How does one program in the AXIOM System

J.H. Davenport
School of Mathematical Sciences
University of Bath
Bath BA2 7AY
England
jhd@maths.bath.ac.uk

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

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

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

\[
\text{fac } n = \begin{cases} 
  n & \text{if } n < 3 \\
  n \times \text{fac}(n-1) & \text{else}
\end{cases}
\]

as a definition of the factorial function. A slightly more complicated example (taken from [IBM, 1991] — see also Jenks & Sutor [1992]) goes as follows.

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

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

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

This line, which produces the following output from AXIOM,

\[\{2, 3, 5, 7, 13, 17, 19, 31, 61, 89, \ldots\}\]

Type: Stream PositiveInteger

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

\[
\text{mersennePrime } n = \text{mersenne mersenneIndex}(n)
\]

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

\[
\text{mersennePrime } 5
\]

Compiling function mersennePrime with type PositiveInteger -> Integer

\[\{8191\} \quad \text{Type: PositiveInteger}\]

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

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

* AXIOM is a trade mark of NAG Ltd.
Programming concepts

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

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

```plaintext
**$\times $**: ($$, $$) -> $$
**$\times $**: (Integer$$, $$) -> $$
**$-$**: $$ -> $$
**$-$**: ($$, $$) -> $$
**$<$**: ($$, $$) -> Boolean
**$=$**: ($$, $$) -> Boolean

1 : () -> $$
abs : $$ -> $$
associates? : ($$, $$) -> Boolean

binomial : ($$, $$) -> $$
bit? : ($$, $$) -> Boolean

coerce : $$ -> $$
coerce : Integer -> $$
convert : $$ -> Float
convert : $$ -> Integer

convert : min : ($$, $$) -> $$

modp : ($$, $$) -> $$
nextItem : $$ -> Union($$, "failed")

odd? : $$ -> Boolean
one? : $$ -> Boolean

positive? : $$ -> Boolean

powmod : ($$, $$, $$) -> $$
prime? : $$ -> Boolean
$q$uot$ : ($$, $$) -> $$

random : () -> $$

rational : $$ -> Fraction Integer
rational? : $$ -> Boolean

recip : $$ -> Union($$, "failed")

retract : $$ -> Integer

sign : $$ -> Integer

squareFree : $$ -> Factored $$

submod : ($$, $$, $$) -> $$

unit? : $$ -> Boolean

unitCanonical : $$ -> $$

characteristic : () -> NonNegativeInteger

differentiate : ($$, NonNegativeInteger) -> $$

divide : ($$, $$) -> Record(quotient: $$, remainder: $$)

euclideanSize : $$ -> NonNegativeInteger

expressIdealMember : (List $$, $$) -> Union(List $$, "failed")

$q$exquo$ : ($$, $$) -> Union($$, "failed")

extendedEuclidean : ($$, $$) -> Record(coef1: $$, coef2: $$, generator: $$)

extendedEuclidean : ($$, $$, $$) -> Union(Record(coef1: $$, coef2: $$), "failed")

multiEuclidean : (List $$, $$) -> Union(List $$, "failed")

patternMatch : ($$, Pattern Integer, PatternMatchResult(Integer$$)) -> PatternMatchResult($$

principalIdeal : List $$ -> Record(coef: List $$, generator: $$)

rationalIfCan : $$ -> Union(Fraction Integer, "failed")

reducedSystem : Matrix $$ -> Matrix Integer

reducedSystem : (Matrix $$, Vector $$) -> Record(mat: Matrix Integer, vec: Vector Integer)

retractIfCan : $$ -> Union(Integer, "failed")

unitNormal : $$ -> Record(unit: $$, canonical: $$, associate: $$)
```
It must be stressed that the use of \texttt{)show} or the browser is essential to understanding what is already present in AXIOM, and what one has to add to produce a valid domain. In fact, a user cannot write a domain, merely a function (see later) which, when called, will create a domain.

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

\begin{verbatim}
$** : ($,$) -> $  $** : (Integer,$) -> $  
$*** : ($,NonNegativeInteger) -> $  $** : ($,$) -> $  
- : $ -> $  $- : ($,$) -> $  
$== : ($,$) -> Boolean  1 : () -> $  
0 : () -> $  coerce : Integer -> $  
recip : $ -> Union($,"failed")  zero? : $ -> Boolean
characteristic : () -> NonNegativeInteger
\end{verbatim}

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

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

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

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

**Package.** Very like a functor, but does not specify a new data object, merely some new functions. A typical example would be \texttt{UnivariatePolynomialFunctions2}, which defines the operation

\begin{verbatim}
map: (R -> S, UnivariatePolynomial(x,R)) ->UnivariatePolynomial(y,S)
where x and y are elements of \texttt{Symbol}, and R and S belong to the category \texttt{Ring}. In
mathematical terms, this function takes a function \texttt{\theta} from \texttt{R} to \texttt{S}, and performs
the corresponding function from \texttt{R[x]} to \texttt{S[y]}. To get the actual function from \texttt{R[x]} to \texttt{S[y]},
one would have to use AXIOM's notation for "lambda expressions", viz. \texttt{map(f,#1)},
i.e. that function which, given an element \texttt{p} of \texttt{UnivariatePolynomial(x,R)}, computes
\texttt{map(f,p)}.
\end{verbatim}

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

3
The first problem — definition

Our aim here is to emulate CAMAL's [fitch, 1974] handling of “weighted polynomials”, a concept which is also found in Reduce [Hearn, 1987] via the commands weight and utlevel. For those not familiar with the idea, we give a quick summary here. We will then develop two alternative implementations incrementally. The complete definitions will be given in appendices.

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

The outline implementation we suggest is conceptually similar to that of Reduce [Hearn, 1987]. The weight is stored as the exponent of a virtual variable ($k\star$ in Reduce), and monomials are stored as coefficients of the appropriate power of this variable. Reduce does not ensure that $k\star$ has to be the most significant variable, in terms of polynomial ordering, and hence the truncation is not as efficient as it might be.

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

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

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

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

* An OrderedCancellationAbelianMonoid is a cancellation abelian monoid which is also a totally ordered set, such that the ordering is compatible with addition: $x < y \Rightarrow x + z < y + z$. Since it is a cancellation abelian monoid, i.e. satisfies $x + z = y + z \Rightarrow x = y$, there is a partial subtraction operation: $x - y$ is the unique $z$ such that $z + y = x$, if it exists. An OrderedAbelianMonoidSup is an OrderedCancellationAbelianMonoid in which, in addition, there is an operation $\sup$ with respect to the partial ordering induced by subtraction. In other words $\sup(x, y) - x$ and $\sup(x, y) - y$ exist, and $\sup(x, y)$ is minimal (with respect to $<$) with this property.
• FreeModule is used in the implementation of PolynomialRing, as can be found by using the Lineage option of the browser. This takes as arguments a Ring $R$ and an OrderedSet $S$, and generates the free module over $R$ whose generators are indexed by the elements of $S$. PolynomialRing builds on this, by keeping the definition of addition etc., but adding definitions of multiplication, relying on the addition between the exponents to define the multiplication of polynomials. Let us see precisely how this is defined (we have deleted lines redundant to the expository points we wish to make*).

PolynomialRing(R:Ring,E:OrderedAbelianMonoid):  
  FiniteAbelianMonoidRing(R,E) with  
  if R has canonicalUnitNormal then canonicalUnitNormal  
  == FreeModule(R,E) add  
  Term:= Record(k:E,c:R)  
  Rep:= List Term  
  1 == [[0$E,1$R]]  
  p1,p2: $  
  if R has EntireRing then  
      p1 * p2 ==  
      null p1 => 0  
      null p2 => 0  
      p1.first.k = 0 => p1.first.c * p2  
      p2 = 1 => p1  
      /**[[t1.k+t2.k,t1.c+t2.c]$Term for t2 in p2]  
        for t1 in reverse(p1)]  
      -- This 'reverse' is an efficiency improvement:  
      -- reduces both time and space [Abbott/Bradford/Davenport]  
  else  
      p1 * p2 ==  
      null p1 => 0  
      null p2 => 0  
      p1.first.k = 0 => p1.first.c * p2  
      p2 = 1 => p1  
      /**[[t1.k+t2.k,r]$Term for t2 in p2 | (r:=t1.c+t2.c) ^= 0]  
        for t1 in reverse(p1)]  
      -- This 'reverse' is an efficiency improvement:  
      -- reduces both time and space [Abbott/Bradford/Davenport]  
  if R has CommutativeRing then  
      p ** nn ==  
      null p => 0  
      nn = 0 => 1  
      p.rest = [] => [[nn * p.first.k, p.first.c ** nn]]  
      binomThmExpt([p.first], p.rest, nn)  
      binomThmExpt(x,y,nn) ==  
      nn = 0 => 1$$.  

* And changed the format of the header. The actual header in the AXIOM system is defined using the where syntax for introducing abbreviations, so it begins  
  PolynomialRing(R:Ring,E:OrderedAbelianMonoid): T == C  
  where  
      T == FiniteAbelianMonoidRing(R,E) with  
      if R has canonicalUnitNormal then canonicalUnitNormal  
      C == FreeModule(R,E) add  
  This has exactly the same meaning as that given in the body of the paper.
ans, xn, yn: $
bincoef: \text{Integer}
powl: \text{List}() := [x]
for i in 2..nn repeat powl:=[x * powl.first, :powl]
yn:=$y; ans:=powl.first; i:=1; bincoef:=nn
for xn in powl.rest repeat
  ans:= bincoef * xn * yn + ans
  bincoef:=(nn-i) * bincoef quo (i+1); i:= i+1
-- last I and BINCOEF unused
yn:=$y * yn
ans + yn

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

The returned domain belongs to the category $\text{FiniteAbelianMonoidRing}^*$, with the additional property $\text{canonicalUnitNormal}$ (see Davenport & Trager [1990] for an explanation of this property) if the ground ring $R$ has this property. The implementation is to take $\text{FreeModule}(R, E)$, and to add (hence the use of this keyword) certain additional operations — we have just quoted the definition of the unit and multiplication, in fact there are more. We find it convenient to work in terms of the internal representation of $\text{FreeModule}$, hence the $\text{Rep}$ line (which in turn relies on the definition of $\text{Term}$). We will see further examples of this methodology later on, as method (4) for the definition of AXIOM functors.

The first problem — specification

Proceeding in a top-down fashion, we can see that we are going to need a construction which takes as arguments a $\text{Ring}$ $R$, some weights for some symbols, and an initial weight level. This will return a $\text{Ring}$ as result, the ring of weighted polynomials, in the named symbols (at least), over $R$, with the weight level as specified. At this point, certain design decisions need to be made.

- Should the weight level be changeable? In Reduce, it is, and advice from the theory of repeated approximation (see, for example, Barton & ffitch [1972]) led us to believe that the weight level should be changeable.
- Should the weights themselves be changeable? In Reduce, they are not, and making them changeable would require the re-computation of the weights of all products. Of course, the user of AXIOM is free to build two different domains with different weights assigned in each, a flexibility that is not possible in Reduce, and we believe that this should be sufficient.
- How should the weights be represented. We could produce a separate AXIOM data type, we could accept them as equations, and insist at run time that they be of the form “symbol = non-negative integer”, or we could treat them as a list of symbols and a corresponding list of non-negative integers. For simplicity, we chose the last as the user interface (but see later for the internal handling).

$^*$ An “abelian monoid ring” bears the obvious relationship to a “group ring”: viz. it is the set of formal sums of ring elements, indexed by elements of the abelian monoid, with addition etc. being defined component-wise, and multiplication making use of the addition of abelian monoid indices. The use of the word “finite” here is to indicate that we consider only finite sums, i.e. the ring element is zero for all but finitely many elements of the abelian monoid.
• Should the weighted polynomials contain only the symbols specified in the weight list, or others? This is debatable, but it seemed simpler, as the implementation progressed, to allow other symbols, which then effectively have a weight of 0.

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

)abbrev domain OWP OrdinaryWeightedPolynomials

OrdinaryWeightedPolynomials(R:Ring,
    vl:List Symbol, w1:List NonNegativeInteger,
    wtlevel:NonNegativeInteger):

    Ring with
    if R has CommutativeRing then Algebra(R)
    coerce: $ -> Polynomial(R)
    ++ convert back into a Polynomial(R), ignoring weights
    coerce: Polynomial(R) -> $
    ++ coerce a Polynomial(R) into Weighted form,
    ++ applying weights and ignoring terms
    if R has Field then "/": ($,$) -> Union($,"failed")
    ++ division (only works if minimum weight
    ++ of divisor is zero, and if R is a Field)
    changeWeightLevel: NonNegativeInteger -> Void
    ++ This changes the weight level to the new value given:
    ++ NB: previously calculated terms are not affected

The )abbrev line is necessary for the definition of any functor*, since the abbreviation (up to eight letters, or seven for a category) defines, among other things, the name of the directory in which the compiled code will be stored.

AXIOM comments begin with either -- or ++. The former fulfil the traditional rôle of commenting programs. The latter, which can only appear in the appropriate contexts, are picked out by the program that builds the HyperDoc database, and can be retrieved by HyperDoc when it comes to describing operations (as in the case of the changeWeightLevel operation quoted above) or whole constructors.

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

== WeightedPolynomials(R,Symbol,IndexedExponents(Symbol),
    Polynomial(R),
    vl,w1,wtlevel)

This is essentially an add form in which nothing is being added: the operations of OrdinaryWeightedPolynomials will be precisely those of WeightedPolynomials.

The header of WeightedPolynomials now practically writes itself.

)abbrev domain WP WeightedPolynomials

WeightedPolynomials(R:Ring,VarSet: OrderedSet, E:OrderedAbelianMonoidSup,
    P:PolynomialCategory(R,E,VarSet),
    vl:List VarSet, w1:List NonNegativeInteger,

* It could well be argued that the syntax ought to be )abbrev functor not )abbrev domain, but this has to be regarded as a historical accident.
**wtlevel**: NonNegativeInteger

Ring with
if R has CommutativeRing then Algebra(R)
coerce: $ -> P
++ convert back into a "P", ignoring weights
if R has Field then "/": ($,$) -> Union($,"failed")
++ division (only works if minimum weight
++ of divisor is zero, and if R is a Field)
coerce: P -> $
++ coerce a "P" into Weighted form, applying weights and ignoring terms
changeWeightLevel: NonNegativeInteger -> Void
++ This changes the weight level to the new value given:
++ NB: previously calculated terms are not affected

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

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

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

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

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

### The first problem — an implementation

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

```
Rep := PolynomialRing(P,NonNegativeInteger)
w,x1,x2:=$
0 == 0$Rep
1 == 1$Rep
x1 = x2 ==
   -- Note that we must strip out any terms greater than wtlevel
   while degree x1 > wtlevel repeat
     x1 := reductum x1
   while degree x2 > wtlevel repeat
     x2 := reductum x2
   x1 =$Rep x2
x1 + x2 == x1 +$Rep x2
-x1 == -$Rep x1
x1 * x2 ==
   -- Note that this is probably an extremely inefficient definition
w:=x1 *$Rep x2
while degree(w) > wtlevel repeat
   w:=reductum w
w
```
One important point that crops up here is the necessity to distinguish the operations of the representation (PolynomialRing) from those of the type being defined. Since elements can be viewed as belonging to either the data type or its representation, there is a potential for ambiguity, when the data type and the representation have operations of the same signature. In this case, the unqualified operation name will refer to that of the data type, and that of the representation has to be obtained by use of the Rep syntax — meaning use the operation from the data type Rep. A trivial example of the definition of + given above, which can be paraphrased in English as “to add two elements of WeightedPolynomials, add them as if they were elements of the representation, i.e. elements of PolynomialRing”. Slightly more complicated is the definition of equality, which can be paraphrased in English as “to test two elements of WeightedPolynomials for equality, first remove any terms of weight greater than the current weight level, then test them for equality as elements of the representation, i.e. elements of PolynomialRing”. It is perhaps worth noting that a side-effect of this is that calls to changeWeightLevel can affect the result of equality tests.

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

```plaintext
import RepeatedDoubling($)
if not ($ has Ring) then
  n:Integer * x:$ ==
  zero? n => 0
  n>0 => double(n pretend PositiveInteger,x)
  double(-n pretend PositiveInteger,-x)
```

i.e. by repeated doubling of the original input: hardly efficient, but certainly correct. In fact, PolynomialRing has a more efficient definition, inherited from FreeModule (who inherits it from IndexedDirectProductAbelianGroup), where the multiplication of the individual coefficients by integers is used. However, since we are only using those operations of PolynomialRing that we name explicitly, this definition is not seen. We could, of course, add it by having an extra line

```plaintext
n * x2 == n *$Rep x2
```

but this would need to be repeated for all such operations where we wished to use the PolynomialRing implementation rather than the default one.

**The first problem — a PolynomialRing implementation**

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

```plaintext
== PolynomialRing(P,NonNegativeInteger)
add
--representations
Term := Record(k:NonNegativeInteger,c:P)
Rep := List Term
w,x1,x2: $
\begin{align*}
x1 * x2 &=
null x1 => 0 
null x2 => 0 
r:P 
x1.first.k = 0 => 
  \left[ \left[ t2.k,rl \right] \right. \left. Term \text{ for } t2 \text{ in } x2 \mid (r:=x1.first.c * t2.c) = 0 \right] 
\end{align*}
```

9
\[ x_2 = 1 \Rightarrow x_1 \]
\*/[[n,r]$Term for t2 in x2 | (n:=t1.k+t2.k)<=w1level and 
\( r:=t1.c*t2.c \) ^= 0 ]

for t1 in reverse(x1)]

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

import RepeatedSquaring($)
x:$ ** n:NonNegativeInteger ==
zero? n => 1
expt(x,n pretend PositiveInteger)

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

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

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

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

The implications of this rule for exponentiation are as follows.

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

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

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

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

A related issue comes up in the definitions of zero? and one?. These are defined, in Abelian-Monoid and Monoid respectively, to have defaulting definitions zero? x == x=0 and one? x == x=1. Since these definitions happen not to be over-ridden in the add chain, they are the definitions that apply in WeightedPolynomials, and so use WeightedPolynomials' definition of equality.
However, were a later author of PolynomialRing to add other definitions, these would be picked up instead.

The first problem — miscellaneous definitions
Here we give (in an appropriate format for the most recent implementation) some miscellaneous definitions that should form part of the implementation of WeightedPolynomials. The first three lines deal with the definition of changeWeightLevel.

\[
\begin{align*}
n: & \text{NonNegativeInteger} \\
\text{changeWeightLevel}(n) : & = \\
\text{wtlevel} : = & n
\end{align*}
\]

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

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

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

\[
\begin{align*}
p:P \\
z: & \text{Integer} \\
\text{innercoerce}: (p,z) & \rightarrow \$ \\
\text{innercoerce}(p,z) : & = \\
z & < 0 \Rightarrow 0 \\
\text{zero?} \ p & \Rightarrow 0 \\
\text{mv} &: = \text{mainVariable} p \\
\text{mv case} & "\text{failed}" \Rightarrow [[0,p]] \\
n &: = \text{lookup}(\text{mv}) \\
\text{up} &: = \text{univariate}(p,\text{mv}) \\
\text{ans} &: = $ \\
\text{ans} &: = 0 \\
\text{while not zero?} \ \text{up} \ \text{repeat} \\
\text{d} &: = \text{degree} \ \text{up} \\
f &: = n + d \\
\text{lcup} &: = \text{leadingCoefficient} \ \text{up} \\
\text{up} &: = \text{up-leadinMonomial} \ \text{up} \\
\text{mon} &: = \text{monomial}(1,\text{mv},d) \\
f & < z \Rightarrow \text{ans} = \text{ans} + \[[\text{tm}.k+f,\text{mon} \cdot \text{tm}.c]\] \\
\text{for } \text{tm} \ \text{in innercoerce}(\text{lcup},z-f)]
\end{align*}
\]
ans
corece(p):$ == innercorece(p,wtlevel)
The inverse operation is much simpler: we have merely to add up the coefficients.
corece(w):P == "*"/[tm.c for tm in w]
The last definition is that of coercion from WeightedPolynomials into OutputForm — AXIOM's

type for output (and conversion to TeX etc.). This is fairly simple: the main complexity is in the
specification. Here we have decided that a single term of zero weight will print as such, but that
otherwise each group of terms of a particular weight will be printed parenthesised (even if there is
only one term of that weight). Clearly, it would be possible to adapt this definition to almost any
other desired behaviour.
corece(p):OutputForm ==
  zero? p => (0$Integer):OutputForm
  p.first.k == 0 => p.first.c :: OutputForm
  reduce("+",(reverse [paren(t1.c::OutputForm for t1 in p]):List OutputForm)
The second problem — definition
Our aim here is to implement an equivalent of CAMAL's [ffitch, 1974] handling of truncated Fourier
series. We have some domain of "angles" — in CAMAL's case linear combinations with integer
coefficients (lying in the range \{-63\ldots63\}) of the eight angular variables \{s\ldots z\}. We can build sin
or cos of these variables, and use them as coefficients in polynomial expressions, where products of
trigonometric functions are always linearised. There are more operations provided in CAMAL, e.g.
integration with respect to an angular variable, but we will not bother with these for simplicity of
exposition.
Within CAMAL, the coefficients of expressions in these trigonometric functions will therefore
not involve other trigonometric functions, but will involve weighted polynomials. These we have
already defined, and there seems no absolute need to use weighted polynomials, though they are
in practice the most common type of coefficients required. However, we probably need to assume
that the coefficients commute with each other and with the trigonometric terms, since otherwise
the linearisation of products is not well-defined. Furthermore, since
\[
\sin(A) \sin B = \frac{\cos(A - B) - \cos(A + B)}{2},
\]
we must be able to divide by two. For simplicity, therefore, we insist on the ability to divide by any
non-zero integer, i.e. that the coefficients should be an Algebra over \( \mathbb{Q} \), the AXIOM type Fraction
Integer.
The second problem — specification
The header of our type FourierSeries now nearly writes itself. The arguments of the trigonometric
functions had better be an ordered set, so that we can order the various trigonometric functions, and
an abelian group so that the addition and subtraction rules can take place. It would therefore be
possible to require that the domain of these arguments should be an OrderedAbelianGroup, but this
may be too strong, and we will restrict ourselves to insisting on Join(OrderedSet,AbelianGroup)^4.
FourierSeries(R:Join(CommutativeRing,Algebra(Fraction Integer)),
E:Join(OrderedSet,AbelianGroup)):

Algebra(R) with

* An OrderedAbelianGroup would also have the property that \( a < b \Rightarrow a + c < b + c \), but we
probably do not need this.
if E has canonical and R has canonical then canonical
coerce: R -> $
++ Convert coefficients into Fourier Series
coerce: FourierComponent(E) -> $
++ Convert sin/cos terms into Fourier Series
makeSin: (E,R) -> $
++ make a sin expression with given argument and coefficient
makeCos: (E,R) -> $
++ make a sin expression with given argument and coefficient

The operations here (with the exception of the last coerce, which will be explained in the next section, are pretty obvious. What about the line containing the work “canonical”? AXIOM’s definition of the attribute canonical is that a domain is canonical if mathematical equality implies equality of data structure. In particular, it authorises the use of hash-based techniques. There is a discussion in Davenport & Trager [1990] of why this is not as obvious as it seems. In our case, we are saying that, if the coefficients and arguments are canonical, then the data type returned will also be.

The obvious implementation of this is via some kind of FreeModule, using $R$ as the coefficients and the trigonometric functions as the indices. However, we first need to define the trigonometric functions themselves, and this is the purpose of the next section. We will return to the type FourierSeries in the section following.

The second problem — implementation of FourierComponent
It would be possible to use AXIOM’s general-purpose type Expression to represent the trigonometric functions, but we settled, partly for pedagogic reasons and partly to keep our code reasonably self-contained, on a separate data type.

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

FourierComponent(E:OrderedSet):
   OrderedSet with
      sin: E -> $
      ++ Makes a sin kernel for use in Fourier series
cos: E -> $
      ++ Makes a cos kernel for use in Fourier series
      sin?: $ -> Boolean
      ++ true if term is a sin, otherwise false
      argument: $ -> E
      ++ returns the argument of a given sin/cos expressions

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

   add
   --representations
   Rep:=Record(SinIfTrue:Boolean, arg:E)
e:E
   x,y:$
sin e == [true,e]
cos e == [false,e]
sin? x == x.SinIfTrue
argument x == x.arg
The harder question is the order to be imposed on FourierComponent. We have chosen, for no very good reason, to use the order of the arguments, and break ties by sorting \( \cos \alpha \) as less than \( \sin \alpha \). Clearly this definition could be adapted to any other strategy.

```haskell
x < y ==
  x.arg < y.arg => true
  y.arg < x.arg => false
  x.SinIfTrue => false
  y.SinIfTrue
```

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

```haskell
coerce(x):OutputForm ==
  hconcat((if x.SinIfTrue then "sin" else "cos")::OutputForm,
            bracket((x.arg)::OutputForm))
```

### The second problem — implementation of FourierSeries

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

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

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

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

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

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

\[
x1*x2 ==
  null x1 => 0
  null x2 => 0
  +/[+[/multiply(t1,t2) for t2 in x2] for t1 in x1]
\]

Bibliography

Appendix A: Problem 1: First Implementation


--% WeightedPolynomials

)abbrev domain WP WeightedPolynomials

++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 12 July 1992
++ Basic Functions: Ring, changeWeightLevel
++ Related Constructors: PolynomialRing
++ Also See: OrdinaryWeightedPolynomials
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This domain represents truncated weighted polynomials over a general
++ (not necessarily commutative) polynomial type. The variables must be
++ specified, as must the weights.
++ The representation is sparse
++ in the sense that only non-zero terms are represented.

WeightedPolynomials(R:Ring,VarSet: OrderedSet, E:OrderedAbelianMonoidSup,
P:PolynomialCategory(R,E,VarSet),
vl:List VarSet, wl:List NonNegativeInteger,
wtlevel:NonNegativeInteger):

Ring with
if R has CommutativeRing then Algebra(R)
  coerce: $ -> P
++ convert back into a "P", ignoring weights
if R has Field then "/": ($,$) -> Union($,"failed")
++ division (only works if minimum weight
++ of divisor is zero, and if R is a Field)
  coerce: P -> $
++ coerce a "P" into Weighted form, applying weights and ignoring terms
changeWeightLevel: NonNegativeInteger -> Void
++ This changes the weight level to the new value given:
++ NB: previously calculated terms are not affected
==

add
--representations
Rep := PolynomialRing(P,NonNegativeInteger)
p:P
w,x1,x2:$
n:NonNegativeInteger
z:Integer
changeWeightLevel(n) ==
  wtlevel:=n
lookupList:List Record(var:VarSet, weight:NonNegativeInteger)
if #vl ~= #w then error "incompatible length lists in WeightedPolynomial"
lookupList:=[(v,n) for v in vl for n in wtlevel]
-- local operation
innercoerce:(p,z) -> $
lookup:Varset -> NonNegativeInteger
lookup v ==
  l:=lookupList
  while l ~= [] repeat
    v = l.first.var => return l.first.weight
  l:=l.rest
0
innercoerce(p,z) ==
z<0 => 0
zero? p => 0
mv:=mainVariable p
mv case "failed" => monomial(p,0)
n:=lookup(mv)
up:=univariate(p,mv)
ans:$
ans:=0
while not zero? up repeat
  d:=degree up
  f:=n*d
  lcup:=leadingCoefficient up
  up:=up-leadingMonomial up
  mon:=monomial(1,mv,d)
  f<=z =>
    tmp:= innercoerce(lcup,z-f)
    while not zero? tmp repeat
      ans:=ans+ monomial(mon*leadingCoefficient(tmp),degree(tmp)+f)
      tmp:=reductum tmp
  ans
coerce(p):$ == innercoerce(p,wtlevel)
coerce(w):P == "+/[c for c in coefficients w]
coerce(p):$):OutputForm ==
  zero? p => (0$Integer):::OutputForm
  degree p = 0 => leadingCoefficient(p):: OutputForm
  reduce("+",(reverse [paren(c::OutputForm) for c in coefficients p])
    ::List OutputForm)
0 == 0$Rep
1 == 1$Rep
x1 = x2 ==
  -- Note that we must strip out any terms greater than wtlevel
  while degree x1 > wtlevel repeat
    x1 := reductum x1
  while degree x2 > wtlevel repeat
    x2 := reductum x2
  x1 =$Rep x2
x1 + x2 == x1 +$Rep x2
-x1 == -(x1::Rep)
x1 * x2 ==
  -- Note that this is probably an extremely inefficient definition
w:=x1 *$Rep x2
  while degree(w) > wtlevel repeat
    w:=reductum w
w

)abbrev domain OWP OrdinaryWeightedPolynomials
++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 12 July 1992
++ Basic Functions: Ring, changeWeightLevel
++ Related Constructors: WeightedPolynomials
++ Also See: PolynomialRing
++ AMS classifications:
++ Keywords:
++ References:
++ Description:
++ This domain represents truncated weighted polynomials over the
++ "Polynomial" type. The variables must be
++ specified, as must the weights.
++ The representation is sparse
++ in the sense that only non-zero terms are represented.

OrdinaryWeightedPolynomials(R:Ring,
    vl:List Symbol, wl:List NonNegativeInteger,
    wtlevel:NonNegativeInteger):

Ring with
    if R has CommutativeRing then Algebra(R)
    coerce: $ -> Polynomial(R)
++ convert back into a Polynomial(R), ignoring weights
    coerce: Polynomial(R) ->$
++ coerce a Polynomial(R) into Weighted form,
++ applying weights and ignoring terms
    if R has Field then "/": ($,$) -> Union($,"failed")
++ division (only works if minimum weight
++ of divisor is zero, and if R is a Field)
    changeWeightLevel: NonNegativeInteger -> Void
++ This changes the weight level to the new value given:
++ NB: previously calculated terms are not affected
    == WeightedPolynomials(R,Symbol,IndexExponents(Symbol),
                  Polynomial(R),
                  vl,wl,wtlevel)

Appendix B: Problem 1: Second Implementation


--% WeightedPolynomials

)abbrev domain WP WeightedPolynomials

++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 12 July 1992
++ Basic Functions: Ring, changeWeightLevel
++ Related Constructors: PolynomialRing
++ Also See: OrdinaryWeightedPolynomials

18
AMS classifications:

Keywords:

References:

Description:

This domain represents truncated weighted polynomials over a general (not necessarily commutative) polynomial type. The variables must be specified, as must the weights.

The representation is sparse in the sense that only non-zero terms are represented.

WeightedPolynomials(R:Ring,VarSet: OrderedSet, E:OrderedAbelianMonoidSup, P:PolynomialCategory(R,E,VarSet),
vl:List VarSet, wl:List NonNegativeInteger, wtlevel:NonNegativeInteger):

Ring with
  if R has CommutativeRing then Algebra(R)
  coerce: $ -> P
  + convert back into a "P", ignoring weights
  if R has Field then "/: (\$(\$) -> Union($,"failed")
  + division (only works if minimum weight
  + of divisor is zero, and if R is a Field)
  coerce: P -> $
  + coerce a "P" into Weighted form, applying weights and ignoring terms
  changeWeightLevel: NonNegativeInteger -> Void
  + This changes the weight level to the new value given:
  + NB: previously calculated terms are not affected

== PolynomialRing(P,NonNegativeInteger)

add
  -- representations
  Term := Record(k:NonNegativeInteger,c:P)
  Rep := List Term
  p:P
  w,x1,x2:$
  n:NonNegativeInteger
  z:Integer
  changeWeightLevel(n) ==
    wtlevel:=n
  lookupList:List Record(var:VarSet, weight:NonNegativeInteger)
  if #vl ~= #wl then error "incompatible length lists in WeightedPolynomial"
  lookupList:=[[v,n] for v in vl for n in wl]
  -- local operation
  innercoerce:(p,z) -> $
  lookup:Varset -> NonNegativeInteger
  lookup v ==
    l:=lookupList
    while l ~= [] repeat
      v = l.first.var => return l.first.weight
      l:=l.rest
    0
  innercoerce(p,z) ==
    z<0 => 0
    zero? p => 0
    mv:= mainVariable p

19
mv case "failed" => [[0,p]]
n:=lookup(mv)
up:=univariate(p,mv)
ans:=$
ans:=0
while not zero? up repeat
d:=degree up
f:=n*d
lcup:=leadingCoefficient up
up:=up-leadingMonomial up
mon:=monomial(1,mv,d)
f<=z => ans:=ans+[[tm.k+f,mon*tm.c]
    for tm in innercoerce(lcup,z-f)]
ans
coose(p):$ == innercoerce(p,wtlevel)
coose(w):P == "+"/[tm.c for tm in w]
x1 = x2 ==
    -- Note that we must strip out any terms greater than wtlevel
    while not null x1 and x1.first.k > wtlevel repeat
        x1 := x1.rest
    while not null x2 and x2.first.k > wtlevel repeat
        x2 := x2.rest
    while not null x1 and not null x2 repeat
        x1.first.k ^= x2.first.k => return false
        x1.first.c ^= x2.first.c => return false
        x1:=x1.rest
        x2:=x2.rest
null x1 and null x2
x1 * x2 ==
nul x1 => 0
null x2 => 0
r:P
x1.first.k = 0 =>
    [[t2.k,r]$Term for t2 in x2 | (r:=x1.first.c * t2.c) ^=0 ]
x2 = 1 => x1
+/[[n,r]$Term for t2 in x2 | (n:=t1.k+t2.k)<=wtlevel and
            (r:=t1.c*t2.c) ^= 0]
    for t1 in reverse(x1])
    -- This 'reverse' is an efficiency improvement:
    -- reduces both time and space [Abbott/Bradford/Davenport]
import RepeatedSquaring($)
x:$ ** n:NonNegativeInteger ==
    zero? n => 1
    expt(x,n pretend PositiveInteger)
coose(p:$):OutputForm ==
    zero? p => (0$Integer)::OutputForm
    p.first.k = 0 => p.first.c :: OutputForm
    reduce("+",(reverse [paren(t1.c::OutputForm) for t1 in p])::List OutputForm)
The implementation of OrdinaryWeightedPolynomials is as given in Appendix A.
Appendix C: Problem 2: Implementation


--% FourierComponent

)abbrev domain FCOMP FourierComponent

++ Author: James Davenport
++ Date Created: 17 April 1992
++ Date Last Updated: 12 June 1992
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:

FourierComponent(E:OrderedSet):
    OrderedSet with
    sin: E -> $
    ++ Makes a sin kernel for use in Fourier series
    cos: E -> $
    ++ Makes a cos kernel for use in Fourier series
    sin?: $ -> Boolean
    ++ true if term is a sin, otherwise false
    argument: $ -> E
    ++ returns the argument of a given sin/cos expressions
    ==
    add
    --representations
    Rep:=Record(SinIfTrue:Boolean, arg:E)
    e:E
    x,y:$
    sin e == [true,e]
    cos e == [false,e]
    sin? x == x.SinIfTrue
    argument x == x.arg
    coerce(x):OutputForm ==
    hconcat(((if x.SinIfTrue then "sin" else "cos")::OutputForm,
               bracket((x.arg)::OutputForm))
    x<y ==
    x.arg < y.arg => true
    y.arg < x.arg => false
    x.SinIfTrue => false
    y.SinIfTrue

--% FourierSeries

)abbrev domain FSERIES FourierSeries

++ Author: James Davenport
FourierSeries(R:Join(CommutativeRing,Algebra(Fraction Integer)),
E:Join(OrderedSet,AbelianGroup)):
  Algebra(R) with
    if E has canonical and R has canonical then canonical
    coerce: R -> $
    + Convert coefficients into Fourier Series
    coerce: FourierComponent(E) -> $
    + Convert sin/cos terms into Fourier Series
    makeSin: (E,R) -> $
    + make a sin expression with given argument and coefficient
    makeCos: (E,R) -> $
    + make a sin expression with given argument and coefficient
    == FreeModule(R,FourierComponent(E))
add
  --representations
  Term := Record(k:FourierComponent(E),c:R)
  Rep := List Term
  multiply : (Term,Term) -> $
  w,x1,x2:$
  t1,t2:Term
  n:NonNegativeInteger
  z:Integer
  e:FourierComponent(E)
  a:E
  r:R
  l == [[cos 0,1]]
  coerce e ==
    sin? e and zero? argument e => 0
    if argument e < 0 then
      not sin? e => e:=cos(- argument e)
    return [[sin(- argument e),-1]]
  [[e,1]]
  multiply(t1,t2) ==
    r:=(t1.c*t2.c)*(1/2)
    s1:=argument t1.k
    s2:=argument t2.k
    sum:=s1+s2
    diff:=s1-s2
    sin? t1.k =>
    sin? t2.k =>
      makeCos(diff,r) + makeCos(sum,-r)
      makeSin(sum,r) + makeSin(diff,r)
    sin? t2.k =>
makeSin(sum,r) + makeSin(diff,r)
makeCos(diff,r) + makeCos(sum,r)
x1*x2 ==
null x1 => 0
null x2 => 0
+[/+[/multiply(t1,t2) for t2 in x2] for t1 in x1]
makeCos(a,r) ==
a<0 => [[cos(-a),r]]
[[cos a,r]]
makeSin(a,r) ==
zero? a => []
a<0 => [[sin(-a),-r]]
[[sin a,r]]