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 3: Axiom Programmers Guide

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

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

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

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))

Chapter 1

A Language for Computational Algebra by Jenks and Trager

1.1 Introduction

Jenks and Trager[Jenk81] describe a language with parameterized types and generic operators particularly suited to computational algebra. A flexible framework is given for building algebraic structures and defining algorithms which work in as general a setting as possible. This section will be an overview of our main concepts: "domains" and "categories".

A language for computational algebra should be able to express algorithms for dealing with algebraic objects at their most natural level of abstraction. We can illustrate this concept with two simple algorithms. First, we wish to write a function *max* which computes the maximum of two elements of any set on which an ordering predicate is defined. One approach to this problem is to explicitly pass the ordering predicate as an additional argument to max. Thus max might be defined by

```
max(x,y,lessThan) == if lessThan(x,y) then y else x
```

where lessThan is the ordering predicate. In more complicated algorithms, the number of additional arguments required gets out of hand. Our approach is instead to require the arguments x and y of max to be elements of some specific algebraic structure which has a "less than" operation '<' implemented by some function. We will call such algebraic structures **domains**. Thus '<' is a "generic" operation which has different function definitions for different domains. Our definition of max, with suitable declarations, becomes:

```
max(x,y) == if x < y then y else x
```

The requirement that a generic operation have a particular name does not characterize its algebraic properties. In the above definition of max, it is implicitly assumed that '<' provides a total ordering on the elements of its domain. To this end, our domains will also have a set of *attributes* which permit a description of the algebraic properties of its operations (e.g. so as to distinguish between totally-ordered and partially-ordered sets).

As a second example, we examine the classical algorithm for computing the *gcd*, the greatest common divisor, of two integers:

```
gcd(x,y) == if x = 0 then y else gcd(y,remainder(x,y))
```

Although this algorithm was originally intended to be used only on integers, a cursory examination shows that only a few properties of the integers are actually required. In fact, the same algorithm can be used on gaussian integers, polynomials over fields, or any other domain which has an appropriate remainder function. We wish to specify the minimum requirements of an algebraic structure for the gcd algorithm to be applicable. To do this, we introduce a grouping of domains called a **category**, in this case, "the category of Euclidean domains". Any domain of this category will be an integral domain with a generic function *remainder* satisfying two requirements. The first is that

```
remainder(x,y) = x - q*x*y
```

for some q in the domain as this implies

```
gcd(x,y) - gcd(y,remainder(x,y))
```

The remainder function must also have the property that the remainder sequence generated by any two elements of the domain always reaches 0 in a finite number of steps. These two requirements are sufficient to guarantee the correctness of our gcd algorithm.

Categories provide a set of required generic operations together with a set of attributes describing the required algebraic properties of these operations. Domains provide specific functions which implement the operations and satisfy the attributes. Thus we may speak of "the category of totally-ordered sets" as the class of all domains which have the above '<' operation with specific algebraic properties, and "the domain of the integers" as an example of a member of that category since it has a function 'integer<' which provides that operation and satisfies those properties.

Once an abstract algorithm has been written, its author specifies the category of domains to which it applies, either by explicitly listing the operations and attributes it requires or by referencing a predefined category. For example, having defined the category EuclideanDomain ("the category of Euclidean domains"), a complete definition of the gcd function could be written:

```
gcd(x:R,y:R):R where R:EuclideanDomain ==
    if x=0 then y else gcd(y,remainder(x,y))
```

Expressions of the form "A:B" are called *declarations*. The arguments x and y of gcd are *declared* to be elements of some domain R which, in turn, is declared to be a member of EuclideanDomain. Declaring "A:B" means "A is a member of B" in the following sense. Declaring a domain to be a member of a category indicates that all the operations of the category are implemented in that domain as functions which satisfy the attributes of the category. Similarly declaring an object to be a member of a domain means that all the functions provided in that domain are applicable to the object.

Domains and categories are both computed objects that can be assigned to a variable, passed as arguments, and returned from functions. A category may be produced by explicitly listing operations and attributes, or by invoking a function which returns a category. Categories may be augmented, diminished, or "joined" with other categories to produce a new category containing all of the operations and attributes of the individual categories.

A domain is always created by a function which we call a **functor**. Some simple domains are "the integers" and "the booleans" which are produced by functors of no arguments. Other domains such as "the integers mod 7" are produced by functors which take arguments (such as the modulus 7). Most algebraic domains are built up from other domains which, along with other parameters, are passed as arguments to the functors that construct them. For example, the domain "polynomials in X over the integers" is created by a polynomial functor which takes a variable (e.g. "X") and an underlying domain (e.g. "the integers") as arguments. With the few exceptions noted in the next section, functors and categories are definable in the language and may be freely modified. A user is free to introduce new categories and define new functors in order to make more domains available for computation.

Max and gcd were both defined in terms of generic operations from domains implicitly passed as arguments. As required by some algebraic algorithms, domains are dynamically created and assigned to local variables. Objects of these newly created local domains can be created, manipulated, and converted to objects of other domains.

Figure 1: Algebraic Categories

category	**extends**	**operations**
Set		$=$
AbelianGroup	Set	$0, +, -$
OrderedSet	Set	$<$
QuotientObject(S:Set)	Set	reduce,lift
SemiGroup	Set	$\times$
Finite	Set	size,random
Monoid	SemiGroup	1
Group	Monoid	inv
Ring	(Monoid,AG)	characteristic, recip
Module(R:Ring)	AbelianGroup	scalar$\times$
Algebra(R:Ring)	(Ring,Module(R))	
DifferentialRing	Ring	deriv
IntegralDomain	Ring	isAssociate./ /
SkewField	Ring	/
UniqueFactorizationDomain	IntegralDomain	gcd, factor, isPrime
EuclideanDomain	UFD	size¡, quo, rem
Field	(ED, SkewField)	
GaloisField	(Field, Finite)	
VectorSpace(S:Field)	Module(S)	

Both categories and domains may be organized into hierarchies. Figure 1 shows an algebraic hierarchy of categories, listing the operations (but not the attributes) introduced by the successive categories. Set is a category with a single operation '='. SemiGroup extends Set by adding an operation '×' etc. More complicated cases will be discussed in the next section. We will also allow one domain to extend another.

For example, one can write a "free-module functor" to provide the module-theoretic aspects of a polynomial ring (additon and multiplication by scalars). One can then write various polynomial, algebraic-extension, and sparse-matrix functors as extensions of the free-module functor. The polynomial functor, for example, would augment the functions provided by the free-module functor, adding explicit definitions only for the other polynomal functions such as multiplication. Similarly, a "localization functor" can be written to provide computations where denominators may be from a different domain than that of the numerator, such as "the odd integers", "powers of 2", "products of factored polynomials in X". The localization functor is thus a function of two arguments, one for the numerator domain, the other for the denominator domain. A "quotient field functor" can then be written which extends the localization functor for the special case when the two argument domains are the same integral domain. From the quotient field functor, one can produce all of the rational function domains and "the rational numbers" as special cases.

To summarize, our language design provides the useful notions of "domains" and "categories" for the abstract description of algorithms for computational algebra. The facility for categories is unique to our language and its use seems to be invaluable for describing algorithms for computational algebra. Our domains are similar to "modes" in EL1[Wegb74] and "types" in RUSSELL[Deme79] and ADAPT[Leve80]. As in EL1, but in contrast to Ada[ADAx83], domains are computed values. Our notions of categories and functors are based on concepts in universal algebra [Cohn65] developed by the ADJ group [That82] and Burstall and Goguen [Burs77]. In addition, categories extend the idea of "type constraint" in CLU [Lisk79] and Alphard [Wulf76] where functions can require that their arguments have certain operations available to them. For related work in computer algebra, see [Ausi79], [Gris76], [Jenk74], and [Loos74].

1.2 Concepts

In this section, we give precise definitions and examples of the concepts of domain, category, and functor.

Domain. By a *domain of computation*, or, simply, **domain**, we mean:

1. a set of generic **operations**
2. a **representation**
3. a set of **functions** which implement the operations in terms of the representation
4. a set of **attributes**, which designate useful facts such as axioms and mathematical theorems which are true for the operations as implemented by the functions.

The simplest examples of domains are those corresponding to the basic data-types offered by the underlying system, such as Integer ("the integers"), Boolean ("the booleans"), etc. Other examples of domains are RationalNumber ("the rational numbers"), Matrix(Integer) ("rectangular matrices with integer coefficients"), and Polynomial(X,RationalNumber) ("the polynomials in X with rational number coefficients").

The generic operations are given by **signatures**, expressions consisting of an *operation name*, a *source*, and a *target*. The domain Integer, for example, has the operation "less than" expressed by the signture:

$$`<` : (\text{Integer}, \text{Integer}) \rightarrow \text{Boolean}$$

with '<' as operation name, (Integer,Integer) as source, and Boolean as target. The source part of the signature is any sequence of domains, and the target part is any domain.

The representation for a domain describes a data structure used to represent the objects of the domain.

The functions component is a set of compiled functions providing a domain-specific implementation for each generic operation. For example, domain Integer has a function "Integer<" which implements "less than". If a domain has an operation signature

$$\text{op} : (D_1, \ldots, Dn) \rightarrow D_0$$

then the associated function must take arguments from the representations of $D_1, \ldots, D_n$ respectively and return a result in the representation of D_0.

The attribute component of a domain is described either by a name, e.g. "finite", or by a form with operator names as parameters, e.g. "distributive('×','+')". The purpose of attributes is not to provide complete axiomatic descriptions of an operation, rather to assert facts which programs can query.

Category A "category" designates a class of domains with common operations and attributes but with different functions and representations. The categories of interest here will be those of algebraic structures such as Ring ("the class of all rings"), Field ("the class of all fields"), and Set ("the class of all sets").

By a **category** we mean:

1. a set of generic **operations**
2. a set of **attributes**, which designate facts which must be true for any implementation of the operations.

As with domains, the generic operations of categories are given by signatures, consisting of an operation name, a source, and a target. In addition to the domains which may appear in the source and target, a special symbol \$ (or, later %) is used to designate an arbitrary member domain of the category. The set of operations and attributes are those which member domains have in common. A simple example of a category is Set, a category which has one operation

$$=: (\$, \$) \to Boolean$$

and no attributes. Another is SemiGroup, which besides the '=' operation, has the operation

$$\times : (\$, \$) \to \$$$

and the attribute associative('×').

We say a domain D is "a member of" a category C, equivalently, D is of C, if D contains every operation and attribute of C with \$ replaced by D. For example, Integer is of Set because it contains an operation

$$=: (\text{Integer}, \text{Integer}) \to \text{Boolean}$$

We say that a category B extends a category A if all of the operations and attributes of A are contained in B. SemiGroup extends Set since all of the operations ('=') of Set are contained in SemiGroup.

Figure 2. Examples of Category Definitions

Set: Category == **category**
    [*operations*] =:(\$,\$) → Boolean

SemiGroup: Category == Set with
    [*operations*] ×:(\$,\$) → \$
    [*attributes*] associative('×') [$(x \times y) \times z = x \times (y \times z)$]

Figure 2 illustrate our language for defining categories. Set is defined by explicitly listing its operations ('=') and its attributes (none). SemiGroup is defined as an extension of Set. Square-bracketed expressions are comments. The '==' signifies a rewrite-rule definition for the category Set. Evaluation of "Set" causes "Set" to be rewritten by the category value indicated to the right of the '=='. Evaluation of "SemiGroup" similary causes "SemiGroup" to be rewritten by the corresponding right-hand expression. Further evaluation causes Set to be replaced by its value, a category to which the '×' operation and associative('×') attribute are added by the **with** operation. As implied by this evaluation mechanism, two categories are equivalent iff they have equivalent sets of operations and attributes, irrespecitive of how they were created.

Figure 3. Examples of Category Definitions

Module(R:Ring) : Category == AbelianGroup with
[*operations*] ×:(R,$) → $
[*attributes*] ...

Algebra(R:Ring) : Category == (Ring,Module(R)) with
[*attributes*] ...

Figure 3 gives two examples of parameterized categories, that is, categories that are produced by functions of one or more arguments. The function Module creates the category of all R-modules, that is, modules over a given ring R. For example, the function Module applied to Integer produces the category of all Z-modules, domains D which are abelian groups with the additional operations

$$\times : (\mathrm{Integer}, \mathrm{D}) \rightarrow \mathrm{Integer}$$

.

This category includes domain Integer since Integer is an abelian group and has the operation

$$\times : (\mathrm{Integer}, \mathrm{Integer}) \rightarrow \mathrm{Integer}$$

.

The function Algebra(R) extends the *join* of a Ring and a Module(R), written (Ring,Module(R)). The join designates the category formed by directly combining the operations and attributes of Ring with those of Module(R).

Another way of parameterizing categories is by operator names. For example, the above definition of SemiGroup could be extended to take a binary operation as a parameter:

SemiGroup(op) : Category == Set with
[*operations*] op: ($,$) → $
[*attributes*] associative(op) [$op(op(x, y), z) = op(x, op(y, z))$]

after which we may refer to the multiplicative form of SemiGroup in Figure 2 by SemiGroup('×').

Functor By a **functor** we mean any function which returns a domain. A functor creates a domain, a member of some category. A category never creates anything: it simply acts as a template for domains, describing which operations and attributes must be present. A functor creates a domain by storing functions into a template given by its target category. Categories never specify representations for objects; functors always do.

Domains can only be produced by functors. Basic domains (e.g. "the integers") are produced through functors bound to identifiers (e.g. Integer). In addition, four built-in functors, List Vector, Struct, and Union, build aggregate domains from other domains passed as arguments. The functor List can be applied to any domain (e.g. Integer) to produce a composite domain (e.g. List(Integer)) with a set of functions which provide operations on lists (e.g. first, rest, cons). The functor Vector takes two arguments, a positive integer n and a domain D, and produces the domain "the set of all vectors of length n with elements from D". Struct produces a domain represented by a set of name-value pairs, e.g.

Struct(real:Integer,imag:Integer) describes an appropriate representation for "the Gaussian integers". Union(A,. . .,B) creates a new domain D form the domains $A, \ldots, B$ which is the disjoint union of the domains $A, \ldots, B$.

The language permits the building of new functors from these basic functors. A simple example is FiniteField in Figure 4.

Figure 4. Example of Functor Definition

```
FiniteField(p:PrimeNumber) : GaloisField ==
    capsule
      [representation]
        Rep := Integer
      [declarations]
        x,y : $
      [definitions]
        0 == Integer.0
        1 == Integer.1
        x+y == if (w ← x Integer.+ y) > p
                 then w - p
                 else w
        . . .
```

The functor FiniteField applied to p, a prime number, creates a domain "the integers modulo p", a member of its target category GaloisField ("the class of all Galois fields"). The set of operations and attributes of this domain are given by GaloisField, the representation and set of functions, by the *capsule* part of the definition which appears to the right of the '=='. The representation is always defined by the distinguished symbol Rep in terms of a "lower level" functor. For FiniteFields, Rep is defined to be Integer (meaning that elements of a finite field are represented by integers). In a more complicated example, Rep might be defined in terms of a functor Matrix, whose Rep, in turn, might be defined in terms of the built-in functor Vector.

Figure 5. Example of Functor Definition

```
IntegerMod(m:Integer — m > 1): T == C where
    T == (Ring,Finite) with
              if isPrime m then GaloisField
    C == capsule
        . . .
      [definitions]
        if isPrime m then
          x / y == . . .
        . . .
```

Figure 5 illustrates the use of conditional expressions to make the target category of a functor depend upon the paramters of the functor. Here the FiniteField functor of Figure 4 is generalized to IntegerMod, a functor which produces the domain "the integers modulo m" for any positive integer (modulus) m. The domain produced by the IntegerMod functor will be a Galois field if m is prime, a finite ring, if it is not. Conditional expressions are also used in the capsule part of a functor to conditionally provide functions (e.g the operation '/' will be provided by IntegerMod only if m is prime), or to provide alternative versions of functions (e.g. more efficient implementations for some functions when the modulus is small).

Figure 6 illustrates a series of functors for localization which illustrate how domains, like categories, can be extended. Localize takes an R-module M, and a denominator domain D which is a monoid contained within R. It produces an R-module of "fractions". LocalAlgebra augments this with a definition of multiplication for fractions producing the localization of an R-algebra. QuotientField uses LocalAlgebra to produce a "field of fractions" in the special situation where the numerators and denominators both come from the same integral domain. When R has a gcd function, QuotientField redefines the arithmetic operations (supplied by LocalAlgebra) to produce reduced fractions. Similarly, if R has a derivation defined for it, QuotientField extends this derivation to the field of fractions.

Conclusions Our language provides the useful notions of "domains" and "categories" for the abstract description of algorithms for computational algebra. Domains are the algebraic structures on which computation is performed. Categories are groupings of domains with common operations and attributes.

There are several advantages to our design. Algorithms can be written to operate over any group, ring, or field, independently of how that algebraic structure is defined or represented in the computer. The algorithm implementor need not know about which domains have actually been created. Rather they need only specify a category which gives the required operations and essential algebraic properties of the algorithm. Also, as required by many algebraic algorithms, domains and categories are dynamically computed objects.

The language we have presented leads to a computer algebra system which is easily extended by any user. All categories are defined in the language and are available for user modification. All domains are created by functions which, with the exception of a few that are built-in, are also defined in the language and can be changed by the user. New domains and categories can be designed and implemented with minimal effort by extending or combining existing structures.

The language permits considerable code economy. An algorithm is implemented by a single function which is applicable to any domain of a declared category. A matrix functor, for example, will use the same compiled function to compute the product of two matrices, regardless of whether the actual matrix coefficents are integers, polynomials, or other matrices. Parameterized functors help to minimize redundant code by providing a set of pre-compiled functions for all domains they can produce. The universal applicability of such functors as QuotientField provide powerful methods for constructing new algebraic objects.

Our primary goal in presenting a language which deals with algebraic objects was to take advantage of as much of the structure implicit in the problem domain as possible. The natural algebraic notions of domains extending one another, and collecting domains with common properties into categories have been show to be useful computational devices. By preserving this natural structure, we hope to have eased the task of finding computational models for algebraic structures.

Figure 6. Definition of Localization Functors

```
Localize(isZeroDivisor,M,D) : Module(R) == C where
    R:Ring
    M: Module(R)
    D: Monoid | D ⊆ R
    isZeroDivisor: M → Boolean
    C == capsule ...
            [representation]
                Rep := Struct(num:M, den:D)
            [declarations]
                x,y: $
                n: Integer
                r:R ; d:D
            [definitions]
                0 == Rep(0,1)
                -x == Rep(-x.num,x.den)
                x=y == isZeroDivisor(y.den × x.num - x.den × y.num)
                x+y == Rep(y.den × x.num + x.den × y.num, x.den × y.den)
                n × x == Rep(n × x.num,x.den)
                r × x == if r=x.den then Rep(x.num,1) else Rep(r × x.num,x.den)
                x / d == Rep(x.num,d × x.den)

LocalAlgebra(isZeroDivisor,A,D): T == C where
    R: Ring
    A: Algebra(R)
    isZeroDivisor: A → Boolean
    D: Monoid | D ⊆ R
    T == Algebra(R) with if A has commutative('×') then commutative('×')
    C == Localize(isZeroDivisor,A,D) add ...
                1 == Rep(1,1)
                x × y == Rep(x.num × y.num, x.den × y.den)
                characteristic == A.characteristic

QuotientField(R; IntegralDomain) : T == C where
    T == (Field,Algebra(R)) with if R of DifferentialRing then DifferentialRing
    C == LocalAlgebra($1 = 0,R,R) add ...
                if R has gcd: (R,R) → R then
                    x + y == ...
                    x × y == ...
                    where cancelGcd(x:$):$ == ...
                if R of DifferentialRing then
                    if R has gcd: (R,R) → R
                        then deriv(x) == ...
                        else deriv(x) == ...

RationalFunction(x:Expression, R:Ring) == QuotientField(Polynomial(x,R))

RationalNumber == QuotientField(Integer)
```

Chapter 2

Details for Programmers

Axiom maintains internal representations for domains. There are functions for examining the internals of objects of a particular domain.

2.1 Examining Internals

One useful function is **devaluate** which takes an object and returns a Lisp pair. The CAR of the pair is the Axiom type. The CDR of the pair is the object representation. For instances, consider the session where we create a list of objects using the domain **List(Any)**.

```
(1) -> w:=[1,7.2,"luanne",3*x^2+5,_
          (3*x^2+5)::FRAC(POLY(INT)),_
          (3*x^2+5)::POLY(FRAC(INT)),_
          (3*x^2+5)::EXPR(INT)]$LIST(ANY)

                               2        2        2        2
   (1)  [1,7.2,"luanne",3x  + 5,3x  + 5,3x  + 5,3x  + 5]
                                                        Type: List(Any)
```

The first object, **1** is a primitive object that has the domain **PI** and uses the underlying Lisp representation for the number.

```
(2) -> devaluate(1)$Lisp

   (2)  1
                                                      Type: SExpression
```

The second object, **7.2** is a primitive object that has the domain **FLOAT** and uses the underlying Lisp representation for the number, in this case, itself a pair whose CAR is the floating point base and whose CDR is the mantissa,

```
(3) -> devaluate(7.2)$Lisp

   (3)  (265633114661417543270 . - 65)
                                                      Type: SExpression
```

The third object, **"luanne"** is from the domain **STRING** and uses the Lisp string representation.

```
(4) -> devaluate("luanne")$Lisp

   (4)  luanne
                                                     Type: SExpression
```

Now we get more complicated. We illustrate various ways to store the formula $3x^2 + 5$ in different domains. Each domain has a chosen representation.

```
(5) -> devaluate(3*x^2+5)$Lisp

   (5)  (1 x (2 0 . 3) (0 0 . 5))
                                                     Type: SExpression
```

The fourth object, $3x^2 + 5$ is from the domain **POLY(INT)**. It is stored as the list

```
(1 x (2 0 . 3) (0 0 . 5))
```

From the domain **POLY** (Vol 10.3, POLY) we see that

```
Polynomial(R:Ring): ...
   == SparseMultivariatePolynomial(R, Symbol) add ...
```

So objects from this domain are represented as **SMP(INT,SYMBOL)**. From this domain we ss that

```
SparseMultivariatePolynomial(R: Ring,VarSet: OrderedSet): ...
  == add
    --representations
    D := SparseUnivariatePolynomial(%)
```

So objects from this domain are represented as a **SUP(INT)**

```
SparseUnivariatePolynomial(R:Ring): ...
  == PolynomialRing(R,NonNegativeInteger) add
```

So objects from this domain are represented as **PR(INT,NNI)**

```
PolynomialRing(R:Ring,E:OrderedAbelianMonoid): ...
  FreeModule(R,E) add
    --representations
    Term:=  Record(k:E,c:R)
    Rep:=  List Term
```

So objects from this domain are represented as **FM(INT,NNI)**

```
FreeModule(R:Ring,S:OrderedSet):
   == IndexedDirectProductAbelianGroup(R,S) add
     --representations
     Term:=  Record(k:S,c:R)
     Rep:=  List Term
```

So objects from this domain are represented as **IDPAG(INT,NNI)**

```
IndexedDirectProductAbelianGroup(A:AbelianGroup,S:OrderedSet):
  ==  IndexedDirectProductAbelianMonoid(A,S) add
```

So objects from this domain are represented as **IDPAM(INT,NNI)**

```
IndexedDirectProductAbelianMonoid(A:AbelianMonoid,S:OrderedSet):
  ==  IndexedDirectProductObject(A,S) add
    --representations
    Term:=  Record(k:S,c:A)
    Rep:=  List Term
```

So objects from this domain are represented as **IDPO(INT,NNI)**

```
IndexedDirectProductObject(A:SetCategory,S:OrderedSet):
 == add
       -- representations
       Term:=  Record(k:S,c:A)
       Rep:=  List Term
```

```
(6) -> devaluate((3*x^2+5)::FRAC(POLY(INT)))$Lisp

   (6)  ((1 x (2 0 . 3) (0 0 . 5)) 0 . 1)
                                                            Type: SExpression

(7) -> devaluate((3*x^2+5)::POLY(FRAC(INT)))$Lisp

   (7)  (1 x (2 0 3 . 1) (0 0 5 . 1))
                                                            Type: SExpression

(8) -> devaluate((3*x^2+5)::EXPR(INT))$Lisp

   (8)  ((1 [[x,0,%symbol()()()],NIL,1,1024] (2 0 . 3) (0 0 . 5)) 0 . 1)
                                                            Type: SExpression

(9) -> devaluate(w)$Lisp

   (9)
   (((PositiveInteger) . 1)  ((Float) 265633114661417543270 . - 65)
    ((String) . luanne)  ((Polynomial (Integer)) 1 x (2 0 . 3) (0 0 . 5))
    ((Fraction (Polynomial (Integer))) (1 x (2 0 . 3) (0 0 . 5)) 0 . 1)
    ((Polynomial (Fraction (Integer))) 1 x (2 0 3 . 1) (0 0 5 . 1))

     ((Expression (Integer))
      (1 [[x,0,%symbol()()()],NIL,1,1024] (2 0 . 3) (0 0 . 5))  0  .  1)
     )
                                                            Type: SExpression
```

2.2 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

[ADAx83] U.S. Government. *The Programming Language Ada Reference Manual.* U.S. Government, 1983.

Comment: STD-1815A-1983

[Ausi79] Giovanni Francesco Mascari Ausiello. On the design of algebraic data structures with the approach of abstract data types. *LNCS*, 72:514–530, 1979.

Abstract: The problem of giving a formal definition of the representation of algebraic data structures is considered and developped in the frame work of the abstract data types approach. Such concepts as canonical form and simplification are formalized and related to properties of the abstract specification and of the associated term rewriting system.

[Burs77] R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. In *IJCAI 77 Volume 2*, pages 1045–1058, 1977.

[Cohn65] Paul Moritz Cohn. *Universal Algebra.* Harper and Row, 1965.

[Deme79] A. Demers and J. Donahue. Revised report on russell. technical report TR 79-389, Cornell, 1979.

[Gris76] Martin L. Griss. The definition and use of data structures in reduce. In *SYMSAC '76*, pages 53–59, 1976.

Abstract: This paper gives a brief description and motivation of the mode analyzing and data-structuring extensions to the algebraic language REDUCE. These include generic functions, user defined recursive data structures, mode transfer functions and user modifiable automatic coercion. A number of examples are given to illustrate the style and features of the language, and how it will aid in the construction of more efficient and reliable programs.

[Jenk74] Richard D. Jenks. The scratchpad language. *ACM SIGPLAN Notices*, 9(4):101–111, 1974.

Abstract: SCRATCHPAD is an interactive system for symbolic mathematical computation. Its user language, originally intended as a special-purpose non-procedural language, was designed to capture the style and succinctness of common mathematical notations, and to serve as a useful, effective tool for on-line problem solving. This paper describes extensions to the language which enable it to serve also as a high-level programming language, both for the formal description of mathematical algorithms and their efficient implementation.

Comment: reprinted in SIGSAM Bulletin, Vol 8, No. 2, pp 20-30 May 1974

[Jenk81] Richard D. Jenks and Barry M. Trager. A language for computational algebra. In *Proc. Symp. on Symbolic and Algebraic Manipulation*, SYMSAC 1981, 1981.

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

Comment: IBM Research Report 8930

[Knut92] Donald E. Knuth. *Literate Programming.* Center for the Study of Language and Information, Stanford CA, 1992, 0-937073-81-4.

[Leve80] B. Levenworth. Adapt reference manual, 1980.

Comment: IBM Research

[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.

[That82] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Data type specification: Parameterization and the power of specification techniques. *ACM TOPLAS*, 4(4):711–732, 1982.

Abstract: Our earlier work on abstract data types is extended by the answers to a number of questions on the power and limitations of algebraic specification techniques and by an algebraic treatment of parameterized data types like **sets-of()** and **stacks-of-()**. The "hidden function" problem (the need to include operations in specifications which are wanted hidden from the user) is investigated; the relative power of conditional specifications and equational specifications is investigated; the relative power of conditional specifications and equational specifications is investigated; and it is shown that parameterized specifications must contain "side conditions" (e.g. that **finite-sets-of-d** requires an equality predicate on **d**).

[Wegb74] Ben Wegbreit. The treatment of data types in el1. *Communications of the ACM*, 17(5):251–264, 1974.

Abstract: In constructing a general purpose programming language, a key issue is providing a sufficient set of data types and associated operations in a manner that permits both natural problem-oriented notation and efficient implementation. The EL1 language contains a number of features specifically designed to simultaneously satisfy both requirements. The resulting treatment of data types includes provision for programmer-defined data types and generaic routines, programmer control over type conversion, and very flexible data type behavior, in a context that allows efficient compiled code and compact data representation.

[Wulf76] William A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of alphard programs. *IEEE Tr. Software Engineering*, SE-2(4):253–265, 1976.

Abstract: The programming language Alphard is designed to provide support for both the methodologies of "well-structured" programming and the techniques of formal program verification. Language constructs allow a programmer to isolate an abstraction, specifying its behavior publicly while localizing. knowledge about its implementation. The verification of such an abstraction consists of showing that its implementation behaves in accordance with its public specifications; the abstraction can then be used with confidence in constructing other programs, and the verification of that use employs only the public specifications. This paper introduces Alphard by developing and verifying a data structure definition and a program that uses it. It shows how each language construct contributes to the development of the abstraction and discusses the way the language design and the verification methodology wete tailored to each other. It serves not only as an introduction to Alphard, but also as an example of the symbiosis between verification and methodology in language design. The strategy of program structuring, illustrated for Alphard, is also applicable to most of the "data abstraction" mechanisms now appearing.