0 0 25 0 0 15 16 0 0 0 24
             0 0 0 26 1 0 17 0 21 2 0 0 0 0 1 2 0 0 0 0 1 1 0
             12 0 40 1 0 0 0 51 1 0 19 0 22 1 0 45 0 46 1 0 0
             19 20 1 0 12 0 36 1 0 52 0 1 0 0 0 27 1 0 12 0 34
             1 0 28 0 29 1 0 0 45 49 1 0 0 17 18 1 0 12 0 44 1
             0 12 0 42 2 0 12 0 0 1 2 0 12 0 0 1 2 0 12 0 0 13
             2 0 12 0 0 1 2 0 12 0 0 14))))))
    (QUOTE |lookupComplete|)))
```

Which is a 5 element list. This contains all kinds of information used at runtime by the compiled routines. In particular, functions are looked up at runtime in the first element of the infovec list. This first element contains 53 items (in this domain). Item 47 is

```
(40 . |minIndex|)
```

which is the minIndex function we seek.

At runtime this infovec lives on the property list of the domain name. The domain name of CHAR is Character. So we look on the property list (a lisp a-list) thus:

```
BOOT>>(symbol-plist '|Character|)

(SYSTEM:DEBUG (#:G85875)
 |infovec| (#<vector 08b34380>
            #<vector 08b34364>
            NIL
            (#<bit-vector 08b34310>
             #<vector 08b34348>
             #<vector 08b3432c> . #<vector 08b342f4>)
            |lookupComplete|)
 LOADED "/home/axiomgnu/new/mnt/linux/algebra/CHAR.o"
 NILADIC T
 PNAME "Character"
 DATABASE #S(DATABASE
             ABBREVIATION CHAR
             ANCESTORS NIL
             CONSTRUCTOR NIL
```

```
             CONSTRUCTORCATEGORY 228064
             CONSTRUCTORKIND |domain|
             CONSTRUCTORMODEMAP 227069
             COSIG (NIL)
             DEFAULTDOMAIN NIL
             MODEMAPS 227404
             NILADIC T
             OBJECT "CHAR"
             OPERATIONALIST 226402
             DOCUMENTATION 152634
             CONSTRUCTORFORM 152626
             ATTRIBUTES 154726
             PREDICATES 154731
             SOURCEFILE "string.spad"
             PARENTS NIL
             USERS NIL
             DEPENDENTS NIL
             SPARE NIL))
```

This list is organized contains many runtime lookup items (notice the PNAME entry is "Character", the LOADED entry says where the file came from, the DATABASE structure entry has database indicies (see daase.lisp.pamphlet for the structure definition), etc).

Lets get the property list

```
BOOT>>(setq a (symbol-plist '|Character|))

(SYSTEM:DEBUG (#:G85875)
 |infovec| (#<vector 08b34380>
            #<vector 08b34364>
            NIL
            (#<bit-vector 08b34310>
             #<vector 08b34348>
             #<vector 08b3432c> . #<vector 08b342f4>)
            |lookupComplete|)
 LOADED "/home/axiomgnu/new/mnt/linux/algebra/CHAR.o"
 NILADIC T
 PNAME "Character"
 DATABASE #S(DATABASE
             ABBREVIATION CHAR
             ANCESTORS NIL
             CONSTRUCTOR NIL
             CONSTRUCTORCATEGORY 228064
             CONSTRUCTORKIND |domain|
             CONSTRUCTORMODEMAP 227069
             COSIG (NIL)
             DEFAULTDOMAIN NIL
             MODEMAPS 227404
             NILADIC T
             OBJECT "CHAR"
             OPERATIONALIST 226402
             DOCUMENTATION 152634
             CONSTRUCTORFORM 152626
             ATTRIBUTES 154726
             PREDICATES 154731
```

```
            SOURCEFILE "string.spad"
            PARENTS NIL
            USERS NIL
            DEPENDENTS NIL
            SPARE NIL))
```

Next we get the infovec value

```
BOOT>>(setq b (fourth a))

(#<vector 08b34380>
 #<vector 08b34364>
 NIL
 (#<bit-vector 08b34310>
  #<vector 08b34348>
  #<vector 08b3432c> . #<vector 08b342f4>)
 |lookupComplete|)
```

Then we get the function table

```
BOOT>>(setq c (car b))

#<vector 08b34380>
```

In this common lisp (GCL) the array is identified by it's memory address.

Notice that it has the right number of entries:

```
BOOT>>(length c)

53
```

And we can ask for the 47th entry thus:

```
BOOT>>(elt c 47)

(40 . |minIndex|)
```

Later we end up calling the 48th function (which is elt and returns the actual character in the string). We ask for it:

```
BOOT>>(elt c 48)

(45 . |elt|)
```

At this point we've reached the metal. Common lisp will evaluate the macro-expanded functions and execute the proper code. Essentially the compiler has changed all of our spad code into runtime table lookups.

0.18 The axiom.input startup file

If you add a file in your home directory called ".axiom.input" it will be read and executed when Axiom starts. This is useful for various reasons including setting various switches. Mine reads:

```
)lisp (pprint ‘‘running /root/.axiom.input’’)
)set quit unprotected
)set message autoload off
)set message startup off
```

You can execute any command in .axiom.input. Be aware that this will ALSO be run while you are doing a "make" so be careful what you ask to do.

0.19 Where are Axiom symbols stored?

You'd think that your question about where the symbol is interned would be easy to answer but it is not. The top level loop uses Bill Burge's dreaded zipper parser. You can see it in action by executing the following sequence:

```
)lisp (setq $DALYMODE t)
  ; this is a special mode of the top level interpreter. If
  ; $DALYMODE is true then any top-level form that begins
  ; with an open-paren is considered a lisp expression.
  ; For almost everything I ever do I end up peeking at the
  ; lisp so this bit of magic helps.
(trace |intloopProcessString|)
  ; from int-top.boot.pamphlet
(trace |intloopProcess|)
  ; the third argument is the "zippered" input
(trace |intloopSpadProcess|)
  ; now it is all clear, no? sigh.
(trace |phInterpret|)
  ; from int-top.boot.pamphlet
(trace |intInterpretPform|)
  ; from intint.lisp.pamphlet
(trace |processInteractive|)
  ; from i-toplev.boot.pamphlet
(setq |$reportInstantiations| t)
  ; shows what domains were created
(setq |$monitorNewWorld| t)
  ; watch the interpreter resolve operations
(trace |processInteractive1|)
  ; from i-toplev.boot.pamphlet
```

ah HA! I remember now. There is the notion of a "frame" which is basically a namespace in Axiom or an alist in Common Lisp. It is possible to maintain different "frames" and move among them. There is the notion of the current frame and it contains all the defined variables. At any given time the current frame is available as $InteractiveFrame. This variable is used in processInteractive1. If you do:

```
a:=7
(pprint |$InteractiveFrame|)
```

you'll see —a— show up on the alist. When you do the

```
pgr:=MonoidRing(Polynomial PrimeField 5, Permutation Integer)
p:pgr:=1
```

you'll see —p— show up with 2 other things: (—p— mode value) where mode is the "type" of the variable. The value is the internal value. In this case MonoidRing has an internal representation. You can find out what the internal representation of a MonoidRing is by first asking where the source file is:

```
(do this at a shell prompt, not in axiom)
asq -so MonoidRing ==> mring.spad

     -- or -- in Axiom type:

)show MonoidRing
```

and you'll see a line that reads:

```
Issue )edit (yourpath)/../../src/algebra/mring.spad
```

If you look in mring.spad.pamphlet you'll see line 91 that reads:

```
   Rep := List Term
```

which says that we will store elements of type MonoidRing as a list of Term objects. Term is defined in the same file (as a macro, which is what '==>' means in spad files) on line 43:

```
   Term ==> Record(coef: R, monom: M)
```

which means that elements of a MonoidRing are Lists of Records. The 'R' is defined on line 42 as the first argument to MonoidRing which in this case is "Polynomial PrimeField 5". The "M" is also defined on line 42 as the second argument to MonoidRing and in this case is "Permutation Integer". So the real representation is

```
  List Record(coef: Polynomial PrimeField 5,
              monom: Permutation Integer)
```

In the $InteractiveFrame we printed out you can see in the value field that the value is:

```
(|value|
  (|MonoidRing| (|Polynomial| (|PrimeField| 5))
                (|Permutation| (|Integer|)))
   WRAPPED ((0 . 1) . #<vector 08af33d4>))
```

which basically means that we know how the MonoidRing was constructed and what it's current value is. The (0 . 1) likely means that this is the zeroth (constant) term with a leading coefficient of 1. This is just a guess as I haven't decoded the representation of either Polynomial PrimeField or Permutation Integer. You can do the same deconstruction of these two domains by setting

```
pi:=Permutation Integer
z:pi:=1

pp5:=Polynomial PrimeField 5
w:pp5:=1

and following the same steps as above:
 (pprint |$InteractiveFrame|)
 )show pi
 (find the source file)
 (find the representation and decode it)

 (pprint |$InteractiveFrame|)
```

```
)show pp5
(find the source file)
(find the representation and decode it)
```

Be sure to set $DALYMODE to nil if you plan to use Axiom for any real computation. Otherwise every expression that begins with an open-paren will go directly to lisp.

0.20 Translating individual boot files to common lisp

If you are making changes to boot code it is sometimes helpful to check the generated lisp code to ensure it does what you want. You can convert an individual boot file to common lisp using the boottran::boottocl function:

```
)fin       -- drop into common lisp
(boottran::boottocl "foo.boot")
```

when you do this it creates a foo.clisp file in ../../int/interp

Alternatively if you work from the pamphlet file the process is more painful as you have to do

```
)cd (yourpath)/int/interp
)sys tangle ../../src/interp/foo.boot.pamphlet >foo.boot
)fin
(boottran::boottocl "foo.boot")
(restart)
```

The)cd step tells axiom to cd to the int/interp subdirectory. The)sys tangle... extracts the boot file from the pamphlet file The)fin step drops into common lisp The (bootran... converts the foo.boot file to foo.clisp The (restart) re-enters the top level loop

0.21 Directories

For this discussion I assume that you have your system rooted at /spad and was build to run on linux. These directories may not yet be in the CVS tree but are documented here so they make sense when the show up.

The AXIOM variable

The usual setting of the AXIOM variable is /spad/mnt/linux. The name is composed of three parts, the rooted path, in this case /spad, “mnt”, and the system you are running, in this case linux. Builds for other systems will have other system names.

/spad

This is the usual root directory of the Axiom system. The name is historical, a contraction of Scratchpad. This name can be anything provided the shell variable AXIOM contains the new prefix.

/spad/mnt

This is a directory which contains files which are specific to a given platform. At a site that contains multiple platforms this directory will contain a subdirectory for each type of platform (e.g. linux, rios, ps2, rt, sun, etc).

/spad/mnt/linux

This directory contains the complete copy of the Axiom system for the linux system. This is the 'mount point' of the system. Executable systems (for RedHat) are shipped relative to this point. In what follows, the ./ refers to /spad/mnt/linux.

```
******************************************************
There are several directories explained below. They are:

./bin     -- user executables
./doc     -- system documentation
./algebra -- algebra libraries
./lib     -- system executables
./etc     -- I haven't a clue....
******************************************************
```

0.21.1 The mnt/linux/bin directory

./bin

This is a directory of user executable commands, either at the top level or thru certain Axiom system calls. Support executables live in ./lib

./bin/htadd

This adds pages to the Hyperdoc database (ht.db, which lives in ./doc/hypertex/pages; hypertex, since we have a penchant for these things, is an historical name for Hyperdoc. The single word 'lawyers' will probably explain away a lot of name changes.)

./bin/spadsys

This is the Axiom interpreter. It is one of the functions started when the user invokes the system using the spadsys command. Normally this command is run under the control of sman (./lib/sman) and the console is under the control of clef (./bin/clef), the wonderous command-line editor. It is possible to start spadsys standalone but it will not talk to Hyperdoc or graphics. Users who rlogin or use an ascii-only terminal (for historical reasons, no doubt) can profit by invoking spadsys directly rather than using ./bin/axiom

./bin/axiom

This is a shell script that spins the world. It kicks off a whole tree of processes necessary to perform the X-related magic we do. It expects the shell variable AXIOM to be set to the 'mount point' (usually to /spad/mnt/linux).

./bin/clef

This is the wonderous command-line editor used by Axiom. It can be used in a stand-alone fashion if you wish.

./bin/SPADEDFN

This script is invoked by the spad)fe command. It can be changed to invoke your favorite editor. While you may invoke your editor, it may not run (as in, yes, I can invoke the devil but will he come when I call?)

./bin/viewalone

This is a function to run the graphics in a stand-alone fashion. The Graphics package (an amazing contribution by several very talented people, most notably Jim Wen and Jon Steinbach) is a C program that communicates with Axiom thru sockets. It will, however, perform its miracles unaided if invoked by the sufficiently chaste...

./bin/hypertex

This is a function to run Hyperdoc (remember the penchant!) stand-alone. The Hyperdoc package owes its existence to the efforts of J.M. Wiley and Scott Morrison. This function works off 'pages' that live in hypertex pages directory and are referenced in the "hyperdoc database" called ht.db (for historical reasons, but you knew that). It is possible for creative plagerists to figure out how to write their own pages and add them to the database (see htadd above), thus gaining fame far and wide...

./bin/sys-init.lsp

This is a file of lisp code that gets loaded before Axiom starts. Thus, we distribute patches by adding lisp (load ...) commands to this file. The sufficiently clever should have a field day with this one. (All others should worship the sufficiently clever and send them money, eh?)

./bin/init.lsp

This is a file of lisp code loaded if and only if you start spadsys in this directory. The user can put a file of this name in her home directory and it will get loaded at startup with the probable effect of injecting luser errors into the running system. sigh.

0.21.2 The mnt/linux/doc directory

./doc

The doc subdirectory contains system documentation.

./doc/command.list

This is a file of command completions used by clef when you hit the tab key. This is a little known feature that will surprise someone someday (hopefully pleasantly).

./doc/book

This is an attempt at a book describing Axiom. It represents a combination of fantasy, describing what never will be and history (remember the penchant?) describing what was. Any description matching what is may be regarded as failure of the imagination and ignored.

./doc/compguide

This is an attempt to describe a compiler that doesn't exist, never did exist, and never will exist. It makes for entertaining reading so we included it.

./doc/hypertex

This is the fabled Hyperdoc subdirectory where all of the pages and the database live, along with several other obscure files needed to make the wizards look good.

./doc/hypertex/pages

This is where the 'pages' live. Each file ending in .ht contains several pages related, if only by chance, to the same topic. You may find it instructive to try to read some of these files. Hyperdoc was learned by the 'campfire' method (sitting around the fire passing along historical facts by word of mouth) and will probably continue to propagate by the same method. Ye may become th' local scribe and soothsayer if ye study the writings here below....

./doc/hypertex/pages/rootpage.ht

This file is the magic 'first page' that gets displayed when Hyperdoc starts. There is a macro (see ./doc/hypertex/pages/util.ht) called /localinfo which is intended to allow the luser to add her own pages without modifying the system copies. How this is done was lost when the campfire got rained out.

./doc/hypertex/pages/util.ht

This file contains the macros used to extend the system commands. The syntax is hard to learn (it was hard to write, it ought to be hard to learn, eh?).

./doc/hypertex/pages/ht.db

This is the Hyperdoc database. It is updated using ./bin/htadd which must be run whenever a page in this directory gets changed. The necessary arguments to htadd are obvious to those in the know.

./doc/hypertex/bitmaps

There are several pretty bitmaps used as cursors, buttons and general decorations that hide in this directory.

./doc/hypertex/ht.files

This is a list of some Hyperdoc files. It seems to have no purpose in life but it is useful as a koan, as in, What is the length of half a list?

./doc/hypertex/ht.db

Another copy of the Hyperdoc database. It isn't clear which one is the real one so I guess we keep both. Maybe we'll figure it out at the friday night campfire provided we don't get too lit.

./doc/hypertex/gloss.text

The text used in the glossary. Many magic words lie herein. Some are spoken only by campfire gurus.

./doc/library

This is a directory of Hyperdoc pages that can be freely smashed, trashed and generally played with. It uses the /localinfo connection to set up a 'library' containing Hyperdoc pages keyed to your favorite textbook. It is interesting to set the shell variable
HTPATH=/spad/mnt/linux/doc/library:
/spad/mnt/linux/doc/hypertex/pages
and then start Hyperdoc. See the file ./doc/library/macros.ht

./doc/msgs

This directory contains several 'message databases'; the only one of which we seem to care about being s2-us.msgs but I can't swear to it.

./doc/spadhelp

This is a directory containing help information for a copy of the system that once ran long ago and far away. It is kept for historical reasons (programmers NEVER throw anything away).

./doc/viewports

There are several dozen truly fine pictures in Axiom. We have created them and hidden them here. Hyperdoc will insert them at various places (where the text gets too boring, hopefully) and you can click on them there. They get snarfed from here. It is possible to view them with stand-alone graphics but don't ask me how. I missed that campfire due to poisoned marshmellows.

./doc/complang

This directory contains fantasy from the past as opposed to facts from the future. Ignore it.

./doc/ug

This directory left intentionally blank :-) (an old IBM joke).

./doc/tex

These are the files necessary to create the famous goertler document. If you figure out how to use these please send us the instructions and we will add a log to the campfire with your name on it (a rare honor indeed as luser's names rarely reach the inner circle).

./doc/htex

This directory contains the original tex-like source for the luser's guide. There are many functions that munch on these between here and paper but this is approximately where they start. If you do your own algebra perchance you might document it like this. Figuring out the syntax will also get your name into the inner circle (probably connnected with a smirk :-))

./doc/newug

Please don't ask me. I couldn't begin to guess. You wouldn't believe how many 'new' things there are that really aren't. We have more NEW things than Madison Avenue has NEW laundry soap.

./doc/gloss.text

This one is here because it is here. Existentially speaking, of course.

./doc/submitted

This was what the htex files said before history was rewritten... (and renamed?)

0.21.3 The mnt/linux/algebra directory

./algebra

This is where all of the interesting action lives. Each .NRLIB directory contains 2 files, a code.o and an index.kaf* file. The code.o contains the executable algebra that gets loaded into the system. The index.kaf* file contains all kinds of things like signatures, source paths, properties and dried bat droppings. The documentation for each of these can be reached by using the BROWSE feature of Hyperdoc.

./algebra/MODEMAP.daase

This is an inverted database that contains information gleaned from the index.kaf* files. Without this there is no way to figure out which .NRLIB file to load. This database is opened on startup and kept open.

./algebra/interp.exposed

This is a control file for the interpeter that limits the number of places to search for function names.

****

0.21.4 The mnt/linux/etc directory

./lib

This directory contains functions that get loaded by the system. Nothing in here is executable by the user but the system needs these functions to run.

./lib/htrefs
./lib/htsearch
./lib/hthits

These three functions are used to search the Hyperdoc pages. There is no way in the current system to request a search of those pages so these files are fascinating examples of history in the making...

./lib/hypertex

This is Hyperdoc. What is in a name?

./lib/sman

This is sman, which comes before all. Methinks the name originated as a contraction of superman, the name of a stack frame in a system long ago and far away (VMLisp) chosen because a certain programmer had a penchant for comic books when he was young.

./lib/session
./lib/spadclient

These two files are processes started by sman for some reason or other. I can never remember what they do or why. However, the campfire fails to smoke if they don't work.

./lib/viewman

This is the controlling function for the graphics.

./lib/view2d

This is invoked when a 2 dimensional window is requested. This is provided mostly for those math majors who never got over the insights from flatland.

./lib/view3d

This is invoked when a 3 dimensional window is requested. Option IBM3634-A is required to convert your 2 dimensional screen to 3 dimensions for realistic viewing. A mathematically accurate, if somewhat more achievable, rendering can be had on a color or monochrome crt without this upgrade.

./lib/gloss.text
./lib/glosskey.text
./lib/glossdef.text

These are three files related to the glossary. The first (gloss.text) is the original glossary text. The second (glosskey.text) is a list of terms and pointers into glossdef.text. The third (glossdef.text for those math majors who can't count) is a list of definitions and pointers back into the second (guess). These files are used by Hyperdoc.

./lib/browsedb.lisp

This is the original file that creates an in-memory hash table used by browse. It is used during system build time. We keep it here to ensure that the bytes on this section of the disk have a well-defined orientation, allowing us to compute the spin vectors of the individual magnetic domains. This allows us to give Heisenburg a sense of direction (at least over the long run).

./lib/comdb.text
./lib/libdb.text

The first file (comdb.text) contains the so-called ++ (plus plus) comments from the algebra files. It contains pointers into the second file. The second file (libdb.text) contains flags (constructor, operation, attribute) and pointers into the first file. These files are used by browse in Hyperdoc.

./lib/loadmprotect
./lib/mprotect

This set of two files has been mercifully de-installed from the system. They will, if used and despite the meaning behind the name, cause random system reboots (yeah, HARDWARE reboots. don't ask me how, I'm just the historian).

./lib/SPADEDIT
./lib/fc
./lib/spadbuf
./lib/SPADEDFN
./lib/obey

./lib/ex2ht

I've drawn a blank; intentionally.

0.21.5 The mnt/linux/lib directory

./etc

This directory intentionally left blank. We just can't figure out WHY we intended to leave it blank. Historical reasons, no doubt.

0.22 The)set command

The **)set** command contains many possible options such as:

```
                    Current Values of )set Variables

Variable  Description                                Current Value
------------------------------------------------------------------
breakmode execute break processing on error          break
compiler  Library compiler options                   ...
expose    control interpreter constructor exposure   ...
functions some interpreter function options          ...
fortran   view and set options for FORTRAN output    ...
kernel    library functions built into the kernel for
          efficiency ...
hyperdoc  options in using HyperDoc                  ...
help      view and set some help options             ...
history   save workspace values in a history file    on
messages  show messages for various system features  ...
naglink   options for NAGLink                        ...
output    view and set some output options           ...
quit      protected or unprotected quit              unprotected
streams   set some options for working with streams  ...
system    set some system development variables      ...
userlevel operation access level of system user      development

Variables with current values of ... have further
sub-options. For example,
issue )set  system to see what the options are for system .
For more information, issue )help set .
```

The table that contains these options lives in setvart.boot.pamphlet. The actual code that implements these options is sprinkled around but most of the first-level calls resolve to functions in setvars.boot.pamphlet. Thus if you plan to add a new output style to the system, or figure out where a current style is broken, these two files are the place to start.

A new)set breakmode command has been implemented to handle the case that you might want an error message or an error return code from AXIOMsys. You can set this option with

```
  )set breakmode quit
```

This will cause AXIOMsys to exit with the return code of 1. Note that if you invoke the "axiom" shell script to start AXIOMsys you will not see this return code (sman swallows it).

0.23 Special Output Formats

The first level of special output formatting is handled by functions in setvart.boot.pamphlet. This handles the options given to the)set command.

0.24 Hand creating the hyperdoc binary

First we need **tanglec** which is used to extract the required chunks.

We extract the source code we need from the books and compile them. Then we copy the executable to the correct point in Axiom's execution path. Now you can modify the source code to debug.

We use the

```
gcc -o tanglec books/tanglec.c
tanglec books/bookvol7.pamphlet hypertex >hypertex.c
tanglec books/bookvol8.pamphlet sockio-c.c >sockio-c.c
tanglec books/bookvol8.pamphlet bsdsignal.c >bsdsignal.c
tanglec books/bookvol8.pamphlet spadcolors.c >spadcolors.c
tanglec books/bookvol8.pamphlet pixmap.c >pixmap.c
tanglec books/bookvol8.pamphlet util.c >util.c
gcc -c sockio-c.c
gcc -c bsdsignal.c
gcc -c spadcolors.c
gcc -c pixmap.c
gcc -c util.c
gcc -O2 -fno-strength-reduce -Wall -D_GNU_SOURCE -DLINUXplatform
   -I/usr/X11/include -L/usr/lib/x86_64-linux-gnu/ -L/usr/X11R6/lib
   -o hypertex hypertex.c spadcolors.o pixmap.o bsdsignal.o sockio-c.o
   util.o -lXpm -lX11 -lm
cp hypertex mnt/ubuntu/bin
axiom
```

0.25 Low Level Debugging Techniques

It should be observed that Axiom is basically Common Lisp and some very low level techniques can be used to find where problems occur in algebra code. This section walks thru a small problem and illustrates some techniques that can be used to find bugs. The point of this exercise is to show a few techniques, not to show a general method.

0.25.1 Finding Anonymous Function Signatures

This is a technique, adapted from Waldek Hebisch, for asking the interpreter to reveal the actual function that will be called in a given circumstance. Here we have a function tanint

from the domain ElementaryIntegration.

```
tanint(f, x, k) ==
  eta' := differentiate(eta := first argument k, x)
  r1  := tanintegrate(univariate(f, k), differentiate(#1,
          differentiate(#1, x), monomial(eta', 2) + eta'::UP),
            rischDEsys(#1, 2 * eta, #2, #3, x, lflimitedint(#1, x, #2),
               lfextendedint(#1, x, #2)))
  map(multivariate(#1, k), r1.answer) + lfintegrate(r1.a0, x)
```

We would like to know the type signature of the first argument to the inner call to the differentiate function:

```
          differentiate(#1, x), monomial(eta', 2) + eta'::UP),
```

We see that differentiate is called with `#1`, which is Axiom's notation for an anonymous function. How can we determine the signature?

Axiom has a second notation for anonymous functions using the `+->` notation. This notation allows you to explicitly specify type information. In the above code, we would like to replace the `#1` variable with the `+->` and explicit type information.

The first step is to look at the output of the Spad compiler. The abbreviation for ElementaryIntegration can be found from the interpreter by:

```
)show ElementaryIntegration
  Abbreviation for ElementaryIntegration is INTEF
```

So the compiler output is in the int/algebra/INTEF.nrlib/code.lsp file.

There we see the definition of the lisp tanint function. Notice that the `$` is a hidden, internal fourth argument to an Axiom three argument function. This is the vector of the current domain containing slots where we can look up information, called the domain vector.

```
(DEFUN |INTEF;tanint| (|f| |x| |k| $)
 (PROG (|eta| |eta'| |r1|)
  (RETURN
   (SEQ
    (LETT |eta'|
     (SPADCALL
       (LETT |eta|
        (|SPADfirst|
          (SPADCALL |k| (QREFELT $ 18)))
          |INTEF;tanint|)
       |x|
       (QREFELT $ 19))
     |INTEF;tanint|)
    (LETT |r1|
     (SPADCALL
       (SPADCALL |f| |k| (QREFELT $ 22))
       (CONS (FUNCTION |INTEF;tanint!1|) (VECTOR |eta'| |x| $))
       (CONS (FUNCTION |INTEF;tanint!4|) (VECTOR |x| $ |eta|))
       (QREFELT $ 50))
     |INTEF;tanint|)
    (EXIT
     (SPADCALL
      (SPADCALL
        (CONS
          (FUNCTION |INTEF;tanint!5|)
```

```
          (VECTOR $ |k|))
        (QCAR |r1|)
        (QREFELT $ 57))
      (SPADCALL (QCDR |r1|) |x| (QREFELT $ 58))
      (QREFELT $ 59)))))))
```

The assignment line for `eta'` is:

```
eta' := differentiate(eta := first argument k, x)
```

which is implemented by the code:

```
(LETT |eta'|
 (SPADCALL
   (LETT |eta|
    (|SPADfirst|
      (SPADCALL |k| (QREFELT $ 18)))
      |INTEF;tanint|)
   |x|
   (QREFELT $ 19))
 |INTEF;tanint|)
```

from which we see that the inner differentiate is slot 19 in the domain vector. Every domain has an associated domain vector which contains references to other functions from other domains, among other things. The QREFELT function takes the domain vector `$` and slot number and does a "quick array reference". The return value is a pair, the car of which is a function to call. The SPADCALL macro uses the last argument, in this case the result of `(QREFELT $ 19)` to find the function to call.

The function from slot 19 can be found with:

```
)lisp (setq $dalymode t)
(setf *print-circle* t)
(setf *print-array* nil)
(setf dv (|ElementaryIntegration| (|Integer|) (|Expression| (|Integer|))))
(|replaceGoGetSlot| (cdr (aref dv 19)))
Value = (#<compiled-function |FS-;differentiate;SSS;99|> . #<vector 090cbccc>)
```

The call of `(setq $dalymode t)` changes the Axiom top level loop to interpret any input that begins with an open parenthesis to be interpreted as a lisp s-expression rather than Axiom input. This saves typing `)lisp` in front of every lisp expression. Be sure to do a `(setq $dalymode nil)` when you are finished.

The *print-circle* needs to be true because the domain vector contains circular references to itself and we need to make sure that we check for this during printing so the print is not infinite.

The *print-array* needs to be nil so that the arrays just print some identifying information rather than the detailed array contents.

The `(setf dv ...` uses the Lisp internal names for the domains. In Axiom, the names of types are case-sensitive symbols. These are represented in lisp surrounded by vertical bars because lisp is not case sensitive. The dv variable is essentially being set to the Axiom equivalent of:

```
dv:=ElementaryIntegration(Integer,Expression(Integer))
```

except we do this in lisp. The end result is that dv will contain the domain vector for the newly constructed domain. From the lisp code

Consider the call of the form:

```
(SPADCALL A B '(C . D))
```

The SPADCALL macro takes a set of arguments, the last of which is a pair where C is the function to call and D is the domain vector. So if we do:

```
(macroexpand-1 '(spadcall a b '(c . d)))
Value =
  (LET ((#0=#:G1417 (QUOTE (C . D))))
   (THE (VALUES T) (FUNCALL (CAR #0#) A B (CDR #0#))))
```

Note that `#0` is a "pointer", in this case to the list '(c) and `#0#` is a use of that pointer. This is done to make sure that you reference the exact cons cell of the argument.

In Axiom compiler output

```
(SPADCALL eta k (QREFELT $ 19))
```

approximately translates to

```
(FUNCALL (CAR (QREFELT $ 19)) eta k (CDR (QREFELT $ 19)))
```

which calls the function from the domain slot 19 on the value assigned to eta and the variable k and the domain. Thus, the full expansion becomes

```
(FUNCALL #<compiled-function |FS-;differentiate;SSS;99|>
  eta k #<vector 090cbccc>)
```

From this we can see a reference to `FS-;differentiate;SSS;99` which is the internal name of the differentiate function from the `FS-` category.

Note that FunctionSpace is a category. When categories contain implementation code the compiler generates 2 nrlibs. The Axiom convention for categorical implementation of code using a trailing "-" so the actual code for `FS-;differentiate;SSS;99` lives in int/algebra/FS-.nrlib/code.lsp

We can see that the differentiate function is coming from the category

```
)show FS
 FunctionSpace R: OrderedSet  is a category constructor
 Abbreviation for FunctionSpace is FS

 ....

 differentiate : (%,Symbol) -> % if R has RING
 differentiate : (%,List Symbol) -> % if R has RING
 differentiate : (%,Symbol,NonNegativeInteger) -> % if R has RING
 differentiate : (%,List Symbol,List NonNegativeInteger) -> % if R has RING
```

From the above signatures we know there is only one differentiate that is a two argument form so the call

```
              differentiate(#1, x), monomial(eta', 2) + eta'::UP),
```

must be the first instance.

From the sources (bookvol10.4) we see that the tanint function has the signature:

```
    tanint      : (F, SE, K) -> IR
```

and that

```
  SE  ==> Symbol
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
```

```
          FunctionSpace R)
  K   ==> Kernel F
```

The differentiate function takes something of type F and a Symbol and returns something of type F. If we write this as an anonymous function it becomes:

```
(x2 : F) : F +-> differentiate(x2, x)
```

Thus, we can rewrite the differentiate call as:

```
              differentiate(#1, x), monomial(eta', 2) + eta'::UP),
```

as

```
                  (x2 : F) : F +-> differentiate(x2, x),
                  monomial(eta', 2) + eta'::UP),
```

Continuing in this way we can fully rewrite the assignments as:

```
      r1  := tanintegrate(univariate(f, k),
                (x1 : UP) : UP +-> differentiate(x1,
                   (x2 : F) : F +-> differentiate(x2, x),
                   monomial(eta', 2) + eta'::UP),
                (x6 : Integer, x2 : F, x3 : F) : Union(List F, "failed") +->
                   rischDEsys(x6, 2 * eta, x2, x3, x,
                     (x4 : F, x5 : List F) : U3 +-> lflimitedint(x4, x, x5),
                     (x4 : F, x5 : F) : U2 +-> lfextendedint(x4, x, x5)))
      map((x1 : RF) : F +-> multivariate(x1, k), r1.answer) + _
            lfintegrate(r1.a0, x)
```

Note that rischDEsys is tricky, because rischDEsys returns only List F, but tanintegrate expects union.

0.25.2 The example bug

Axiom can generate TeX output by typing:

```
)set output tex on
```

Here we give an example of TeX output that contains a bug:

```
(1) -> )set output tex on
(1) -> radix(10**10,32)
   Loading /axiom/mnt/linux/algebra/RADUTIL.o for package
      RadixUtilities
   Loading /axiom/mnt/linux/algebra/RADIX.o
     for domain RadixExpansion
   Loading /axiom/mnt/linux/algebra/ANY1.o
     for package AnyFunctions1
   Loading /axiom/mnt/linux/algebra/NONE1.o
     for package NoneFunctions1
   Loading /axiom/mnt/linux/algebra/ANY.o
     for domain Any
   Loading /axiom/mnt/linux/algebra/SEX.o
     for domain SExpression

   (1)  9A0NP00
   Loading /axiom/mnt/linux/algebra/TEX.o
     for domain TexFormat
   Loading /axiom/mnt/linux/algebra/CCLASS.o
```

```
     for domain CharacterClass
   Loading /axiom/mnt/linux/algebra/IBITS.o
     for domain IndexedBits
   Loading /axiom/mnt/linux/algebra/UNISEG.o
     for domain UniversalSegment
$$
9#\A0#\N#\P00
\leqno(1)
$$
   Loading /axiom/mnt/linux/algebra/VOID.o for domain Void

                                        Type: RadixExpansion 32
```

The correct output should be:

```
$$
9A0NP00
\leqno(1)
$$
```

So we need to figure out where the # prefixes are being generated. In the above code we can see various domains being loaded. These domains are lisp code. Each domain lives in a subdirectory of its own. For example, the ANY domain lives in ANY.NRLIB. The ANY.NRLIB directory contains a common lisp file named code.lsp. The compiled form of this code ANY.o is loaded whenever the domain Any is referenced. We can look at the lisp code:

```
(/VERSIONCHECK 2)

(PUT (QUOTE |ANY;obj;$N;1|)
     (QUOTE |SPADreplace|)
     (QUOTE QCDR))

(DEFUN |ANY;obj;$N;1| (|x| $) (QCDR |x|))

(PUT (QUOTE |ANY;dom;$Se;2|)
     (QUOTE |SPADreplace|)
     (QUOTE QCAR))

(DEFUN |ANY;dom;$Se;2| (|x| $) (QCAR |x|))

(PUT (QUOTE |ANY;domainOf;$Of;3|)
     (QUOTE |SPADreplace|)
     (QUOTE QCAR))

(DEFUN |ANY;domainOf;$Of;3| (|x| $) (QCAR |x|))

(DEFUN |ANY;=;2$B;4| (|x| |y| $)
 (COND
  ((SPADCALL (QCAR |x|) (QCAR |y|) (QREFELT $ 17))
    (EQ (QCDR |x|) (QCDR |y|)))
  ((QUOTE T) (QUOTE NIL))))

(DEFUN |ANY;objectOf;$Of;5| (|x| $)
 (|spad2BootCoerce|
  (QCDR |x|)
```

```
  (QCAR |x|)
  (SPADCALL
   (SPADCALL "OutputForm" (QREFELT $ 21))
   (QREFELT $ 23))))

(DEFUN |ANY;showTypeInOutput;BS;6| (|b| $)
 (SEQ
  (SETELT $ 10 (SPADCALL |b| (QREFELT $ 9)))
  (EXIT
   (COND
    (|b| "Type of object will be displayed in
          output of a member of Any")
    ((QUOTE T) "Type of object will not be displayed in
                output of a member of Any")))))

(DEFUN |ANY;coerce;$Of;7| (|x| $)
 (PROG (|obj1| |p| |dom1| #0=#:G1426 |a| #1=#:G1427)
  (RETURN
   (SEQ
    (LETT |obj1|
          (SPADCALL |x| (QREFELT $ 24))
          |ANY;coerce;$Of;7|)
    (COND
     ((NULL (SPADCALL (QREFELT $ 10) (QREFELT $ 26)))
       (EXIT |obj1|)))
    (LETT |dom1|
     (SEQ
      (LETT |p|
       (|prefix2String| (|devaluate| (QCAR |x|)))
       |ANY;coerce;$Of;7|)
      (EXIT
       (COND
        ((SPADCALL |p| (QREFELT $ 27))
         (SPADCALL |p| (QREFELT $ 23)))
        ((QUOTE T) (SPADCALL |p| (QREFELT $ 29)))))))
     |ANY;coerce;$Of;7|)
    (EXIT
     (SPADCALL
      (CONS |obj1|
       (CONS ":"
        (PROGN
         (LETT #0# NIL |ANY;coerce;$Of;7|)
         (SEQ
          (LETT |a| NIL |ANY;coerce;$Of;7|)
          (LETT #1# |dom1| |ANY;coerce;$Of;7|)
          G190
          (COND
           ((OR (ATOM #1#)
                (PROGN
                  (LETT |a| (CAR #1#) |ANY;coerce;$Of;7|)
                  NIL))
              (GO G191)))
          (SEQ
           (EXIT
```

```
            (LETT #0#
             (CONS
              (SPADCALL |a| (QREFELT $ 30))
               #0#)
             |ANY;coerce;$Of;7|)))
          (LETT #1# (CDR #1#) |ANY;coerce;$Of;7|)
          (GO G190)
          G191
          (EXIT (NREVERSE0 #0#))))))
      (QREFELT $ 31)))))))

(DEFUN |ANY;any;SeN$;8| (|domain| |object| $)
 (SEQ
  (COND
   ((|isValidType| |domain|) (CONS |domain| |object|))
   ((QUOTE T)
    (SEQ
     (LETT |domain| (|devaluate| |domain|) |ANY;any;SeN$;8|)
     (EXIT
      (COND
       ((|isValidType| |domain|) (CONS |domain| |object|))
       ((QUOTE T)
        (|error|
"function any must have a domain as first argument")))))))))

(DEFUN |Any| NIL
 (PROG NIL
  (RETURN
   (PROG (#0=#:G1432)
    (RETURN
     (COND
      ((LETT #0#
             (HGET |$ConstructorCache| (QUOTE |Any|))
             |Any|)
        (|CDRwithIncrement| (CDAR #0#)))
      ((QUOTE T)
        (UNWIND-PROTECT
         (PROG1
          (CDDAR
           (HPUT |$ConstructorCache|
                 (QUOTE |Any|)
                 (LIST (CONS NIL (CONS 1 (|Any;|))))))
          (LETT #0# T |Any|))
         (COND
          ((NOT #0#)
            (HREM |$ConstructorCache| (QUOTE |Any|)))))))))))))

(DEFUN |Any;| NIL
 (PROG (|dv$| $ |pv$|)
  (RETURN
   (PROGN
    (LETT |dv$| (QUOTE (|Any|)) . #0=(|Any|))
    (LETT $ (make-array 35) . #0#)
    (QSETREFV $ 0 |dv$|)
```

```
    (QSETREFV $ 3 (LETT |pv$| (|buildPredVector| 0 0 NIL) . #0#))
    (|haddProp| |$ConstructorCache| (QUOTE |Any|) NIL (CONS 1 $))
    (|stuffDomainSlots| $)
    (QSETREFV $ 6
      (|Record| (|:| |dm| (|SExpression|)) (|:| |ob| (|None|))))
    (QSETREFV $ 10 (SPADCALL (QUOTE NIL) (QREFELT $ 9)))
    $))))

(setf (get (QUOTE |Any|) (QUOTE |infovec|))
 (LIST
  (QUOTE
   #(NIL NIL NIL NIL NIL NIL (QUOTE |Rep|)
     (|Boolean|) (|Reference| 7) (0 . |ref|)
     (QUOTE |printTypeInOutputP|) (|None|)
     |ANY;obj;$N;1| (|SExpression|) |ANY;dom;$Se;2|
     (|OutputForm|) |ANY;domainOf;$Of;3| (5 . =)
     |ANY;=;2$B;4| (|String|) (|Symbol|) (11 . |coerce|)
     (|List| 20) (16 . |list|) |ANY;objectOf;$Of;5|
     |ANY;showTypeInOutput;BS;6| (21 . |deref|)
     (26 . |atom?|) (|List| $) (31 . |list|)
     (36 . |coerce|) (41 . |hconcat|) |ANY;coerce;$Of;7|
     |ANY;any;SeN$;8| (|SingleInteger|)))
  (QUOTE #(~= 46 |showTypeInOutput| 52 |objectOf| 57 |obj|
           62 |latex| 67 |hash| 72 |domainOf| 77 |dom| 82
           |coerce| 87 |any| 92 = 98))
  (QUOTE NIL)
  (CONS (|makeByteWordVec2| 1 (QUOTE (0 0 0)))
   (CONS (QUOTE #(|SetCategory&| |BasicType&| NIL))
    (CONS
     (QUOTE #((|SetCategory|) (|BasicType|) (|CoercibleTo| 15)))
     (|makeByteWordVec2| 34
      (QUOTE (1 8 0 7 9 2 13 7 0 0 17 1 20 0 19 21 1 22 0 20
              23 1 8 7 0 26 1 13 7 0 27 1 20 28 0 29 1 20 15
              0 30 1 15 0 28 31 2 0 7 0 0 1 1 0 19 7 25 1 0
              15 0 24 1 0 11 0 12 1 0 19 0 1 1 0 34 0 1 1 0
              15 0 16 1 0 13 0 14 1 0 15 0 32 2 0 0 13 11 33
              2 0 7 0 0 18))))))
  (QUOTE |lookupComplete|)))

(setf (get (QUOTE |Any|) (QUOTE NILADIC)) T)
```

We can ignore this information and focus on the functions that are defined in this file. These functions can be traced with the usual common lisp tracing facility. So lets create a file /tmp/debug.lisp that contains a trace statement for each DEFUN in ANY.NRLIB/code.lsp. It looks like:

```
(trace |ANY1;retractable?;AB;1|)
(trace |ANY1;coerce;SA;2|)
(trace |ANY1;retractIfCan;AU;3|)
(trace |ANY1;retract;AS;4|)
(trace |AnyFunctions1|)
(trace |AnyFunctions1;|)
```

We can now restart the axiom system, rerun the failing expression (this will autoload ANY.o; alternatively we could hand-load the ANY.NRLIB/code.lsp file), and then load

our /tmp/debug.lisp file. Now all of the functions in the ANY domain are traced and we can watch the trace occur while the expression is evaluated. In this example I've created a larger file that traces all of the loaded domains:

```
(trace |RADUTIL;radix;FIA;1|)
(trace |RadixUtilities|)
(trace |RadixUtilities;|)

(trace |RADIX;characteristic;Nni;1|)
(trace |RADIX;differentiate;2$;2|)
(trace |RADIX;Zero;$;3|)
(trace |RADIX;One;$;4|)
(trace |RADIX;-;2$;5|)
(trace |RADIX;+;3$;6|)
(trace |RADIX;-;3$;7|)
(trace |RADIX;*;I2$;8|)
(trace |RADIX;*;3$;9|)
(trace |RADIX;/;3$;10|)
(trace |RADIX;/;2I$;11|)
(trace |RADIX;<;2$B;12|)
(trace |RADIX;=;2$B;13|)
(trace |RADIX;numer;$I;14|)
(trace |RADIX;denom;$I;15|)
(trace |RADIX;coerce;$F;16|)
(trace |RADIX;coerce;I$;17|)
(trace |RADIX;coerce;F$;18|)
(trace |RADIX;retractIfCan;$U;19|)
(trace |RADIX;retractIfCan;$U;20|)
(trace |RADIX;ceiling;$I;21|)
(trace |RADIX;floor;$I;22|)
(trace |RADIX;wholePart;$I;23|)
(trace |RADIX;fractionPart;$F;24|)
(trace |RADIX;wholeRagits;$L;25|)
(trace |RADIX;fractRagits;$S;26|)
(trace |RADIX;prefixRagits;$L;27|)
(trace |RADIX;cycleRagits;$L;28|)
(trace |RADIX;wholeRadix;L$;29|)
(trace |RADIX;fractRadix;2L$;30|)
(trace |RADIX;intToExpr|)
(trace |RADIX;exprgroup|)
(trace |RADIX;intgroup|)
(trace |RADIX;overBar|)
(trace |RADIX;coerce;$Of;35|)
(trace |RADIX;checkRagits|)
(trace |RADIX;radixInt|)
(trace |RADIX;radixFrac|)
(trace |RadixExpansion|)
(trace |RadixExpansion;|)

(trace |ANY1;retractable?;AB;1|)
(trace |ANY1;coerce;SA;2|)
(trace |ANY1;retractIfCan;AU;3|)
(trace |ANY1;retract;AS;4|)
(trace |AnyFunctions1|)
```

```
(trace |AnyFunctions1;|)

(trace |NONE1;coerce;SN;1|)
(trace |NoneFunctions1|)
(trace |NoneFunctions1;|)

(trace |ANY;obj;$N;1|)
(trace |ANY;dom;$Se;2|)
(trace |ANY;domainOf;$Of;3|)
(trace |ANY;=;2$B;4|)
(trace |ANY;objectOf;$Of;5|)
(trace |ANY;showTypeInOutput;BS;6|)
(trace |ANY;coerce;$Of;7|)
(trace |ANY;any;SeN$;8|)
(trace |Any|)
(trace |Any;|)

(trace |SExpression|)
(trace |SExpression;|)

(trace |TEX;new;$;1|)
(trace |TEX;newWithNum|)
(trace |TEX;coerce;Of$;3|)
(trace |TEX;convert;OfI$;4|)
(trace |TEX;display;$IV;5|)
(trace |TEX;display;$V;6|)
(trace |TEX;prologue;$L;7|)
(trace |TEX;tex;$L;8|)
(trace |TEX;epilogue;$L;9|)
(trace |TEX;setPrologue!;$2L;10|)
(trace |TEX;setTex!;$2L;11|)
(trace |TEX;setEpilogue!;$2L;12|)
(trace |TEX;coerce;$Of;13|)
(trace |TEX;ungroup|)
(trace |TEX;postcondition|)
(trace |TEX;stringify|)
(trace |TEX;lineConcat|)
(trace |TEX;splitLong|)
(trace |TEX;splitLong1|)
(trace |TEX;group|)
(trace |TEX;addBraces|)
(trace |TEX;addBrackets|)
(trace |TEX;parenthesize|)
(trace |TEX;precondition|)
(trace |TEX;formatSpecial|)
(trace |TEX;formatPlex|)
(trace |TEX;formatMatrix|)
(trace |TEX;formatFunction|)
(trace |TEX;formatNullary|)
(trace |TEX;formatUnary|)
(trace |TEX;formatBinary|)
(trace |TEX;formatNary|)
(trace |TEX;formatNaryNoGroup|)
(trace |TEX;formatTex|)
```

```
(trace |TexFormat|)
(trace |TexFormat;|)

(trace |CCLASS;digit;$;1|)
(trace |CCLASS;hexDigit;$;2|)
(trace |CCLASS;upperCase;$;3|)
(trace |CCLASS;lowerCase;$;4|)
(trace |CCLASS;alphabetic;$;5|)
(trace |CCLASS;alphanumeric;$;6|)
(trace |CCLASS;=;2$B;7|)
(trace |CCLASS;member?;C$B;8|)
(trace |CCLASS;union;3$;9|)
(trace |CCLASS;intersect;3$;10|)
(trace |CCLASS;difference;3$;11|)
(trace |CCLASS;complement;2$;12|)
(trace |CCLASS;convert;$S;13|)
(trace |CCLASS;convert;$L;14|)
(trace |CCLASS;charClass;S$;15|)
(trace |CCLASS;charClass;L$;16|)
(trace |CCLASS;coerce;$Of;17|)
(trace |CCLASS;#;$Nni;18|)
(trace |CCLASS;empty;$;19|)
(trace |CCLASS;brace;$;20|)
(trace |CCLASS;insert!;C2$;21|)
(trace |CCLASS;remove!;C2$;22|)
(trace |CCLASS;inspect;$C;23|)
(trace |CCLASS;extract!;$C;24|)
(trace |CCLASS;map;M2$;25|)
(trace |CCLASS;map!;M2$;26|)
(trace |CCLASS;parts;$L;27|)
(trace |CharacterClass|)
(trace |CharacterClass;|)

(trace |IBITS;minIndex;$I;1|)
(trace |IBITS;range|)
(trace |IBITS;coerce;$Of;3|)
(trace |IBITS;new;NniB$;4|)
(trace |IBITS;empty;$;5|)
(trace |IBITS;copy;2$;6|)
(trace |IBITS;#;$Nni;7|)
(trace |IBITS;=;2$B;8|)
(trace |IBITS;<;2$B;9|)
(trace |IBITS;and;3$;10|)
(trace |IBITS;or;3$;11|)
(trace |IBITS;xor;3$;12|)
(trace |IBITS;setelt;$I2B;13|)
(trace |IBITS;elt;$IB;14|)
(trace |IBITS;Not;2$;15|)
(trace |IBITS;And;3$;16|)
(trace |IBITS;Or;3$;17|)
(trace |IndexedBits|)
(trace |IndexedBits;|)

(trace |UNISEG;segment;S$;1|)
```

```
(trace |UNISEG;segment;2S$;2|)
(trace |UNISEG;BY;$I$;3|)
(trace |UNISEG;lo;$S;4|)
(trace |UNISEG;low;$S;5|)
(trace |UNISEG;hasHi;$B;6|)
(trace |UNISEG;hi;$S;7|)
(trace |UNISEG;high;$S;8|)
(trace |UNISEG;incr;$I;9|)
(trace |UNISEG;SEGMENT;S$;10|)
(trace |UNISEG;SEGMENT;2S$;11|)
(trace |UNISEG;coerce;S$;12|)
(trace |UNISEG;convert;S$;13|)
(trace |UNISEG;=;2$B;14|)
(trace |UNISEG;coerce;$Of;15|)
(trace |UNISEG;expand;$S;16|)
(trace |UNISEG;map;M$S;17|)
(trace |UNISEG;plusInc|)
(trace |UNISEG;expand;LS;19|)
(trace |UNISEG;expand;LS;19!0|)
(trace |UniversalSegment|)
(trace |UniversalSegment;|)
```

Now we rerun the function and get the trace output

```
(2) -> )lisp (load "/axiom/debug.lisp")

Value = T
(2) -> radix(10**10,32)

  1> (|RadixUtilities|)
  <1 (|RadixUtilities| #<vector 08b565cc>)
  1> (|RadixExpansion| 32)
  <1 (|RadixExpansion| #<vector 08b8cc94>)
  1> (|AnyFunctions1| #<vector 08b8cc94>)
  <1 (|AnyFunctions1| #<vector 08b5647c>)
  1> (|RadixExpansion| 32)
  <1 (|RadixExpansion| #<vector 08b8cc94>)
  1> (|RADIX;radixInt| 10000000000 32 #<vector 08b8cc94>)
  <1 (|RADIX;radixInt| (9 10 0 23 25 0 0))
  1> (|RADIX;radixFrac| 0 1 32 #<vector 08b8cc94>)
  <1 (|RADIX;radixFrac| (NIL 0))

  1> (|RadixExpansion| 32)
  <1 (|RadixExpansion| #<vector 08b8cc94>)
  1> (|RADIX;intgroup| (9 10 0 23 25 0 0) #<vector 08b8cc94>)
    2> (|RADIX;intToExpr| 9 #<vector 08b8cc94>)
    <2 (|RADIX;intToExpr| 9)
    2> (|RADIX;intToExpr| 10 #<vector 08b8cc94>)
    <2 (|RADIX;intToExpr| #\A)
    2> (|RADIX;intToExpr| 0 #<vector 08b8cc94>)
    <2 (|RADIX;intToExpr| 0)
    2> (|RADIX;intToExpr| 23 #<vector 08b8cc94>)
    <2 (|RADIX;intToExpr| #\N)
    2> (|RADIX;intToExpr| 25 #<vector 08b8cc94>)
```

```
  <2 (|RADIX;intToExpr| #\P)
  2> (|RADIX;intToExpr| 0 #<vector 08b8cc94>)
  <2 (|RADIX;intToExpr| 0)
  2> (|RADIX;intToExpr| 0 #<vector 08b8cc94>)
  <2 (|RADIX;intToExpr| 0)
<1 (|RADIX;intgroup| (CONCAT 9 #\A 0 #\N #\P 0 0))
1> (|RADIX;exprgroup|
    ((CONCAT 9 #\A 0 #\N #\P 0 0)) #<vector 08b8cc94>)
<1 (|RADIX;exprgroup| (CONCAT 9 #\A 0 #\N #\P 0 0))
 (2)  9AONP00
1> (|TexFormat|)
<1 (|TexFormat| #<vector 08b24000>)
1> (|TexFormat|)
<1 (|TexFormat| #<vector 08b24000>)
1> (|TEX;newWithNum| 2 #<vector 08b24000>)
<1 (|TEX;newWithNum| #<vector 08b8c284>)
1> (|TEX;precondition|
     (CONCAT 9 #\A 0 #\N #\P 0 0) #<vector 08b24000>)
<1 (|TEX;precondition| (CONCAT 9 #\A 0 #\N #\P 0 0))
1> (|TEX;formatTex|
     (CONCAT 9 #\A 0 #\N #\P 0 0) 0 #<vector 08b24000>)
  2> (|TEX;stringify| CONCAT #<vector 08b24000>)
  <2 (|TEX;stringify| "CONCAT")
  2> (|TEX;formatSpecial| "CONCAT"
       (9 #\A 0 #\N #\P 0 0) 0 #<vector 08b24000>)
    3> (|TEX;formatNary| ""
        (9 #\A 0 #\N #\P 0 0) 0 #<vector 08b24000>)
      4> (|TEX;formatNaryNoGroup| ""
          (9 #\A 0 #\N #\P 0 0) 0 #<vector 08b24000>)
        5> (|TEX;formatTex| 9 0 #<vector 08b24000>)
          6> (|TEX;stringify| 9 #<vector 08b24000>)
          <6 (|TEX;stringify| "9")
        <5 (|TEX;formatTex| "9")
        5> (|TEX;formatTex| #\A 0 #<vector 08b24000>)
          6> (|TEX;stringify| #\A #<vector 08b24000>)
          <6 (|TEX;stringify| "#\\A")
          6> (|IBITS;range|
              #<bit-vector 0831d930> 35 #<vector 085da658>)
          <6 (|IBITS;range| 35)
        <5 (|TEX;formatTex| "#\\A")
        5> (|TEX;formatTex| 0 0 #<vector 08b24000>)
          6> (|TEX;stringify| 0 #<vector 08b24000>)
          <6 (|TEX;stringify| "0")
        <5 (|TEX;formatTex| "0")
        5> (|TEX;formatTex| #\N 0 #<vector 08b24000>)
          6> (|TEX;stringify| #\N #<vector 08b24000>)
          <6 (|TEX;stringify| "#\\N")
          6> (|IBITS;range|
              #<bit-vector 0831d930> 35 #<vector 085da658>)
          <6 (|IBITS;range| 35)
        <5 (|TEX;formatTex| "#\\N")
        5> (|TEX;formatTex| #\P 0 #<vector 08b24000>)
          6> (|TEX;stringify| #\P #<vector 08b24000>)
          <6 (|TEX;stringify| "#\\P")
```

```
          6> (|IBITS;range|
              #<bit-vector 0831d930> 35 #<vector 085da658>)
          <6 (|IBITS;range| 35)
        <5 (|TEX;formatTex| "#\\P")
        5> (|TEX;formatTex| 0 0 #<vector 08b24000>)
          6> (|TEX;stringify| 0 #<vector 08b24000>)
          <6 (|TEX;stringify| "0")
        <5 (|TEX;formatTex| "0")
        5> (|TEX;formatTex| 0 0 #<vector 08b24000>)
          6> (|TEX;stringify| 0 #<vector 08b24000>)
          <6 (|TEX;stringify| "0")
        <5 (|TEX;formatTex| "0")
      <4 (|TEX;formatNaryNoGroup| "9#\\A0#\\N#\\P00")
      4> (|TEX;group| "9#\\A0#\\N#\\P00" #<vector 08b24000>)
      <4 (|TEX;group| "{9#\\A0#\\N#\\P00}")
    <3 (|TEX;formatNary| "{9#\\A0#\\N#\\P00}")
  <2 (|TEX;formatSpecial| "{9#\\A0#\\N#\\P00}")
<1 (|TEX;formatTex| "{9#\\A0#\\N#\\P00}")
1> (|TEX;postcondition|
    "{9#\\A0#\\N#\\P00}" #<vector 08b24000>)
  2> (|TEX;ungroup| "{9#\\A0#\\N#\\P00}" #<vector 08b24000>)
  <2 (|TEX;ungroup| "9#\\A0#\\N#\\P00")
<1 (|TEX;postcondition| "9#\\A0#\\N#\\P00")
$$
1> (|TEX;splitLong|
    "9#\\A0#\\N#\\P00" 77 #<vector 08b24000>)
  2> (|TEX;splitLong1|
      "9#\\A0#\\N#\\P00" 77 #<vector 08b24000>)
    3> (|TEX;lineConcat|
        "9#\\A0#\\N#\\P00 " NIL #<vector 08b24000>)
    <3 (|TEX;lineConcat| ("9#\\A0#\\N#\\P00 "))
  <2 (|TEX;splitLong1| ("9#\\A0#\\N#\\P00 "))
<1 (|TEX;splitLong| ("9#\\A0#\\N#\\P00 "))
9#\A0#\N#\P00
\leqno(2)
$$

                                        Type: RadixExpansion 32
```

Notice the call that reads:

```
  2> (|RADIX;intToExpr| 10 #<vector 08b8cc94>)
  <2 (|RADIX;intToExpr| #\A)
```

This means that calling —RADIX;intToExpr— with the number 10 and "the domain vector" generates the character #
A which fails. If we had the domain vector in a variable we could hand-execute this algebra function directly and watch it fail. So we go to the file RADIX.NRLIB/code.lsp which contains the definition of RADIX;intToExpr. The definition is:

```
(DEFUN |RADIX;intToExpr| (|i| $)
 (COND
  ((< |i| 10)
    (SPADCALL |i| (QREFELT $ 66)))
  ((QUOTE T)
    (SPADCALL
```

```
     (SPADCALL
       (QREFELT $ 64)
       (+ (- |i| 10) (SPADCALL (QREFELT $ 64) (QREFELT $ 68)))
       (QREFELT $ 70))
     (QREFELT $ 71)))))
```

We can put this definition into our /tmp/debug.lisp file and modify it to capture the domain vector passed in the $ variable thus:

```
(DEFUN |RADIX;intToExpr| (|i| $)
 (setq tpd $)
 (COND
  ((< |i| 10)
    (SPADCALL |i| (QREFELT $ 66)))
  ((QUOTE T)
    (SPADCALL
     (SPADCALL
       (QREFELT $ 64)
       (+ (- |i| 10) (SPADCALL (QREFELT $ 64) (QREFELT $ 68)))
       (QREFELT $ 70))
     (QREFELT $ 71)))))
```

Now when this function is executed the tpd variable will contain the value of $, the domain vector. So we load debug.lisp again to redefine RADIX;intToExpr and re-execute the function. The trace results will be the same but now the global variable tpd will have the domain vector:

```
(4) -> (identity tpd)

Value = #<vector 08b8cc94>
```

Now we can use common lisp to step the RADIX;intToExpr function:

```
(4) -> (step (|RADIX;intToExpr| 10 tpd))

Type ? and a newline for help.
  (|RADIX;intToExpr| 10 ...) ?

Stepper commands:
n (or N or Newline): advances to the next form.
s (or S):      skips the form.
p (or P):      pretty-prints the form.
f (or F) FUNCTION:   skips until the FUNCTION is called.
q (or Q):      quits.
u (or U):      goes up to the enclosing form.
e (or E) FORM:      evaluates the FORM and prints the value(s).
r (or R) FORM:      evaluates the FORM and returns the value(s).
b (or B):      prints backtrace.
?:      prints this.

  (|RADIX;intToExpr| 10 ...)
10
TPD
    = #<vector 08b8cc94>
    (SYSTEM::TRACE-CALL (QUOTE #:G1624) ...)
(QUOTE #:G1624)
SYSTEM::ARGS
```

```
      = (10 #<vector 08b8cc94>)
      (QUOTE T)
(QUOTE T)
(QUOTE (CONS # ...))
(QUOTE T)
(QUOTE (CONS # ...))
(LET (#) ...)
(QUOTE (10 #<vector 08b8cc94>))
T
      = T
  1>  (LET (#) ...)
(QUOTE (10 #<vector 08b8cc94>))
(CONS (QUOTE |RADIX;intToExpr|) ...)
(QUOTE |RADIX;intToExpr|)
SYSTEM::ARGLIST
          = (10 #<vector 08b8cc94>)
        = (|RADIX;intToExpr| 10 ...)
      = (|RADIX;intToExpr| 10 ...)
(|RADIX;intToExpr| 10 ...)
      (SETQ TPD ...)
$
        = #<vector 08b8cc94>
      = #<vector 08b8cc94>
      (COND (# #) ...)
(< |i| ...)
|i|
          = 10
          10
        = NIL
        (QUOTE T)
(SPADCALL (SPADCALL # ...) ...)
(LET (#) ...)
(QREFELT $ ...)
(SVREF $ ...)
$
               = #<vector 08b8cc94>
               71
             = (#<compiled-function |CHAR;coerce;$Of;12|> .
                 #<vector 08b3901c>)
           = (#<compiled-function |CHAR;coerce;$Of;12|> .
               #<vector 08b3901c>)
           (THE (VALUES T) ...)
(FUNCALL (CAR #:G1776) ...)
(CAR #:G1776)
#:G1776
                 = (#<compiled-function |CHAR;coerce;$Of;12|> .
                     #<vector 08b3901c>)
               = #<compiled-function |CHAR;coerce;$Of;12|>
               (SPADCALL (QREFELT $ ...) ...)
(LET (#) ...)
(QREFELT $ ...)
(SVREF $ ...)
$
                  = #<vector 08b8cc94>
```

```
                    70
                    = (#<compiled-function
                        |ISTRING;elt;$IC;30|> .
                        #<vector 08b26850>)
                    = (#<compiled-function
                        |ISTRING;elt;$IC;30|> .
                        #<vector 08b26850>)
                    (THE (VALUES T) ...)
(FUNCALL (CAR #:G1777) ...)
(CAR #:G1777)
#:G1777
                    = (#<compiled-function
                        |ISTRING;elt;$IC;30|> .
                        #<vector 08b26850>)
                    = #<compiled-function |ISTRING;elt;$IC;30|>
                    (QREFELT $ ...)
(SVREF $ ...)
$
                    = #<vector 08b8cc94>
                    64
                    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                    (+ (- |i| ...) ...)
(- |i| ...)
|i|
                    = 10
                    10
                    = 0
                    (SPADCALL (QREFELT $ ...) ...)
(LET (#) ...)
(QREFELT $ ...)
(SVREF $ ...)
$
                    = #<vector 08b8cc94>
                    68
                    = (#<compiled-function
                        |ISTRING;minIndex;$I;11|> .
                        #<vector 08b26850>)
                    = (#<compiled-function
                        |ISTRING;minIndex;$I;11|> .
                        #<vector 08b26850>)
                    (THE (VALUES T) ...)
(FUNCALL (CAR #:G1778) ...)
(CAR #:G1778)
#:G1778
                    = (#<compiled-function
                        |ISTRING;minIndex;$I;11|> .
                        #<vector 08b26850>)
                    = #<compiled-function
                        |ISTRING;minIndex;$I;11|>
                    (QREFELT $ ...)
(SVREF $ ...)
$
                    = #<vector 08b8cc94>
```

```
                    64
                    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                    (CDR #:G1778)
#:G1778
                    = (#<compiled-function
                        |ISTRING;minIndex;$I;11|> .
                        #<vector 08b26850>)
                    = #<vector 08b26850>
                    = 1
                    = 1
                    = 1
                    = 1
                    = 1
                    (CDR #:G1777)
#:G1777
                    = (#<compiled-function
                        |ISTRING;elt;$IC;30|> .
                        #<vector 08b26850>)
                    = #<vector 08b26850>
                    = 65
                    = 65
                  = 65
                = 65
                (CDR #:G1776)
#:G1776
                  = (#<compiled-function
                      |CHAR;coerce;$Of;12|> .
                      #<vector 08b3901c>)
                = #<vector 08b3901c>
              = #\A
            = #\A
          = #\A
        = #\A
      = #\A
  <1  (LET (# #) ...)
(QUOTE (10 #<vector 08b8cc94>))
(QUOTE (#\A))
(CONS (QUOTE |RADIX;intToExpr|) ...)
(QUOTE |RADIX;intToExpr|)
VALUES
          = (#\A)
        = (|RADIX;intToExpr| #\A)
      = (|RADIX;intToExpr| #\A)
(|RADIX;intToExpr| #\A)
    = #\A
  = #\A
Value = #\A
(4) ->
```

If we examine the source code for this function in int/algebra/radix.spad we find:

```
    ALPHAS : String := "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    intToExpr(i:I): OUT ==
```

```
      -- computes a digit for bases between 11 and 36
      i < 10 => i :: OUT
      elt(ALPHAS,(i-10) + minIndex(ALPHAS)) :: OUT
```

We do some lookups by hand to find out what functions are being called from the domain vectors thus:

```
(4) -> )lisp (qrefelt tpd 68)

Value = (#<compiled-function
         |ISTRING;minIndex;$I;11|> . #<vector 08b26850>)
```

The #
A value appears from a call to CHAR;coerce;$Of;12. We can look in CHAR.NRLIB/code.lsp for this function and continue our descent into the code. The function looks like:

```
(DEFUN |CHAR;coerce;$Of;12| (|c| $)
  (ELT (QREFELT $ 10)
       (+ (QREFELT $ 11) (SPADCALL |c| (QREFELT $ 21)))))
```

Again we need to get the domain vector, this time from the CHAR domain. The domain vector has all of the information about a domain including what functions are referenced and what data values are used. The QREFELT is a "quick elt" function which resolved to a highly type optimized function call. The SPADCALL function funcalls the second argument to SPADCALL with the first argument to SPADCALL effectively giving:

```
(funcall (qrefelt $ 21) |c|)
```

So we modify the CHAR;coerce;$Of;12 function to capture the domain vector thus:

```
(DEFUN |CHAR;coerce;$Of;12| (|c| $)
   (format t "|CHAR;coerce;$Of;12| called")
   (setq tpd1 $)
  (ELT (QREFELT $ 10)
       (+ (QREFELT $ 11) (SPADCALL |c| (QREFELT $ 21)))))
```

Again we rerun the failing function and now tpd1 contains the domain vector for the domain CHAR:

0.25.3 Operating system level I/O trace (strace)

If the bug seems to happen during startup the only method of debugging might be to use strace. To do this, replace the $AXIOM/bin/AXIOMsys binary with a shell script. You should:

1. rename $AXIOM/bin/AXIOMsys to $AXIOM/bin/AXIOMsys.bin
2. create the shell script shown below
3. copy the shell script to the file $AXIOM/bin/AXIOMsys so it will execute in place of the normal Axiom image
4. chmod +x $AXIOM/bin/AXIOMsys to make the script executable
5. start axiom normally

The script reads:

```
#!/bin/sh
```

```
exec strace -o /tmp/str.$$ /research/test/mnt/ubuntu/bin/AXIOMsys.bin "$@"
      | tee /tmp/tee.$$
```

The script will create 2 files in the tmp directory, "str.NNNNN" and "tee.NNNNN" where NNNNN is the process id assigned to axiom at runtime.

The tee.NNNNN file contains the console output you saw. The str.NNNNN contains the output of strace which is a list of all of the system calls and their result.

0.26 How to make graphs in algebra books

dot -Tps ¡pic ¿books/ps/domain.ps

where file pic contains something like:

```
digraph pic {
 fontsize=10;
 bgcolor="#ECEA81";
 node [shape=box, color=white, style=filled];

"OrderlyDifferentialVariable"
 [color=lightblue,href="bookvol10.3.pdf#nameddest=ODVAR"];

}
```

In book Volume 10.3 there are .ps files generated for each domain.

These pictures show the category, domain, or package that is the highest in the algebra tower. That means that the parent domain has to be compiled before this domain. Since the parent is the highest in the tower then all of the other domains will already have been compiled.

We derive the dependency information from the algebra Makefile. In order to insert algebra into the tower we derive its parents and then create a table of all of the parents. However, the table is mostly comments except for the highest parent. So if the highest parent is compiled in layer 12 then the new domain is inserted into layer 13.

The graph information at the end of a domain (e.g. UTSZ) contains only the uncommented information. It is used to generate the picture. So for UTSZ we see:

```
"UTSZ" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UTSZ"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"UTSZ" -> "ACFS"
```

This tells us several pieces of information. UTSZ depends on ACFS as the highest parent. ACFS is in layer 17 so is in layer 18. ACFS is a category (color 4488FF) and lives in bookvol10.2. UTSZ is a domain (color 88FF44, note the rotation of hex codes, a package is color FF4488) and lives in bookvol10.3.

The necessary stanzas exist after each algebra domain. Copy the stanza into a digraph block and run the dot function. So for the UTSZ domain we would create a file (e.g. called 'pic') that contains a block that reads:

```
digraph pic {
 fontsize=10;
 bgcolor="#ECEA81";
 node [shape=box, color=white, style=filled];
```

```
"UTSZ" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UTSZ"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"UTSZ" -> "ACFS"

}
```

and then run the command:

```
  dot -Tps <pic >pic.ps
  cp pic.ps ps/v103univariatetaylorseriesczero.ps
```

The output file "ps/v103univariatetaylorseriesczero.ps" is included after the pagepic of each domain in the book.

0.27 Adding or Editing pages in Hyperdoc

In Axiom it is easy to develop new pages in hyperdoc. All of the hyperdoc pages live in bookvol7.1.pamphlet.

The structure of each page, say for the Fantastic domain, consists of a few lines of latex followed by a literate chunk. All of the hyperdoc page information goes into the literate chunk and will be extracted as a hyperdoc page.

```
 \section{fantastic.ht}  <-- ordinary latex
 \pagehead{FantasticPage}{fantastic.ht}{Fantastic} <-- special latex
 \pageto{.....           <-- for latex (not hyperdoc) links to other pages
 <<fantastic.ht>>=       <-- this is what htadd looks for

   all the documentation for domain Fantastic

 @                       <-- this is the end of the page for htadd
```

When you add new pages to bookvol7.1.pamphlet you need to tell the system about the changes. The "htadd" function will search the bookvol7.1.pamphlet file for chunks with the name "*.ht" and build the file ht.db.

The ht.db file is used by hyperdoc to find pages. Each line in ht.db contains a line that

```
FantasticPage (byteIndex) (lineIndex)
```

So the two critical files, bookvol7.1.pamphlet and ht.db live in $AXIOM/doc

There is a very fast cycle for editing.

```
cd $AXIOM/doc

while (1) do
   axiom                          <-- at the shell prompt
   (navigate to page in hyperdoc) <-- to check your work
   )lisp (bye)                    <-- at the axiom prompt
   (modify bookvol7.1.pamphlet)   <-- change the page
   rm ht.db                       <-- remove the old database
   htadd bookvol7.1.pamphlet      <-- remake the database
```

0.28 Graphviz file creation

The graphviz output used on the website is a scaled vector graphics file (SVG). The dot command to output this file is:

```
  dot -Tsvg:cg <pic >pic.svg
```

The SVG file that gets generated has the following preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd" [
 <!ATTLIST svg xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink">
]>
<!-- Generated by dot version 2.8 (Thu Sep 14 20:34:11 UTC 2006)
     For user: (root) root -->
<!-- Title: AxiomSept2008 Pages: 1 -->
<svg width="3960pt" height="2312pt"
 viewBox = "0 0 3960 2312"
 xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink">
```

There are two pieces of information that are important. First, we need to add the following text by removing the trailing > character from the svg tag and replacing it with the following block. This block exports some javascript functions that we use to scale the graphics.

```
 onload="RunScript(evt)">
<script type="text/ecmascript">
<![CDATA[
var g_element;
var SVGDoc;
var SVGRoot;
function setDimension(w,h) {
  SVGDoc.documentElement.setAttribute("width",w);
  SVGDoc.documentElement.setAttribute("height",h);
}
function setScale(sw,sh) {
  g_element.setAttribute("transform","scale("+sw+" "+sh+")");
}
function RunScript(LoadEvent) {
  top.SVGsetDimension=setDimension;
  top.SVGsetScale=setScale;
  SVGDoc=LoadEvent.target.ownerDocument;
  g_element=SVGDoc.getElementById("graph0");
}
]]>
</script>
```

A second item of interest is the viewbox line which gives us the width and height information. We use this information to place the graph on the web page. A simple example of the web page looks follows. We need to replace the X and Y sizes with the sizes from the viewbox above.

```
<html>
<head>
<title>Axiom Abbreviated Category and Domain graph</title>
<script type="text/javascript">
```

```
var W3CDOM = (document.createElement && document.getElementsByTagName);
window.onload  = init;
function init(evt) {
SVGscale(0.5);
}
function SVGscale(scale) {
window.SVGsetDimension(8162*scale, 3068*scale);
window.SVGsetScale(scale,scale);
var box  = document.getElementById('svgid');
box.width   = 8162*scale;
box.height  = 3068*scale;
}
</script>

</head>
<body>
<h1>Axiom Abbreviated Category and Domain graph</h1>
<div>
choose here:
<a href="#" onclick="SVGscale(0.1);">0.1</a> or
<a href="#" onclick="SVGscale(0.25);">0.25</a> or
<a href="#" onclick="SVGscale(0.5);">0.5</a> or
<a href="#" onclick="SVGscale(1);">1.0</a> or
<a href="#" onclick="SVGscale(1.5);">1.5</a> or ...
</div>
<div>
  <object id='svgid' data="dotabb.svg" type="image/svg+xml"
   width="8162" height="3068" wmode="transparent" style="overflow:hidden;" />
  </object>
</div>

</body>
</html>
```

0.29 Adding Algebra

0.29.1 Adding algebra to the books

Assume you have a piece of algebra code that you want to permanently add to Axiom. This is a fairly complex process since the system automatically runs regression tests, creates help files, etc. and has other standard features you need to support. Lets assume your algebra looks like this:

```
)abbreviation package INTERGB InterfaceGroebnerPackage
InterfaceGroebnerPackage(K,symb,E,OV,R):Exports == Implementation where
  K    : FIELD
  symb : List Symbol
  E    : OrderedAbelianMonoidSup
  OV   : OrderedSet
  R    : PolynomialCategory(K,E,OV)

  LIST ==> List
```

```
Exports ==>  with
  groebner: LIST R -> LIST R

Implementation ==>  add
   PF ==> PrimeField(q)
   DPF ==> DistributedMultivariatePolynomial(symb,PF)
   D ==> DistributedMultivariatePolynomial(symb,K)
   JCFGBPack ==> GroebnerPackage(PF,E,OV,DPF)
   GBPack ==> GroebnerPackage(K,E,OV,D)

   coerceDtoR: D->R
   coerceDtoR(pol) == map(#1,pol)$MPolyCatFunctions2(OV,E,E,K,K,D,R)

    groebner(l)==
      ldmp:List D:= [coerceRtoD(pol) for pol in l]
      gg:=groebner(ldmp)$GBPack
      [coerceDtoR(pol) for pol in gg]
```

There are some things to check and things to change.

- remove all tabs. Spad is a language that assigns meaning to indentation and tabs are not going to survive the build process intact.
- try to stay within 80 characters. Spad code is printed in the books so it should try to limit line lengths everywhere.
- change ")abbreviation" to ")abbrev". The Makefile will search for the abbrev line and expect this exact text.

  ```
  )abbrev package INTERGB InterfaceGroebnerPackage
  ```

- make sure there is only a single space between the items in the abbrev line. The Makefile assumes this.
- Add the comment header block. The author information is used to check that all authors are included in the credits. The description tag is used as output by the ")describe" command. For example:

  ```
  ++ Author: Gaetan Hache
  ++ Date Created: September 1996
  ++ Date Last Updated: April, 2010, by Tim Daly
  ++ Description:
  ++ Part of the Package for Algebraic Function Fields in one variable PAFF
  ```

  Other tags can be included but are not used. Do not assume that any format information will be correctly translated or preserved. Make the description section be simple text with a single character between the "++" and the first word of the text.

1. Choose the right book:
   - Category goes into book bookvol10.2.pamphlet
   - Domain goes into book bookvol10.3.pamphlet
   - Package goes into book bookvol10.4.pamphlet
   - Numerics goes into book bookvol10.5.pamphlet
2. Find the right chapter. Chapter ordered by name, not abbreviation

3. Create a new section. The easiest way to do this is to copy another section and change the names. In any case, your new section will need
4. Create a dividiing line consisting of all % signs. Note that this character is the comment character for TeX.
5. Create the `\section` tag. This tag should contain the exact text of the)abbrev line from your algebra code. It reads: `)section{package INTERGB InterfaceGroebnerPackage}`
6. Create a regression test chunk. This chunk derives its name from the algebra name as in "InterfaceGroebnerPackage.input". This chunk will be automatically extracted and run during regression testing. For the moment the chunk should be a simple empty input file as in:

```
)set break resume
)sys rm -f InterfaceGroebnerPackage.output
)spool InterfaceGroebnerPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InterfaceGroebnerPackage
--E 1

)spool
)lisp (bye)
```

7. Create a help text chunk. This chunk dervies its name from the algebra name as in "InterfaceGroebnerPackage.help". This chunk will be automatically extracted and used to build help text during build. For the moment the chunck should be a simple empty help text file as in:

```
=============================================================================
 examples InterfaceGroebnerPackage
=============================================================================

See Also:
o )show InterfaceGroebnerPackage
```

At this point we need information from the Axiom interpreter to continue. Start Axiom and compile your program with "autoload" on. For example:

```
axiom -nox
)set message autoload on
)co InterfaceGroebnerPackage.spad
```

You will see output containing lines which detail the category, domains, and packages needed by your code, for example:

```
Loading /research/test/mnt/ubuntu/algebra/FIELD.o for category Field

Loading /research/test/mnt/ubuntu/algebra/EUCDOM.o for category
   EuclideanDomain
Loading /research/test/mnt/ubuntu/algebra/PID.o for category
   PrincipalIdealDomain
Loading /research/test/mnt/ubuntu/algebra/GCDDOM.o for category
```

```
   GcdDomain
Loading /research/test/mnt/ubuntu/algebra/INTDOM.o for category
   IntegralDomain
Loading /research/test/mnt/ubuntu/algebra/COMRING.o for category
   CommutativeRing
Loading /research/test/mnt/ubuntu/algebra/RING.o for category Ring
Loading /research/test/mnt/ubuntu/algebra/RNG.o for category Rng
Loading /research/test/mnt/ubuntu/algebra/ABELGRP.o for category
   AbelianGroup
Loading /research/test/mnt/ubuntu/algebra/CABMON.o for category
   CancellationAbelianMonoid
Loading /research/test/mnt/ubuntu/algebra/ABELMON.o for category
   AbelianMonoid
Loading /research/test/mnt/ubuntu/algebra/ABELSG.o for category
   AbelianSemiGroup
Loading /research/test/mnt/ubuntu/algebra/SETCAT.o for category
   SetCategory
Loading /research/test/mnt/ubuntu/algebra/BASTYPE.o for category
   BasicType
Loading /research/test/mnt/ubuntu/algebra/KOERCE.o for category
   CoercibleTo
Loading /research/test/mnt/ubuntu/algebra/SGROUP.o for category
   SemiGroup
Loading /research/test/mnt/ubuntu/algebra/MONOID.o for category
   Monoid
Loading /research/test/mnt/ubuntu/algebra/LMODULE.o for category
   LeftModule
Loading /research/test/mnt/ubuntu/algebra/BMODULE.o for category
   BiModule
Loading /research/test/mnt/ubuntu/algebra/RMODULE.o for category
   RightModule
Loading /research/test/mnt/ubuntu/algebra/ALGEBRA.o for category
   Algebra
Loading /research/test/mnt/ubuntu/algebra/MODULE.o for category
   Module
Loading /research/test/mnt/ubuntu/algebra/ENTIRER.o for category
   EntireRing
Loading /research/test/mnt/ubuntu/algebra/UFD.o for category
   UniqueFactorizationDomain
Loading /research/test/mnt/ubuntu/algebra/DIVRING.o for category
   DivisionRing
Loading /research/test/mnt/ubuntu/algebra/OAMONS.o for category
   OrderedAbelianMonoidSup
Loading /research/test/mnt/ubuntu/algebra/OCAMON.o for category
   OrderedCancellationAbelianMonoid
Loading /research/test/mnt/ubuntu/algebra/OAMON.o for category
   OrderedAbelianMonoid
Loading /research/test/mnt/ubuntu/algebra/OASGP.o for category
   OrderedAbelianSemiGroup
Loading /research/test/mnt/ubuntu/algebra/ORDSET.o for category
   OrderedSet
Loading /research/test/mnt/ubuntu/algebra/POLYCAT.o for category
   PolynomialCategory
Loading /research/test/mnt/ubuntu/algebra/PDRING.o for category
```

```
      PartialDifferentialRing
   Loading /research/test/mnt/ubuntu/algebra/FAMR.o for category
      FiniteAbelianMonoidRing
   Loading /research/test/mnt/ubuntu/algebra/AMR.o for category
      AbelianMonoidRing
   Loading /research/test/mnt/ubuntu/algebra/CHARZ.o for category
      CharacteristicZero
   Loading /research/test/mnt/ubuntu/algebra/CHARNZ.o for category
      CharacteristicNonZero
   Loading /research/test/mnt/ubuntu/algebra/FRETRCT.o for category
      FullyRetractableTo
   Loading /research/test/mnt/ubuntu/algebra/RETRACT.o for category
      RetractableTo
   Loading /research/test/mnt/ubuntu/algebra/EVALAB.o for category
      Evalable
   Loading /research/test/mnt/ubuntu/algebra/IEVALAB.o for category
      InnerEvalable
   Loading /research/test/mnt/ubuntu/algebra/FLINEXP.o for category
      FullyLinearlyExplicitRingOver
   Loading /research/test/mnt/ubuntu/algebra/LINEXP.o for category
      LinearlyExplicitRingOver
   Loading /research/test/mnt/ubuntu/algebra/KONVERT.o for category
      ConvertibleTo
   Loading /research/test/mnt/ubuntu/algebra/PATMAB.o for category
      PatternMatchable
   Loading /research/test/mnt/ubuntu/algebra/PFECAT.o for category
      PolynomialFactorizationExplicit
   Loading /research/test/mnt/ubuntu/algebra/FFIELDC.o for category
      FiniteFieldCategory
   Loading /research/test/mnt/ubuntu/algebra/FPC.o for category
      FieldOfPrimeCharacteristic
   Loading /research/test/mnt/ubuntu/algebra/FINITE.o for category
      Finite
   Loading /research/test/mnt/ubuntu/algebra/STEP.o for category
      StepThrough
   Loading /research/test/mnt/ubuntu/algebra/DIFRING.o for category
      DifferentialRing

   Loading /research/test/mnt/ubuntu/algebra/NNI.o for domain
      NonNegativeInteger
   Loading /research/test/mnt/ubuntu/algebra/INT.o for domain Integer
```

These are all of the algebra files on which your algebra depends. Collect all of these names into a list:

```
FIELD.o        EUCDOM.o       PID.o          GCDDOM.o       INTDOM.o
COMRING.o      RING.o         RNG.o          ABELGRP.o      CABMON.o
ABELMON.o      ABELSG.o       SETCAT.o       BASTYPE.o      KOERCE.o
SGROUP.o       MONOID.o       LMODULE.o      BMODULE.o      RMODULE.o
ALGEBRA.o      MODULE.o       ENTIRER.o      UFD.o          DIVRING.o
OAMONS.o       OCAMON.o       OAMON.o        OASGP.o        ORDSET.o
POLYCAT.o      PDRING.o       FAMR.o         AMR.o          CHARZ.o
CHARNZ.o       FRETRCT.o      RETRACT.o      EVALAB.o       IEVALAB.o
FLINEXP.o      LINEXP.o       KONVERT.o      PATMAB.o       PFECAT.o
FFIELDC.o      FPC.o          FINITE.o       STEP.o         DIFRING.o
```

```
NNI.o          INT.o
```

The algebra files are arranged into layers. Algebra in a given layer can only depend on algebra in lower layers. So algebra in layer 4 can only depend on algebra in layers 0 through 3.

Your algebra depends on all of the above algebra so we first need to answer the question "What is the highest layer of algebra my code depends upon".

To answer that question we need to know the layer of each of the above files. This can be determined by searching the Makefile and finding the layer. We do this here and annotate the above list, rearranged by layers:

```
FIELD.o        EUCDOM.o       PID.o          GCDDOM.o       INTDOM.o
COMRING.o      RING.o         RNG.o          ABELGRP.o      CABMON.o
ABELMON.o      ABELSG.o       SETCAT.o       BASTYPE.o      KOERCE.o
SGROUP.o       MONOID.o       LMODULE.o      BMODULE.o      RMODULE.o
ALGEBRA.o      MODULE.o       ENTIRER.o      UFD.o          DIVRING.o
OAMONS.o       OCAMON.o       OAMON.o        OASGP.o        ORDSET.o
POLYCAT.o      PDRING.o       FAMR.o         AMR.o          CHARZ.o
CHARNZ.o       FRETRCT.o      RETRACT.o      EVALAB.o       IEVALAB.o
FLINEXP.o      LINEXP.o       KONVERT.o      PATMAB.o       PFECAT.o
FFIELDC.o      FPC.o          FINITE.o       STEP.o         DIFRING.o
NNI.o          INT.o
```

We find that

```
layer 0
EUCDOM.o GCDDOM.o INTDOM.o COMRING.o RING.o RNG.o ABELGRP.o CABMON.o ABELMON.o
ABELSG.o SETCAT.o BASTYPE.o KOERCE.o MONOID.o ENTIRER.o UFD.o DIVRING.o
POLYCAT.o KONVERT.o FFIELDC.o DIFRING.o NNI.o INT.o

layer 1
SGROUP.o LMODULE.o RMODULE.o ORDSET.o RETRACT.o IEVALAB.o FINITE.o STEP.o
PATMAB.o

layer 2
BMODULE.o OASGP.o PDRING.o CHARZ.o CHARNZ.o EVALAB.o LINEXP.o

layer 3
MODULE.o OAMON.o

layer 4
ALGEBRA.o OCAMON.o

layer 5
PID.o OAMONS.o

layer 6
FIELD.o AMR.o FRETRCT.o FLINEXP.o

layer 7
FAMR.o FPC.o

layer 10
PFECAT.o
```

So we know that our algebra belongs in layer 11 since it depends on PFECAT which lives in layer 10.

Now we have all of the information needed to add the algebra to the book. We create a new section that contains the following parts:

- the `\section{...}` header
- the input file
- the help file
- the page head markup
- the pagepic markup
- the Exports section
- the theory and discussion section
- the algebra
- the dot picture file information

We've already explained most of the sections. Now we insert a chunk for our algebra. The name of this chunk is important because the Makefile will look for this exact chunk name. In our example case it looks like:

```
\begin{chunk}{package INTERGB InterfaceGroebnerPackage}
```

which is followed immediately by the algebra starting with the "abbrev" command.

Make sure that the section names, input file names, help file names, and chunk names all reflect the new algebra names.

Next we copy the chunkname to the bottom of the file as part of the algebra chunk.

The next piece to be handled is the "pagepic" tag. This inserts a picture into the book which shows what files our algebra depends upon. We do this using the graphviz dot program.

We create a file that shows the graph relationship. Since our code INTERGB depends on PFECAT we show that here. We create a line for our code that looks like:

```
"INTERGB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTERGB"]
```

This tells us the name of the node, the background color where

- #4488FF is a category
- #88FF44 is a domain
- #FF4488 is a package
- #FF8844 is a numeric

(note the rotation of bytes) and the html link from the graph to the position in the document so the user can click on the picture and go to the source code. It names the book and the offset.

We have the same information for PFECAT:

```
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
```

We combine this information to create a picture file for the graphviz dot program:

```
digraph pic {
 fontsize=10;
 bgcolor="#ECEA81";
```

```
 node [shape=box, color=white, style=filled];

"INTERGB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTERGB"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTERGB" -> "PFECAT"

}
```

Save this file into `$AXIOM/books/pic`. Then run:

```
dot -Tps <pic >ps/v104interfacegroebnerpackage.ps
```

This will generate a postscript file `$AXIOM/books/ps/v104interfacegroebnerpackage.ps`

Note that the pagepic name should be lower case as this is the name of a file in the file system. Axiom uses only lower case names as files to avoid the problem of mixed-case or ambiguous case file systems.

The pagepic file is saved into a subdirectory (ps) under the books directory. The file contains the name of the book (v104 is volume 10.4), the name of the algebra in lowercase, and the extension of ps.

Now we modify the pagepic tag in book volume 10.4 to read:

```
\pagepic{ps/v104interfacegroebnerpackage.ps}{INTERGB}{1.00}
```

This tells us where to find the file, what the abbreviation for the file will be, and the scale factor to use to display the file (1.00).

We copy the body of the pic file to the book so we can easily recreate the graphs:

```
  \begin{chunk}{INTERGB.dotabb}
  "INTERGB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTERGB"]
  "PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
  "INTERGB" -> "PFECAT"
  \end{chunk}
```

Next we return to the command line and get explicit information about our new domain:

```
(1) -> )show INTERGB
 InterfaceGroebnerPackage(K: Field,
                          symb: List Symbol,
                          E: OrderedAbelianMonoidSup,
                          OV: OrderedSet,
                          R: PolynomialCategory(K,E,OV))
             is a package constructor
 Abbreviation for InterfaceGroebnerPackage is INTERGB
 This constructor is exposed in this frame.
 Issue )edit /research/silver/PAFF/PAFF/spad/interGBwoGB.spad to see
            algebra source code for INTERGB

------------------------------ Operations --------------------------------
 groebner : List R -> List R
```

We need to insert the results of this command into the trivial input file we have created. The trivial input file will be run during build time and the results of the command at build time will be compared with the results we provide. The results we provide are actually comments so they begin with two dashes, the Axiom comment character.

The regression programm ("regress") compares each line generated by the program with the line stored in the input file. It only compares lines that start with "–R". Each test is surrounded by the comment pair "–S m of n" and "–E m" which indicates the start and end of test "m" respectively.

Since we will eventually compile this domain from the book the final output line for the)edit location will be the book. Thus we need to change the line:

```
Issue )edit /PAFF/interGBwoGB.spad to see algebra source code for INTERGB
```

to read:

```
Issue )edit bookvol10.4.pamphlet to see algebra source code for INTERGB
```

So we update our input file section to read:

```
)set break resume
)sys rm -f InterfaceGroebnerPackage.output
)spool InterfaceGroebnerPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InterfaceGroebnerPackage
--R InterfaceGroebnerPackage(K: Field,
--R                           symb: List Symbol,
--R                           E: OrderedAbelianMonoidSup,
--R                           OV: OrderedSet,
--R                           R: PolynomialCategory(K,E,OV))
--R    is a package constructor
--R Abbreviation for InterfaceGroebnerPackage is INTERGB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTERGB
--R
--R------------------------------ Operations --------------------------------
--R groebner : List R -> List R
--R
--E 1

)spool
)lisp (bye)
```

There is another section, Exports, which lists all of the unique function names exported by this algebra. Note that this list is longer than just the functions locally defined because the algebra inherits functions. We make a tabular environment with the number of columns specified by the "l" characters ("l" means left-justify, "c" would be center, and "r" would be right justify). We include just enough columns to keep from overflowing the 80 column limit.

For this domain we see generate we only have a single export.

```
{\bf Exports:}\\
\cross{INTERGB}{groebner}
```

The cross function creates a cross-reference entry in the index so you can look up the function and find the domain or look up the domain and find the function.

Now we have finished setting up the algebra.

We have to add the algebra, regression, and help information to the Makefile.

We know from above that the algebra belongs in layer 11 so we find layer 11 and add this abbreviation in alphabetical order. We can use this order since none of the files depend on each other in this layer. The new line looks like:

```
  ${OUT}/INMODGCD.o ${OUT}/INNMFACT.o ${OUT}/INPSIGN.o  ${OUT}/INTERGB.o  \
```

Note that the trailing slash is required by the Makefile in order to continue the input line.

Take note of the prior algebra abbreviation INPSIGN. We search for that lower down and find the block reading:

```
"INPSIGN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INPSIGN"]
/*"INPSIGN" -> {"RING"; "RNG"; "ABELGRP"; "CABMON"; "ABELMON"; "ABELSG"}*/
/*"INPSIGN" -> {"SETCAT"; "BASTYPE"; "KOERCE"; "SGROUP"; "MONOID"}*/
/*"INPSIGN" -> {"LMODULE"; "UPOLYC"; "POLYCAT"; "PDRING"; "FAMR"; "AMR"}*/
/*"INPSIGN" -> {"BMODULE"; "RMODULE"; "COMRING"; "ALGEBRA"; "MODULE"}*/
/*"INPSIGN" -> {"CHARZ"; "CHARNZ"; "INTDOM"; "ENTIRER"; "FRETRCT"}*/
/*"INPSIGN" -> {"RETRACT"; "EVALAB"; "IEVALAB"; "FLINEXP"; "LINEXP"}*/
/*"INPSIGN" -> {"ORDSET"; "KONVERT"; "PATMAB"; "GCDDOM"}*/
"INPSIGN" -> "PFECAT"
/*"INPSIGN" -> {"UFD"; "ELTAB"; "DIFRING"; "DIFEXT"; "STEP"; "EUCDOM"}*/
/*"INPSIGN" -> {"PID"; "FIELD"; "DIVRING"; "INT"; "INS-"}*/
```

This information is used to create the total graph of the algebra. Most of this information is kept for future graph use but commented out. We only uncomment lines that are direct dependence relationships. As you can see from the above INPSIGN depends only on PFECAT.

First we create the signature line which is the same one used above for the pagepic graph.

We also need to create a similar block for our code. We use the list generated above, replacing the ".o" with semicolons.

The total result is:

```
"INTERGB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTERGB"]
/*"INTERGB" -> {
/*"INTERGB" -> {"FIELD"; "EUCDOM"; "PID"; "GCDDOM"; "INTDOM"; "COMRING"}*/
/*"INTERGB" -> {"RING"; "RNG"; "ABELGRP"; "CABMON"; "ABELMON"; "ABELSG"}*/
/*"INTERGB" -> {"SETCAT"; "BASTYPE"; "KOERCE"; "SGROUP"; "MONOID"}*/
/*"INTERGB" -> {"LMODULE"; "BMODULE"; "RMODULE"; "ALGEBRA"; "MODULE"}*/
/*"INTERGB" -> {"ENTIRER"; "UFD"; "DIVRING"; "OAMONS"; "OCAMON"; "OAMON"}*/
/*"INTERGB" -> {"OASGP"; "ORDSET"; "POLYCAT"; "PDRING"; "FAMR"; "AMR"}*/
/*"INTERGB" -> {"CHARZ"; "CHARNZ"; "FRETRCT"; "RETRACT"; "EVALAB"}*/
/*"INTERGB" -> {"IEVALAB"; "FLINEXP"; "LINEXP"; "KONVERT"; "PATMAB"}*/
"INTERGB" -> "PFECAT"
/*"INTERGB" -> {"FFIELDC"; "FPC"; "FINITE"; "STEP"; "DIFRING"; "NNI"; "INT"}*/
```

Help files are automatically extracted from the books using the lisp function "makeHelpFiles" which lives in books/tangle.lisp. This will find all of the .help chunks in books of interest and write each chunk to the target directory in its own filename. So if a chunk name is somedomain.help we create the help file somedomain.help containing the chunk value.

Now we create the regression hook. Look for the list REGRESS and add the line containing the algebra thus:

```
 InterfaceGroebnerPackage.regress \
```

This will cause regression to be run on the InterfaceGroebnerPackage.input file.

We need to modify book volume 5 (the interpreter) to add this algebra to the list of exposed algebra. To do this, add the line:

```
(|InterfaceGroebnerPackage| . INTERGB)
```

to the variable `$globalExposureGroupAlist` under the "basic" sublist.

Now the algebra should build and have the new algebra available. There are ways this can fail which we will cover in more detail.

Once the code compiles cleanly there are a few ways to check that it works. First, check that the file int/input/InterfaceGroebnerPackage.regress shows no failures. Second, run the command

```
)show InterfaceGroebnerPackage
```

to see if the domain exists. Run the command

```
)help InterfaceGroebnerPackage
```

to see that the help file exists. Run the command

```
)describe InterfaceGroebnerPackage
```

to see that the description paragraph exists.

0.29.2 Creating a stand-alone pamphlet file

Suppose we want to add a new algebra file from a new pamphlet. For example, we want to add BLAS1.spad in books/newbook.pamphlet. The explanation for the steps follow. The steps are:

1. write the algebra code
2. create the pamphlet file called newbook.pamphlet
3. create a new section in newbook.pamphlet
4. create the chunk name stanza in newbook.pamphlet
5. insert BLAS1.spad into the new stanza
6. create the dotabb stanza in newbook.pamphlet
7. update src/algebra/Makefile.pamphlet to insert BLAS1.spad
8. update src/algebra/Makefile.pamphlet to insert BLAS1 into graph
9. update the dotabb stanza in newbook.pamphlet
10. update src/Makefile.pamphlet to copy the newbook file to src/algebra
11. update $globalExposureGroupAlist in bookvol5 to add BLAS1 to basic

0.30 Makefile

This book is actually a literate program[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.

Bibliography

[Bake14] Martin Baker. Axiom Architecture, 2014.

Link: http://www.euclideanspace.com/prog/scratchpad/internals/ccode

[Blac80] A.P. Black. Exception Handling and Data Abstraction. Research Report RC8059, IBM Research, 1980.

[Cohe80] J.D. Cohen and R.D. Jenks. On Resolution and Coercion in MODLISP, 1980.

Comment: in preparation

[Dave80a] James H. Davenport and Richard D. Jenks. MODLISP. Research Report RC 8537 (#37198), IBM Research, 1980.

Comment: http://www.computerhistory.org/collections/catalog/102719109

[Dona77] J. Donahue. On the semantics of "Data Type", 1977.

Comment: Cornell University

[Ehri80a] Hartmut Ehrig, Hans-Jorg Kreowski, James Thatcher, Eric Wagner, and Jesse Wright. Parameterized Data Types in Algebraic Specification Languages. *LNCS*, 85, 1980.

[Ersh77] A.P. Ershov. On the Essence of Compilation, 1977.

Comment: Proc. IFIP Working Conf. on Formal Description of Programming Concepts, Vol. 1

[Gogu76] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. Research Report RC-6487, IBM Research, 1976.

[Hear74] Anthony C. Hearn. A Mode Analysing Algebraic Manipulation Program. In *Proc. 1974 annual ACM Conference. Vol 2.* ACM, 1974.

[Hend76] Peter Henderson and James H. Morris Jr. A Lazy Evaluator. In *3rd Symp. on Principles of Programming Languages*, pages 95–103. ACM, 1976.

Abstract: A different way to execute pure LISP programs is presented. It delays the evaluation of parameters and list structures without ever having to perform more evaluation steps than the usual method. Although the central idea can be found in earlier work this paper is of interest since it treats a rather well-known language and works out an algorithm which avoids full substitution. A partial correctness proof using Scott-Strachey semantics is sketched in a later

section.

[IBMx78] IBM. *LISP/370 Program Description / Operations Manual.* IBM Research, 1978.

Comment: SH20-2076-0

[Jenk77] Richard D. Jenks. On the Design of a Mode-Based Symbolic System. *SIGGAM Bulletin*, 11(1):16–19, 1977.

Abstract: This paper is a preliminary report on the design and implementation of a mode-based symbolic programming system and compiler which allows programming with rewrite rules and LET and IS pattern-match constructs. An important feature of this design is the provision for mode-valued variables which allow algebraic domains to be run-time parameters.

[Jenk79] Richard D. Jenks. MODLISP: An Introduction. In *Proc. ISSAC 1979*, EUROSAM 79, pages 466–480. Springer-Verlag, 1979, 3-540-09519-5.

Comment: IBM Research Report RC 8073 Jan 1980

[Jens75] Kathleen Jensen and Niklaus Wirth. *PASCAL User Manual and Report.* Springer-Verlag, 1975, 0-387-90144-2.

[Knut92] Donald E. Knuth. *Literate Programming.* Center for the Study of Language and Information, Stanford CA, 1992, 0-937073-81-4.

[Lisk77] Barbara Liskov, Alan Synder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *CACM*, 20(8), 1977.

Abstract: CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions – procedural, control, and especially data abstractions – are useful in the programming process. Of these, only the procedural abstraction is supported well by conventional languages, through the procedure or subroutine. CLU provides, in addition to procedures, novel linguistic mechanisms that support the use of data and control abstractions. This paper provides an introduction to the abstraction mechanisms of CLU. By means of programming examples, the utility of the three kinds of abstractions in program construction is illustrated, and it is shown how CLU programs may be written to use and implement abstractions. The CLU library, which permits incremental program development with complete type checking performed at compile time, is also discussed.

Link: https://www.cs.virginia.edu/~weimer/615/reading/liskov-clu-abstraction.pdf

[Lisk77a] Barbara Liskov and Stephen Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, 9(4):50–59, 1977.

Abstract: The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the re-

sulting program will be more reliable as a result. Clearly, this is a worthwhile goal. Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient. This paper presents an approach which allows the set of built-in abstractions to be augmented when the need for a new data abstraction is discovered. This approach to the handling of abstraction is an outgrowth of work on designing a language for structured programming. Relevant aspects of this language are described, and examples of the use and definitions of abstraction are given.

[Lisk79] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU Reference Manual. Technical report, Massachusetts Institute of Technology, 1979.

[Loos74] Ruediger G. K. Loos. Toward a Formal Implementation of Computer Algebra. *SIGSAM*, 8(3):9–16, 1974.

Abstract: We consider in this paper the task of synthesizing an algebraic system. Today the task is significantly simpler than in the pioneer days of symbol manipulation, mainly because of the work done by the pioneers in our area, but also because of the progress in other areas of Computer Science. There is now a considerable collection of algebraic algorithms at hand and a much better understanding of data structures and programming constructs than only a few years ago.

[Morr73] J.H. Morris Jr. Types are not Sets. In *Symp. on the Principles of Programming Languages*, pages 120–124. ACM, 1973.