

Explaining algebraic theory with functional programs

Jeroen Fokker

Dept. of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
jeroen@cs.ruu.nl, <http://www.cs.ruu.nl/people/jeroen>

Abstract. A hierarchy of six important structures from abstract algebra (groups, rings, fields etc.) is introduced as Gofer class definitions and laws about them. Many instance declarations are provided, explaining the algebraic construction of integers, quotients, adding i , function spaces, polynomials, and matrices. The definitions include generalized implementations of polynomial division and matrix inversion. Monadic parsers are provided for all constructs discussed. As an application, a one-line program is given for calculating the eigenvalue equation of a matrix.

1 Introduction

Mathematicians are said to have recognized the importance of software reuse long before the advent of computers. Large ‘libraries’ of theorems are available, which can be applied to prove new theorems. A generally accepted quality criterion for mathematical theorems is that applicability should be limited by as few constraints as possible. The quest for simple, universally applicable theorems resembles the construction of general purpose software libraries.

In programming languages, a lot of notations and concepts were borrowed from mathematics: from the use of arithmetic expressions in Fortran, through the notions of sums and products of datatypes in Algol and C, to the exploitation of category theory to build graphical user interfaces in Haskell. But, being an engineering discipline, programming has developed its own notations for managing the complexity of large programs. Evolution has led us from a simple notion of modules in Simula and Modula to class hierarchies and inheritance in today’s object oriented languages. Typed functional programming languages contributed the idea of polymorphic data types.

It is time that programming pays back mathematics with some of these notations for structuring information. The concept of classes in Haskell and Gofer for overloading operators on polymorphic, parameterized data structures is very suitable for making explicit the assumptions for various concepts and algorithms in an algebra course.

In this article we introduce the notions of (among others) group, ring and field as being ‘classes’ in section 2. Examples of these classes (integers, polynomials etc.) are constructed as algebraic (!) data types, and made instance of the applicable classes in section 3. As an added benefit, the manipulation of sequences (e.g. in the case of polynomials and matrices) is eased by the use of well known standard functions on lists, available in the programming language [BW88,?,?]. In section 4, conversion to and from string representation is defined for the examples in section 3. Applications of the definitions are given in section 5, both in the form of new types as a special case, and in the form of concrete calculations.

Most of the examples were taken from a textbook on linear algebra [Lan70]. We believe that using a notation which appeals to the programmer’s intuition eases the comprehension of abstract ideas, especially in the case of computer science students. In fact, the material presented in this paper is covered, with some more explanation, as a case study in the first year course on functional programming for computer science students in Utrecht. Available time (10 weeks for the whole course, with no former knowledge of programming) and scope of that course prohibit us to treat all details in the course, but it should be enough to convince the students that mathematical algorithms can be presented quite elegantly using a functional programming language.

All courses on functional programming in Utrecht are summarized in appendix A. The code that is described in this article is available from the WWW address given below the title.

2 Classes

2.1 The algebraic hierarchy

One of the simplest algebraic structures one can imagine is that of a *monoid*. A set forms a monoid if it is closed under an associative binary operator, that has a neutral element. Using a class definition we can declare the types of the operator and its neutral element:

```
class Eq a => Monoid a where
  (<+>)  :: a->a->a
  zzero :: a
```

The laws that the <+> operator and the zzero constant have to obey are:

```
-- x <+> (y<+>z) == (x<+>y) <+> z
-- zzero <+> y   == y
-- x <+> zzero   == x
```

The laws are written in a comment to the class definition. Gofer, Haskell and most other functional languages do not allow laws to be defined in the language. This is a pity, because even if the implementation wouldn’t check the validity of the laws, at least they could be type checked.

Because in the laws the notion of equality is used, we have made the `Monoid` class a subclass of `Eq`, that is all instances of `Monoid` must also be instance of

Eq, by providing a definition for the equality operator ==. This is not strictly necessary, because it might be argued that the equality in the laws is only needed on a meta-level.

The next class in the algebraic hierarchy is a *group*. Instances of this class provide a function `neg`, which for each value gives its inverse under addition:

```
class Monoid a => Group a where
  neg      :: a->a
  -- x <+> neg x == zzero
  -- neg x <+> x == zzero
  -- x <+> y     == y <+> x
```

We also specified that in a group, addition should be commutative. Some authors do not include this law, and speak of a ‘commutative’ or ‘Abelian’ group when commutativity holds. We do not introduce a separate class here, as no new operators are introduced.

If a type supports an associative operator which distributes over addition, it is said to form a *rng*. Usually, this operator is called ‘multiplication’:

```
class Group a => Rng a where
  (<*>)  :: a->a->a
  -- x <*> (y<*>z) == (x<*>y) <*> z
  -- x <*> (y<+>z) == x<*>y <+> x<*>z
  -- (x<+>y) <*> z == x<*>z <+> y<*>z
```

The multiplication operator does not need to have a neutral element. If it has, the structure is called a *ring*:

```
class Rng a => Ring a where
  one     :: a
  -- one <*> x == x
  -- x <*> one == x
```

The word *rng* is chosen for the previous class because it is a ring which lacks an unity – symbolized by dropping the letter ‘i’.

If it is possible to divide with remainder, a structure is called an *Euclidean space*. The remainder of a division should be less than the denominator in some sense. In which sense ‘being smaller’ should be understood, is specified by the `degree` function:

```
class Ring a => Euclid a where
  degree  :: a->Int
  divide  :: a->a->a
  modulo  :: a->a->a
  x ‘modulo’ y | y/=zzero = x <+> neg((x ‘divide’ y) <*> y)
  -- y/=zzero | degree (x ‘modulo’ y) < degree y
  --           || x ‘modulo’ y == zzero
  -- y/=zzero | degree x <= degree (x<*>y)
```

One of the laws for Euclidian spaces, which states that multiplying the quotient by the denominator again and adding the remainder yields the numerator, is given as a default definition for `modulo` in the class definition.

The richest structure in the hierarchy is a *field*. In a field, every element but zero is required to have an exact inverse under multiplication:

```

class Ring a => Field a where
  inv      :: a->a
  --      x/=zzero | x <*> inv x == one

```

The definition of the `Field` class is independent from the `Euclid` class. Nevertheless, the classes introduced (`Monoid`, `Group`, `Rng`, `Ring`, `Euclid` and `Field`) can be thought of as a linear hierarchy, as every field can be made into an Euclidean space in a trivial way:

```

instance Field a => Euclid a where
  degree x      = 0
  x 'divide' y = x <*> inv y

```

2.2 Derived operators

Operators that are defined in terms of the operators in the class hierarchy are automatically overloaded for instances of the classes involved. We will give an example of this for all six classes in the hierarchy.

The well known `sum` function, which adds all element of a list starting with zero, can be generalized to an arbitrary `Monoid`:

```

summ :: Monoid a => [a] -> a
summ = foldr (<+>) zzero

```

In a group, we can use the notion of subtraction, which is defined as adding the negation:

```

(<->) :: Group a => a->a->a
x <-> y = x <+> neg y

```

When a multiplication is available (i.e., in a `Rng`), the square of a value is that value multiplied by itself:

```

square :: Rng a => a->a
square x = x <*> x

```

In a ring with unity, raising a value to a natural power can be defined. The type `Nat` of natural numbers will be defined in section 3.2.

```

(<^>) :: Ring a => a -> Nat -> a
x <^> Zer      = one
x <^> Suc n    = x <*> x <^> n

```

The Euclidean algorithm for determining the greatest common divisor can be applied not only to integers, but also to arbitrary instances of an Euclidean space:

```

gcDiv :: Euclid a => a->a->a
gcDiv x y | degree y<=0 = x
          | otherwise   = gcDiv y (x 'modulo' y)

```

Finally, in a field we can define a shorthand notation for division:

```

(</>) :: Field a => a->a->a
x </> y = x <*> inv y

```

Useful priorities for all operators defined are given by:

```

infixl 8 <^>
infix 7 <*>, </>
infix 7 'divide', 'modulo', 'gcDiv'
infixl 6 <+>, <->

```

2.3 Additional classes

In section 3 a lot of types will be made instance of the six classes in the hierarchy. For the easy manipulation of values, we will use a string representation for most of them. Parse and unparse functions are provided by making a type instance of `Repr`:

```
class Repr a where
  parse  :: Parser a
  unpars :: a -> ShowS
```

Here, `Parser` is the type of backtracking parsers:

```
type Parser a = String -> [(a,String)]
```

and `ShowS` is the type of functions that prepend some string to their argument:

```
type ShowS = String -> String
```

Finally, a class `Finite` will be used in section 3.6. Instances are finite types of which all members can be enumerated in a list:

```
class Finite a where
  members :: [a]
```

3 Instances

3.1 Primitives

The numeric built-in type `Int` can be made a `Group` and a `Ring` in a trivial way, by using the built-in addition and multiplication functions.

```
instance Monoid Int where
  zzero = 0
  (<+>) = (+)
instance Group Int where
  neg = negate
instance Rng Int where
  (<*>) = (*)
instance Ring Int where
  one = 1
```

Similarly, the type `Float` can be made a `Group` and a `Ring`. The integers form an Euclidean space. The `degree` function is taken to be the absolute value, so that `degree (x 'modulo' y) < degree y` as required, even if `y` is negative.

```
instance Euclid Int where
  degree = abs
  divide = div
  modulo = mod
```

Floating point values, as an approximation of real numbers, form a field:

```
instance Field Float where
  inv = (1.0/)
```

By the generic definition of section 2.2, they also form an Euclidean space in a trivial way.

3.2 Natural numbers

Without using built-in features, natural numbers can be defined as being either zero or the successor of another natural number.

```
data Nat = Zer
         | Suc Nat
```

The Peano axioms for the natural numbers (no confusion, ability to do induction) are implicit in the semantics of a `data` declaration. We can abstract from induction over natural numbers by a ‘fold’ over natural numbers:

```
foldNat :: a -> (a->a) -> Nat -> a
foldNat e f Zer      = e
foldNat e f (Suc x) = f (foldNat e f x)
```

Equality and ordering of natural numbers is expressed by:

```
instance Eq Nat where
  Zer == Zer      = True
  Suc x == Suc y  = x==y
  _    == _       = False
instance Ord Nat where
  Zer <= _        = True
  Suc _ <= Zer    = False
  Suc x <= Suc y  = x <= y
```

Natural numbers are enumerable. Also, they form a monoid, because an associative addition operator with zero as identity can be defined.

```
instance Enum Nat where
  enumFrom n = n : enumFrom (Suc n)
instance Monoid Nat where
  zzero      = Zer
  Zer <+> y  = y
  Suc x <+> y = Suc (x<+>y)
```

Natural numbers do not form a group, because the set is not closed under negation. The function `natMin` is a partial minus function, using the `Maybe` datatype to indicate failure.

```
data Maybe a = No | Yes a
natMin :: Nat -> Nat -> Maybe Nat
natMin x Zer      = Yes x
natMin Zer _      = No
natMin (Suc x) (Suc y) = natMin x y
```

A total multiplication function can be defined. Nevertheless, natural numbers do not form a ring, as this requires that they form a group, too. Similarly, a division function can be defined, but this does not make the natural numbers an Euclidean space.

```
natMul :: Nat -> Nat -> Nat
natMul Zer _ = Zer
natMul (Suc x) y = y <+> natMul x y
natDiv :: Nat -> Nat -> Nat
natDiv x y = case natMin x y
              of No    -> Zer
                 Yes d -> Suc (natDiv d y)
```

3.3 Integers

We will define integers as either negative or positive natural numbers. This gives us two representations for zero. For symmetry, we add a third representation for zero explicitly, and assume that `Neg` and `Pos` are never applied to the natural number `Zer`.

```
data Integ = Neg Nat
           | Zero
           | Pos Nat
```

Integers can be compared and ordered:

```
instance Eq Integ where
  Neg x == Neg y = x==y
  Zero  == Zero  = True
  Pos x == Pos y = x==y
  _     == _     = False
instance Ord Integ where
  Neg x <= Neg y = x>=y
  Neg _ <= _     = True
  Zero <= Neg _  = False
  Zero <= _     = True
  Pos x <= Pos y = x<=y
  Pos _ <= _     = False
```

For addition of integers with the same sign, addition of the natural number after the sign can be used. If the signs differ, we make use of the fact that the sign constructors are applied to non-zero natural numbers only.

```
instance Monoid Integ where
  zzero = Zero
  Zero <+> y = y
  Neg x <+> Neg y = Neg (x<+>y)
  Pos x <+> Pos y = Pos (x<+>y)
  Neg (Suc Zer) <+> Pos (Suc Zer) = Zero
  Neg (Suc Zer) <+> Pos (Suc y) = Pos y
  Neg (Suc x) <+> Pos (Suc Zer) = Neg x
  Neg (Suc x) <+> Pos (Suc y) = Neg x <+> Pos y
  x <+> y = y <+> x
```

Integers are a group, with negation just changing the sign. In the definition of multiplication, multiplication of natural numbers is augmented with the sign rule. Once we have defined a `one`, we can define enumeration as repeatedly adding one.

```
instance Group Integ where
  neg Zero = Zero
  neg (Neg x) = Pos x
  neg (Pos x) = Neg x
instance Rng Integ where
  Zero <*> y = y
  Neg x <*> Neg y = Pos (natMul x y)
  Pos x <*> Pos y = Pos (natMul x y)
  Neg x <*> Pos y = Neg (natMul x y)
```

```

x    <*> y    = y <*> x
instance Ring Integ where
  one      = Pos (Suc Zer)
instance Enum Integ where
  enumFrom n      = n : enumFrom (n<+>one)

```

Finally, integers form an Euclidean space. Integer division is natural division augmented with a sign rule; the degree function is the absolute value, defined by a fold over the natural number following the sign. The modulo function was already defined with a default definition in section 2.

```

instance Euclid Integ where
  degree Zero      = 0
  degree (Pos n)   = foldNat 0 (+1) n
  degree (Neg n)   = foldNat 0 (+1) n
  divide Zero y    = Zero
  divide (Pos x) (Pos y) = Pos (natDiv x y)
  divide (Pos x) (Neg y) = Neg (natDiv x y)
  divide (Neg x) (Pos y) = Neg (natDiv x y)
  divide (Neg x) (Neg y) = Pos (natDiv x y)

```

3.4 Quotients

From the integers, rational numbers can be constructed by taking pairs modulo an equivalence relation. The rational numbers form a field. A generalization of this is the construction of the quotient field over an arbitrary ring a .

```

data Quot a = Quot (a,a)

```

We will assume that the second part of the tuples are not zero. Two quotients are equivalent, and thus denote the same fraction, if cross multiplication yields equal results:

```

instance Rng a => Eq (Quot a) where
  Quot (a,b) == Quot (c,d) = a<*>d == c<*>b

```

For being able to multiply, we need the base type to be a ring, although this ring needs not to have a 'one'. If the base ring can be ordered, then so can be quotients:

```

instance (Ord a,Rng a) => Ord (Quot a) where
  Quot (a,b) <= Quot (c,d) = a<*>d <= c<*>b

```

An associative addition can be defined for quotients, by making a common denominator:

```

instance Ring a => Monoid (Quot a) where
  zzero = Quot (zzero, one)
  Quot (a,b) <+> Quot (c,d) = Quot (a<*>d <+> c<*>b, b<*>d)

```

For the zero element, we need a non-zero denominator. The `one` of the base ring is the only element guaranteed to be non-zero, so we need the base ring to have a unit for making quotients to be a monoid. The type information given is sufficient to deduce that by `zzero`, `one`, `<+>` and `<*>` on the right hand side of the definitions, the operations of the base ring are denoted.

Finally, quotients can easily be made into a ring and a field:

```

instance Ring a => Rng (Quot a) where
  Quot (a,b) <*> Quot (c,d) = Quot (a<*>c, b<*>d)
instance Ring a => Ring (Quot a) where
  one = Quot (one, one)
instance Ring a => Field (Quot a) where
  inv (Quot (a,b)) = Quot (b,a)

```

3.5 Addition of i

Complex numbers can be constructed from real numbers by considering formal sums $a + b * i$, where i has the property $i * i = -1$. An generalization of this are formal sums $a + b * i$ over an arbitrary base type. The closure of adding i can be made to belong to the same class (monoid, group, rng, ring or field) as the base type. This is formalized in the following instance declarations, in which the rules for manipulating complex numbers can be recognized.

```

data Iadd a = Iadd (a,a)

instance Eq a => Eq (Iadd a) where
  Iadd (a,b) == Iadd (c,d) = a==c && b==d
instance Monoid a => Monoid (Iadd a) where
  zzero = Iadd (zzero, zzero)
  Iadd (a,b) <+> Iadd (c,d) = Iadd (a<+>c, b<+>d)
instance Group a => Group (Iadd a) where
  neg (Iadd (a,b)) = Iadd (neg a, neg b)
instance Rng a => Rng (Iadd a) where
  Iadd (a,b) <*> Iadd (c,d) = Iadd (a<*>c <-> b<*>d, a<*>d <+> b<*>c)
instance Ring a => Ring (Iadd a) where
  one = Iadd (one, zzero)
instance Field a => Field (Iadd a) where
  inv (Iadd (c,d)) = Iadd (c</>e, neg d</>e)
  where e = square c <+> square d

```

3.6 Function spaces

Function spaces form a monoid, with function composition as ‘addition’ and the identity function as ‘zero’.

```

instance Monoid (a->a) where
  zzero = id
  (<+>) = (.)

```

Function spaces in general do not form a group, as negation should be function inversion. For some special cases, inversion can be defined. A well-known example is invertible real functions, of which the inverse can be computed by Newton’s method (adapted here from Bird and Wadler [BW88]):

```

instance Group (Float->Float) where
  neg g a = until satisfied improve 1.0
  where satisfied b = f b ~= 0.0
        improve b = b - f b / f' b

```

```

f x    = g x - a
a~b    = a-b<h && b-a<h
f' x   = (f (x+h) - f x) / h
h      = 0.00001

```

Also, functions on a finite domain are invertible, by trying all elements, and thus form a group:

```

instance (Eq a, Finite a) => Group (a->a) where
  neg g a = head [ x | x<-members, g x==a ]

```

To provide some finite types, we define some instances of the `Finite` class: booleans, characters, pairs of finite types, and function spaces between finite types:

```

instance Finite Bool where
  members = [False, True]
instance Finite Char where
  members = map chr [0..127]
instance (Finite a, Finite b) => Finite (a,b) where
  members = [ (x,y) | x<-members, y<-members ]
instance (Eq a, Finite a, Finite b) => Finite (a->b) where
  members = funspace members members
funspace :: Eq a => [a] -> [b] -> [a->b]
funspace [x] ys = map const ys
funspace (x:xs) ys = [ \p->if p==x then y else f p
                      | y <- ys
                      , f <- funspace xs ys
                      ]

```

As a condition for forming a monoid and group, we need a type to be instance of `Eq`. For function spaces this is a problem: it can only be done constructively for finite domains, using the Leibnitz property that functions are equal if they are equal for all arguments:

```

instance (Finite a, Eq b) => Eq (a->b) where
  f == g = and [ f x==g x | x<-members ]

```

Because these definitions are applied recursively, we are now capable of e.g. verifying De Morgan's law by evaluating

```

(\x y -> not(x&& y)) == (\x y->not x|| not y)

```

Equality is only required for the monoid and group classes for being able to formulate laws; it is not used in computations. If we are only interested in computing, and not in mechanically checking laws, we may give a dummy implementation for function equality.

3.7 Matrices

Matrices are blocks of values. Often, the values of a matrix are taken to be real or complex numbers, but here we will allow for matrices over an arbitrary type. Depending on the class of the base type, matrices can be made an instance of the same class. Matrices are represented as lists of their rows:

```

data Matrix a = Mat [[a]]

```

We will assume some constraints on the size of matrices. Firstly, all rows should have the same length. In the class `Monoid`, for addition of two matrices, they should have the same size. In the class `Ring`, for multiplication of matrices, the number of rows of the first operand should be equal to the number of columns of the second. In the class `Field`, for inverting matrices, we consider only square invertible matrices.

For some operations on matrices, a function is applied to all its elements. For this, we make auxiliary functions which are a 2-dimensional analogue to `map` and `zipWith`:

```
mapp      :: (a->b)    -> [[a]] -> [[b]]
zipWith  :: (a->b->c) -> [[a]] -> [[b]] -> [[c]]
mapp     = map      . map
zipWith  = zipWith . zipWith

pointwise1 f (Mat xss)          = Mat (mapp f xss)
pointwise2 op (Mat xss) (Mat yss) = Mat (zipWith op xss yss)
```

For example, scaling a matrix by a constant value, the multiplication is applied to all its elements, and equality of matrices is the conjunction of equality of all respective elements:

```
scalemat :: Rng a => a -> Matrix a -> Matrix a
scalemat x = pointwise1 (x<*>)
instance Eq a => Eq (Matrix a) where
    Mat xss == Mat yss = all and (zipWith (==) xss yss)
```

Matrices can be added by adding their elements pointwise. As a zero element, we take an infinite matrix of zeroes, which can be added or compared to a matrix of arbitrary size:

```
instance Monoid a => Monoid (Matrix a) where
    zzero = Mat (repeat (repeat zzero))
    (<+>) = pointwise2 (<+>)
instance Group a => Group (Matrix a) where
    neg = pointwise1 neg
```

In textbooks, multiplication of matrices is often described using a notation with indices. Indices are however seldomly necessary in a functional program, and indeed matrix multiplication can be described using standard list operations:

```
instance Rng a => Rng (Matrix a) where
    Mat xss <*> Mat yss = Mat (map f xss)
    where f a = map (inprod a) (transpose yss)
          inprod xs ys = summ (zipWith (<*>) xs ys)
```

If `transpose` were not defined in the prelude, here is a nice definition:

```
transpose = foldr (zipWith (:)) (repeat [])
```

The unit matrix is a matrix with ones on its diagonal and zeroes elsewhere.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \\ \vdots & & & & \ddots \end{pmatrix}$$

As in the case of the zero matrix, we make it an infinite matrix, which will size down to the right size when multiplied by another matrix:

```
instance Ring a => Ring (Matrix a) where
  one = Mat (iterate (zzero:) (one:repeat zzero))
```

An important notion for matrices is its *determinant*, which can be computed if the base type is a ring. The determinant can be computed recursively by multiplying the elements of the first row with sub-matrices that are obtained by deleting the first row and the corresponding column, and adding the products with alternating signs. For example, in the 3×3 case:

$$\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{matrix} +a \times \det \begin{pmatrix} e & f \\ h & i \end{pmatrix} \\ -b \times \det \begin{pmatrix} d & f \\ g & i \end{pmatrix} \\ +c \times \det \begin{pmatrix} d & e \\ g & h \end{pmatrix} \end{matrix}$$

This is formalized in the following definition:

```
det :: Ring a => Matrix a -> a
det (Mat [[x]]) = x
det (Mat (xs:xss)) = ( altsum
  . zipWith (<*>) xs
  . map (det.Mat)
  . gaps
  . transpose
  ) xss
```

The auxiliary function `gaps` deletes one element from a list in all possible ways, and the function `altsum` calculates a sum with alternating signs:

```
gaps      :: [a] -> [[a]]
gaps []   = [ ]
gaps (x:xs) = xs : map (x:) (gaps xs)

altlist  :: Ring a => [a]
altlist  = one : neg one : altlist
altsum   :: Ring a => [a] -> a
altsum   = summ . zipWith (<*>) altlist
```

For square matrices with non-zero determinant, an inverse can be calculated by Cramer's rule. For calculating an element of the inverse matrix, its row and column should be deleted from the transpose of the original matrix, its determinant calculated, and divided by the determinant of the entire original matrix. Again, the signs should be taken alternatingly. See figure 1 for a 3×3 example. It is easier to express this rather awkward description using a function composition:

$$\text{inv} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \frac{\begin{pmatrix} +\det \begin{pmatrix} \alpha & \alpha k & g \\ b & e & h \\ \alpha f & i & \end{pmatrix} & -\det \begin{pmatrix} \alpha & d & g \\ b & \alpha k & k \\ \alpha f & i & \end{pmatrix} & +\det \begin{pmatrix} \alpha & d & g \\ b & e & h \\ \alpha f & i & \end{pmatrix} \\ -\det \begin{pmatrix} \alpha & \alpha k & g \\ b & \alpha k & h \\ c & f & i \end{pmatrix} & +\det \begin{pmatrix} a & \alpha k & g \\ b & \alpha k & k \\ c & f & i \end{pmatrix} & -\det \begin{pmatrix} a & \alpha k & g \\ b & \alpha k & h \\ \alpha f & i & \end{pmatrix} \\ +\det \begin{pmatrix} \alpha & \alpha k & g \\ b & e & k \\ c & f & i \end{pmatrix} & -\det \begin{pmatrix} a & d & g \\ b & \alpha k & k \\ c & f & i \end{pmatrix} & +\det \begin{pmatrix} a & d & g \\ b & e & k \\ \alpha f & i & \end{pmatrix} \end{pmatrix}}{\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}}$$

Fig. 1. Calculating the inverse of a matrix

```
instance Field a => Field (Matrix a) where
  inv m@(Mat xss) = ( Mat
    . zipWith (<*>) altmat
    . mapp ((</>(det m)).det.Mat)
    . map (gaps.transpose)
    . gaps
    . transpose
  ) xss
  altmat  :: Ring a => [[a]]
  altmat  = iterate tail altlist
```

The function `map(gaps.transpose).gaps` generates a matrix of matrices, where rows and columns are left out in all possible ways.

3.8 Polynomials

Polynomials are formal finite sums of terms, where each term is a value multiplied with a variable raised to a natural power. We will represent polynomials by lists of terms, where each term is a pair of a coefficient and an exponent:

```
type Term a = (a,Int)
data Poly a = Poly [Term a]
```

some auxiliary functions manipulate terms and simple polynomials:

```
coef    :: Term a -> a
expo    :: Term a -> Int
psingle :: a -> Int -> Poly a
pconst  :: a -> Poly a
hdcoef  :: Poly a -> a
coef    = fst
expo    = snd
psingle a n = Poly [(a,n)]
pconst a   = psingle a 0
hdcoef (Poly xs) = coef (head xs)
```

We will assume that polynomials are always in *normal form*, that is: no terms have zero coefficient, all exponents are unique, and terms are sorted with decreasing exponent. The following function brings a polynomial in normal form. It uses addition of coefficients, and thus requires the coefficients to form a monoid:

```
pnorm :: Monoid a => [Term a] -> [Term a]
pnorm = filter ((/=zzero).coef) . puniq . sort
  where sort          = foldr ins []
        ins x []      = [x]
        ins x (y:ys) | expo x > expo y = x:y:ys
                    | otherwise       = y:ins x ys
        puniq []      = []
        puniq [t]     = [t]
        puniq (x@(a,n):xs@((b,m):ys))
            | n==m     = puniq ((a<+>b,n):ys)
            | otherwise = x : puniq xs
```

Terms can be compared, negated and multiplied, provided that the coefficients can be. Terms can not be added, however, and thus don't form a monoid.

```
teq :: Eq a => Term a -> Term a -> Bool
teq (a,n) (b,m) = n==m && a==b
tneg :: Group a => Term a -> Term a
tneg (a,n)      = (neg a, n)
tmul :: Rng a => Term a -> Term a -> Term a
tmul (a,n) (b,m) = (a<*>b, n+m)
```

Polynomials can be made instance of the same classes (eq, monoid, group, rng and ring) as their base type. Addition is done by concatenating all terms and normalizing the result, negation is done termwise, and multiplication is done by taking the cross product of all terms:

```
instance Eq a => Eq (Poly a) where
  Poly xs == Poly ys = length xs==length ys && and(zipWith teq xs ys)
instance Monoid a => Monoid (Poly a) where
  zzero          = Poly []
  Poly xs <+> Poly ys = Poly (pnorm (xs++ys))
instance Group a => Group (Poly a) where
  neg (Poly xs) = Poly (map tneg xs)
instance Rng a => Rng (Poly a) where
  Poly xs <*> Poly ys = Poly (pnorm [ tmul x y | x<-xs, y<-ys])
instance Ring a => Ring (Poly a) where
  one = pconst one
```

A more interesting instance is that polynomials can be made an Euclidean space, that is division and remainder can be defined, provided that the base type is a field. As 'degree' function we can take the highest exponent, and -1 for the zero polynomial. An example is the division:

$$\frac{2x^4 + 5x^3 + 4x^2 - 3x + 2}{x^2 + x + 1}$$

The quotient of this division $2x^2 + 3x - 1$, the remainder is $-5x + 3$. It can be calculated by a kind of ‘long division’:

$$\begin{array}{r}
 x^2+x+1 \overline{) 2x^4+5x^3+4x^2-3x+2} \setminus 2x^2+3x-1 \\
 \underline{2x^4+2x^3+2x^2} \\
 3x^3+2x^2-3x \\
 \underline{3x^3+3x^2+3x} \\
 -x^2-6x+2 \\
 \underline{-x^2-x-1} \\
 -5x+3
 \end{array}$$

To start with, the head term of the numerator ($2x^4$) and the head term of the denominator (x^2) are inspected. They are divided (this is why the base type needs to be a field), which results in the first term of the quotient ($2x^2$). This term is multiplied with the denominator ($x^2 + x + 1$). The product ($2x^4 + 2x^3 + 2x^2$) is subtracted from the numerator, and thus the term with the highest exponent ($2x^4$) vanishes. The process is repeated with the remaining part of the numerator ($3x^3 + 2x^2 - 3x + 2$), which yields the second term of the quotient ($3x$). The recursion stops when the remaining numerator has lower degree than the denominator, which is required for the remainder. The algorithm is formalized in:

```

instance Field a => Euclid (Poly a) where
  degree (Poly [])      = -1
  degree (Poly xs)     = expo (head xs)
  f 'divide' g | n<m   = zzero
                  | otherwise = (f <-> h<*>g) 'divide' g <+> h
    where n = degree f
          m = degree g
          h = psingle (hdcoef f</>hdcoef g)(n-m)

```

Now with the derived function `gdDiv` from section 2.2 we can calculate the gcd of polynomials over an arbitrary field.

4 Parsers

We will make use of Gofers’ constructor classes `prelude` and of a monadic parser library, inspired by Wadler [Wad95]. In that library, the type `Parser` is made instance of `Functor`, `Monad`, `MonadPlus` and `MonadZero`, so that monad comprehensions (see [Jon94]) can be used for defining parsers. In the library, some common parser combinators are defined, like

```

satisfy :: (Char->Bool) -> Parser Char
many    :: Parser a -> Parser [a]
option  :: Parser a -> Parser [a]

```

Using the library, a parser for a single digit can be defined using a monad comprehension:

```

digit :: Parser Int
digit = [ ord x - ord '0'
        | x <- satisfy isDigit
        ]

```

From this, a parser for the type `Nat` of (Peano-like) natural numbers can be defined as:

```
instance Repr Nat where
  parse = [ foldl f zzero ds | ds <- sequence digit ]
  where f a b = a 'natMul' g 10 <+> g b
        g n = iterate Suc zzero !! n
  unpars x = shows (f x)
  where f Zer = 0
        f (Suc n) = 1 + f n
```

More involved parsers can be build by combining parsers using combinators and monad comprehensions. Unparsers can be combined by functional composition. For example, integers are represented as:

```
instance Repr Integ where
  parse = [ if n==Zer
            then Zero
            else (if null s then Pos else Neg) n
            | s <- option (symbol '-')
            , n <- parse
            ]
  unpars Zero = showChar '0'
  unpars (Pos n) = unpars n
  unpars (Neg n) = showChar '-' . unpars n
```

Similarly, a parser for floating point numbers can be defined (see [Fok95]).

For the parameterized types, like `Quot` and `Iadd`, (un)parsers can be defined by calling the overloaded `parse` function for the base type.

```
instance (Ring a, Repr a) => Repr (Quot a) where
  parse = [ Quot (n, if null d then one else head d)
            | n <- parse
            , d <- option (second (symbol '/')) parse
            ]
  unpars (Quot (x,y))
    | y==one = unpars x
    | otherwise = unpars x . showChar '/' . unpars y
```

A term of a polynomial can be represented by strings like $3x^2$. This format (where the coefficient defaults to one, the exponent to 1, unless the x is missing, in which case it is 0) is parsed and generated by:

```
instance (Ring a, Repr a) => Repr (Term a) where
  parse = [ ( if null c then one else head c
            , if null e then 1 else head e
            )
            | c <- option parse
            , x <- option(token "x")
            , e <- if null x
                  then result [0]
                  else option (second (token "^")) parse
            ]
```

```

unpars (c,e)
  | e==0 = unpars c
  | otherwise
    = (if c==one && e>0 then id else unpars c)
      . (if e==0 then id else showChar 'x')
      . (if e==1 then id else showChar '^' . unpars e)

```

Then, parsers for polynomials are easily constructed using the parser combinator `listOf`, and the dual unparsing combinator `listify`:

```

instance Repr (Term a) => Repr (Poly a) where
  parse = [ Poly xs | xs <- listOf parse (token "+") ]
  unpars (Poly xs) = listify unpars (showString " + ") xs

```

where

```

listify :: (a -> ShowS) -> ShowS -> [a] -> ShowS
listify f s [] = id
listify f s [x] = f x
listify f s (x:xs) = f x . s . listify f s xs
listOf :: Parser a -> Parser () -> Parser [a]
listOf p s = [ a:as
              | a <- p
                , as <- many [ c | _ <- s, c <- p ]
              ]

```

A parser for matrices can be constructed in a similar way.

Two auxiliary functions are defined for parsing and showing a single item, by making an initial call to the functions `parse` and `unpars`:

```

pars  :: Repr a => String -> a
shw   :: Repr a => a -> String
pars  = head . just parse
shw x = unpars x ""

```

5 Applications

The construction of the quotient field over an arbitrary ring in section 3.4 was a generalization of the construction of the rational numbers. Therefore, the rational numbers can be defined as a special case of it:

```

type Rat      = Quot Int

```

Likewise, complex numbers can be defined as taking the closure of adding i to the floating point numbers

```

type Compl    = Iadd Float

```

By using `Iadd` over other base types, we get Gauss' integer numbers, or Hamilton's quaternions:

```

type Gauss    = Iadd Int
type Qi       = Iadd Rat
type Quaternion = Iadd Compl
type Octonion = Iadd Quaternion

```


For example, the eigenvalues of

```
m2 :: Matrix Int
m2 = pars "(1 2 3 4|5 6 7 8|9 10 11 12|13 14 15 16)"
```

are solutions of

```
? shw (eigenPoly m2)
x^4 + -34x^3 + -80x^2
```

6 Conclusion

We have shown that structures from abstract algebra can be successfully modelled as Gofer classes. By providing parsers for constructions that are instances of these classes, the algorithms can be applied immediately to practical problems.

This work could be extended in several directions, e.g.:

- Adding more classes, e.g. vector spaces and modules, or algebraically closed fields.
- Proving that the required laws hold for the implementations given, in the style of the induction proofs in [BW88].
- Adding more instances, e.g. lists as a monoid with `++` as addition and `[]` as zero, or Klein’s 4-group as a group, or symbolic expressions as a field.
- Adding more derived operations for the classes that were defined. For example, find the solution of a polynomial. One could use the *abc* formula for polynomials of degree 2, and Cardano’s formulas for degree 3 and 4 (making use of polynomials over complex numbers), or use special strategies (for polynomials over `Rat`, every solution has the form b/c , where b is a divisor of the last coefficient, and c is a divisor of the head coefficient). Or program algorithms for handling matrices, e.g. finding the Jordan normal form.

Appendix

A Functional Programming Education in Utrecht

The material treated in this article is part of a first course on functional programming for computer science students in Utrecht. This course assumes no previous programming experience, and is taught *in parallel* with a course on imperative/OO programming (putting an end to the debate which paradigm to teach first).

After this introductory course, functional programming is used throughout the computer science curriculum (which takes 8 semesters). The course sequence is coordinated by Doaitse Swierstra. By using functional programming in teaching various subjects, the student’s fluency in functional programming is maintained and deepened. At the same time, all these subjects benefit from the concise formulation of algorithms made possible through the use of functional languages.

All courses have lectures, tutorials and lab assignments for approximately two hours once a week, and last 10 to 15 weeks. Students are assessed by a written exam and by their lab works. The courses involved, with their workload in hours and the literature used, are:

course	semester	lecture	tutorial	lab	homew.	literature
Functional Programming	1	20	20	40	40	[Fok92]
Grammars and Parsing	2	30	30	40	60	[JDS94]
Implementation of Prog.Lang.	3	20	20	80	40	[Meij95]
Advanced Functional Progr.	6	30	0	60	30	[JM95]
Partial Evaluation	7	30	0	0	90	[JGS93]
Category Theory	7	30	0	0	90	[Mee95]

The specialization courses in semester 5–8 are optional. Furthermore, there is a connection with the course on User Interfaces (semester 6). Unfortunately, the course on Linear Algebra (semester 4) is not integrated in the course sequence. Also, mathematics students do not participate.

References

- [BW88] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall, 1988.
- [Fok92] Jeroen Fokker, *Functional programming*. Course notes, Department of Computer Science, Utrecht University, 1992. Also available as *Functioneel Programmeren* and as *Programación Funcional*. From <http://www.cs.ruu.nl/~jeroen>.
- [Fok95] Jeroen Fokker, ‘Functional parsers’. In [JM95], pp. 1–23. Also <http://www.cs.ruu.nl/~jeroen>.
- [JDS94] Johan Jeurig, Luc Duponcheel and Doaitse Swierstra, *Grammars and Parsing*. Course notes, Department of Computer Science, Utrecht University, 1994. From <http://www.cs.ruu.nl/~luc>.
- [JM95] Johan Jeuring and Erik Meijer (eds.), *Advanced functional programming*. Springer LNCS 925, 1995.
- [Jon94] Mark Jones, *Gofer 2.30 release notes*. <http://www.cs.nott.ac.uk/Department/Staff/mpj>.
- [JGS93] Neil Jones, Carsten Gomard and Peter Sestoft, *Partial evaluation and automatic program generation*. Prentice Hall, 1993.
- [Lan70] Serge Lang, *Linear Algebra*. Addison-Wesley, 1970.
- [Mee95] Lambert Meertens, *Category theory*, Course notes, Department of Computer Science, Utrecht University, 1995. Ask lambert@cs.ruu.nl.
- [Meij95] Erik Meijer and Doaitse Swierstra, *Implementation of Programming Languages*. Course notes, Department of Computer Science, Utrecht University, 1995. From <http://www.cs.ruu.nl/~erik>.
- [Wad95] Philip Wadler, ‘Monads for functional programming’. In [JM95], pp. 24–52. Also: <http://www.dcs.glasgow.ac.uk/~wadler>.