Functional Programming and Mathematical Objects

Jerzy Karczmarczuk

Dept. of Computer Science, University of Caen, France

Abstract. We discuss the application of the Haskell/Gofer type classes and constructor classes to the implementation and teaching of the manipulation techniques in the domain of formal mathematical expressions. We show also how the lazy evaluation paradigms simplify the construction and the presentation of several algorithms dealing with iterative data types, such as power series or formal Padé expansion. We show the application of higher order functions to algebra and geometry, and specifically — to the construction of parametric surfaces.

1 Introduction

Formal manipulation of algebraic and geometric objects seems *à priori* to be a wonderful training ground for mathematically oriented students who want to learn the design and the implementation of complex algorithms and heterogeneous data structures. It is not very often exploited, though. While it is standard at quite elementary level, to teach formal differentiation of algebraic tree structures, or the arithmetic of compound data structures such as complex numbers or rationals, it is not very easy to leave the *syntactic* approach to the modelling of the mathematical objects, and to make use of the *structure* (in the mathematical, rather than programming) sense of the underlying domain.

The students learn easily how to operate on fractions and polynomials, but the algebra of rational functions has usually to be implemented again, without code reusing. Standard list or term manipulation languages, as Lisp or Prolog lack the inheritance concept. But the classical object-oriented systems, such as Smalltalk or Common Lisp Object System are too heavy, and they exploit intensely the *data inheritance*, which might not be very useful if we just need to assert that both polynomials and matrices belong to a Ring category, and that from a Ring and some additional properties we can construct effectively the field of quotients, with the same (in principle) algorithms in the case of rational numbers and ratios of polynomials.

The necessity to formulate the computer algebra algorithms and data structures in a mathematically structured way was recognized many years ago. The system Axiom was built explicitly upon this principle, and the Maple package is distributed now with the public library Gauss[1] – an "object-oriented" sublanguage which heavily uses inheritance. Their learning curves are unfortunately quite steep, and their pedagogical influence is rather weak, as too many details are hidden in "black boxes".

This paper is devoted to some aspects of teaching of programming to students of Mathematics and Computer Science. We decided to look at the problem mentioned above from the perspective of modern, typed lazy functional programming. The relation between types – as realized in the language Haskell[2] – and the mathematical domains, will be discussed in the section 3. With lazy streams it is easy to create potentially infinite data structures such as series, continuous fractions, etc. But there are more arguments as well: thanks to the deferred evaluation and higher order functions it is easier to formulate the algorithms in a static, declarative manner, without polluting them with countless **for/while** loops and other imperative constructs, which hide sometimes the clarity of the underlying strategy. We have chosen Gofer[3], a popular dialect of Haskell, which permits the redefinition of the *standard prelude* where the properties of the standard operators are specified.

We wanted to bridge the gap between formal descriptions of the properties of mathematical operations, and their *practical* implementation, and also to show the students *how* this is done. So, we have designed a simplified, but quite elaborate mathematical hierarchy of classes in Gofer, and we have applied it to numerous, small, but not always trivial examples of algorithms, some of which one might not easily find in standard textbooks. This talk presents partially the work done. We have consciously omitted all computer algebra "classics" such as the algebra of symbolic extensions. The presentation below is not self-contained, we assume some knowledge of Haskell, and of standard algebra. Our aim was threefold:

- To popularize advanced functional techniques among experienced students too conditioned by the object-oriented paradigms, or "handicapped" a little by imperative programming languages.
- To show how a practical, effective programming language can be used as a tool to construct quite abstract mathematics.
- To analyse the possible traps and inadequacies of the current polymorphic functional languages, in order to propose one day a more powerful machinery.

Several structure definitions and algorithms have been reconstructed together with the students, and it was a real pleasure seeing how fast they could recover from all possible bugs and pitfalls thanks to the clarity of the language used.

2 Pedagogical Context

This paper is *not* based on *one* homogeneous course. The techniques presented here have been taught to 4th year (Maîtrise) students of Mathematics and Mathematical Engineering during a one-semester course on Functional Programming, and offered also to students of Computer Science, (4th year) as a complementary material for the course on Image Synthesis. Some of the techniques discussed below have been presented on a seminar attended by the students of DEA (Diplôme d'Études Approfondies; 5th year) on Computer Science, and have been applied to graphic modelling on a DEA stage. The 4th year courses were accompanied by practical exercices (3 hours per week) and were assessed by a written examination, and by one or two programming projects demanding a few weeks of work.

So, the audience was rather advanced. All students were acquainted with the essentials of list processing, and heard about lazy evaluation semantics (realized in Scheme). Unfortunately, for historical reasons, Gofer could not be used as a principal programming language, the course on Functional Programming was essentially based on CAML. The implementation of algorithms in Gofer was proposed informally. Nothing was really enforced, and the success of the project could be estimated by the enthusiasm of the best students, who did not restrict themselves to obligatory topics.

Thus, neither our teaching of the lazy techniques and hierarchical polymorphism, nor this paper stress upon the strictly pedagogical issues, but — as mentioned above — concentrates on a methodology of construction of mathematical objects. We used the implementations of Gofer and CAML running on Sparc stations, and in DOS boxes under MS-Windows.

3 Modified Classes of Types in Gofer

One of the powerful mechanisms in the programming language Haskell is the concept of *type classes* whose aim is to control selectively the operator overloading. Such overloading is an old concept. Everybody knows that the mathematical sign `+` is used to represent the addition of integer and real numbers in almost all popular programming languages. Sometimes the same symbol denotes the concatenation of character strings. However, in order to add two matrices by writing just `A + B` one needs an extensible language, such as the powerful object-oriented machinery of C++. For any new type of objects, one has to implement the new addition operation completely ad hoc, and it not easy to ensure some uniformity of the generalized arithmetic domain.

The creators of Haskell decided to base the polymorphism on the existence of common operational properties of a *class* of different, possibly heterogeneous data types. For example, the class `Ord a` declares the relation of order `<=`, and specifies that for all objects belonging to the type `a`, and only for them, such relation exists. Later we declare that some type, for example the field of rational fractions: pairs `(Num,Den)` is an *instance* of the class `Ord`, and we define explicitly an appropriate order relation. If, as it is usual – the class `Ord` is a subclass of the class `Eq` which encompasses all data types with the equivalence relation defined, there is no need for other order relations, all can be derived from `<=` by inversion and/or composition with the inequality.

It was exactly what we needed — to be able to define abstract mathematical operations such as additions, independently of the data structures representing the added objects. We found it methodologically harmful that our students engraved too deeply in their memories that a complex number is essentially a record with two real fields, and other similar “truths”.

But the arithmetic operators in standard Haskell are essentially numeric, and their genericity is restricted to integers, floats, etc. There are definitions of rational fractions and complex numbers in the standard preludes, but the possibility of introducing more general data categories in this context is largely unexploited. The question has been posed already several times: why not provide a more general class, say, `AdditiveGroup` where the operation `(+)` would be declared, another class `SemiGroup` defining the multiplication, etc. There are some theoretical problems, for example the `AdditiveGroup` is also a `SemiGroup`, so it should somehow inherit something from it, but it cannot: the system of type classes in Haskell states that a given operation *exists*, and not that it *has some properties.* We cannot declare within the `SemiGroup` *one* neutral element for both arithmetic operations. In order to do this, we would have to declare the operations `(+)` and`(*)` as having polymorphic types belonging to two instances of the same class and that would need a system of functional meta-classes which is currently not implemented in any known functional language.

Moreover, the standard Haskell system is too rigid. For example, one cannot specify that a given type belongs to a `VectorSpace`, as this would imply the engagement of at least two usually distinct types: the additive group of vectors and the field of scalars, and this demanded a multi-parameter class. But this was exactly our aim: to take a standard textbook on algebra and to show to our students that not only polynomials are implementable, but the Galois fields, or Rings as well.

Fortunately the Haskell-like language Gofer is more flexible, although according to its author, the multi-parameter classes are a can of worms for the type-checking system. But for pedagogical purposes this constructive approach to mathematics is very fruitful, and the question of the kind "why is this definition ambiguous?" is not only a code debugging exercise, but provides an insight into the coherence of the defined mathematical structure. So, we have chosen Gofer as our battle horse, and we have reconstructed the algebraic layer of its automatically loaded *standard prelude.*

The above mentioned dilemma with having two different group structures in a `Ring` could not be solved satisfactorily, so we defined independently the additive and the multiplicative group. In order not to multiply the number of rarely used classes, we started already with semi-groups with unity. Here is the beginning of the hierarchy, we assume that the reader is acquainted with the syntax of Haskell:

```
class Monoid a where
  groupOne :: a;     (*) :: a -> a -> a
  powerInt, (^) :: a -> Int -> a  -- Pos. expt.
  negExp :: a -> Int -> a

  x `powerInt` n = itbin (*) x n groupOne
  negExp _ _ = error "Ring: negative exponent"
  x ^ n | n >= 0 = x `powerInt` n
        | otherwise = negExp x (-n)
```

```
class Monoid a => Group a where
  (/) :: a -> a -> a
  recip :: a -> a

  recip x = groupOne / x
  x/y = x * recip y        -- Beware! Cyclic defs!

class AddGroup a where
  addgroupZero :: a;
  (+), (-), subtract :: a -> a -> a
  negate, double :: a -> a
  (#) :: Int -> a -> a     -- Mult. by integer

  negate x = addgroupZero - x
  x - y = x + negate y     -- Beware! cyclic!
  subtract = flip (-)
  n # x | n<0 = negate ((-n) # x)
        | otherwise = iterbin (+) x n addgroupZero
```

The idea was to choose the minimal set of objects which characterized these structures, but also to deduce some secondary operations existing by default, such as the multiplication by integer if the addition was defined. The essential point – not always understood – is that all the *definitions* above are effective pieces of program, not just specifications. They are *default* definitions which hold for any type, unless overrided by the instances. Later on, when we defined the data type `Zp` of modular integers, which for prime p is a field, everybody recognized easily that the most evident implementation of the division x/y was the multiplication of x by the inverse of y. It suffices thus to define the latter. On the other hand, the reciprocal of a power series is just a particular case of the division, so we define the division as presented in the next section, and we keep the default for the function `recip`. Obviously, the operation `n # x` for `x` belonging to integers was reimplemented through hardware primitives. We introduced also some standard and less standard polymorphic combinators such as `flip` which switches the order of the arguments of a given function, or `iterbin` which iterates an associative binary operation, and we have shown that they provide the necessary implementation "glue". The students were supposed to discover the analogy between the integer power as the iterated multiplication, and the multiplication by an integer as an iterated addition, and to propose an abstract iterator which used the binary splitting of the integer:

```
iterbin op = g where
   g x 1 = op x
   g x 0 = id
   g x n | even n    = p
         | otherwise = p . (op x)
           where p = g (op x x) (n `div` 2)
```

The construction of `Ring` was a small non-trivial discovery: having at our disposal both the *unity* and the *zero*, we could construct an abstract conversion function `fromInteger` which mapped $\mathbb{N}$ to *any* Ring. It suffices to define `fromInteger n` as `n # groupOne`.

We have discovered also a dilemma: how to define x^n for *any* integer n? The problem is that for a positive n it is just an iterated multiplication, so it should be defined within the `Monoid`, but for a negative n the base x should belong to a Group in order to compute its inverse. We cannot *redefine* the operation `(^)` inside the `Monoid` subclasses, the inheritance mechanism in Haskell is really different from the classical object-oriented paradigms. But we can define the appropriate `negExp` function within the *instances* of `Monoid`, For all the fields such as Floats or Rationals this function returns $(1/x)^{(-n)}$, but for integers etc. we leave the default. This example shows a visible limitation of the type classes for our purposes.

The next steps of our creation of the mathematical world were quite straightforward. We have defined some *ordered* structures necessary to establish the existence of such functions as the predicate `negative` or the function `abs`. Later on, when we made the `Module` and `LinearSpace` the students were asked to define the absolute value of a vector, and they obviously had a little surprise, which helped them to assess the status of the concept of *norm*, and proved that our progression is not linear, that we should go back and take into account that even such simple structures as `AdditiveGroups` are already `Modules` over integers, etc.

We continued with the definition of the `DivisionRing` with such operations as `div`, `mod`, or `gcd`, and we have constructed all the typical numerical instances of our abstract classes, such as `Floats, Integers, RationalFractions`, complex numbers, etc. with all the appropriate operations. This part of the work was rather trivial, it was mainly straightforward coding of mathematical formulae, something a little optimised, see for example the book of Knuth[4]. Still, it was interesting to see how the system protects itself from an attempt of making fractions of *anything/anything*, demanding that a specific algebraic context of *anything* be respected. There is almost nothing interesting in the definition of the data structure representing a fraction `num :% den`:

```
data DivisionRing a => Ratio a = a :% a
type Rat = Ratio Int   -- classics
```

apart from the fact that the type of the numerator and the denominator is *statically* restricted. The students did appreciate the fact that an attempt to operate on "fractions" composed of floats or strings did not result in some execution error, but such "fractions" could not be constructed, were statically rejected, being mathematically ill-defined structures. When we try to define the addition of two generic fractions declaring that the type `Ratio a` belongs to an additive group, we have to provide a detailed algebraic context for the type `a`:

```
instance (AddGroup a, Monoid a, DivisionRing a) =>
  AddGroup (Ratio a) where
```

```
addgroupZero = addgroupZero :% groupOne
(n1:%d1)+(n2:%d2) =
let g1 = gcd d1 d2 in
 if g1==ringOne
  then (n1*d2+d1*n2) :% (d1*d2)
  else let t = n1*(d2 `div` g1)+n2*(d1 `div` g1)
           g2 = gcd t g1
       in  (t `div` g2) :%
           ((d1 `div` g1)*(d2 `div` g2))
```

The detailed code of the addition is irrelevant here, this is the optimized algorithm presented in the book of Knuth. What is interesting, is the genericity of the construction. The operation above will add *any* two fractions, not necessarily rational numbers. The definition of the `addgroupZero` is not cyclic, but recursive along the chain of types.

3.1 A Non-standard Example: the Peano-Church Arithmetic

Construction of "concrete", known, composite numbers, such as the rational fractions or complex numbers is interesting and useful, but does not teach anything new in the field of sophisticated functional programming. We have constructed and played with the ring of univariate polynomials, and the field of modular integers. We have then constructed the Galois field as the class of polynomials on the modular integers and the quotient field of the rational functions. Of course each such construction ended with a comprehensive set of examples. These packages which are quite short, are available from the author.

Paradoxically, a much more primitive model gives a more fruitful insight into the structure of functional computations. We have played with the Peano-Church numerals, a minimalistic construction of the integer arithmetic. The model is frequently used to present some applications of the abstract lambda calculus, but then one usually neglects the problems of polymorphic typing.

The model is based on the following premises. There are two abstract objects, a "dummy" constant *zero* which can be really anything, and an abstract endomorphism, the *successor*, which can be applied to objects of the same type as *zero*. The number N is represented by the *Church numeral* $\mathcal{N}$. This is a function which applies N times the *successor* to *zero*. We declare thus the types of the *successor* and of the Church numerals as:

```
type Succ a = a -> a
type Chnum a = Succ a -> a -> a
```

We will name the first Church numerals `ch0, ch1` etc. The object `ch0` is a function which does nothing to its argument *zero* and returns it: `ch0 s z = z`. The Church `ch1` applies the *successor* once: `ch1 s z = s z`. After having introduced some combinatoric shortcuts, and exploiting the standard combinator `(f . g) x = f (g x)` it is easy to prove the validity of the following instance definitions:

```
instance AddGroup (Chnum a) where
  addgroupZero = flip const
  (n1 + n2) s = n1 s . n2 s
instance Monoid (Chnum a) where
  groupOne = id
  (*) = (.)
```

The arithmetic operations are based on the following observation: if one applies $n1$ times the *successor* to *zero*, and then more $n2$ times to the previous result, one gets the above definition for `(n1 + n2)`. The multiplication is even simpler, since the partial application `n s` is a function which applied n times the *successor* `s` to something, so `(n1 * n2) s z = n1 (n2 s) z`.

Some ambitious students ask the obvious question about the subtraction, and usually they cannot find themselves the solution. The problem is that even if the successor operation in the domain of Church numerals (which should not be confound with the abstract successor which is the *argument* of the Church numeral; the present author used intensely the coloured chalk, but it was not always fully appreciated. . .) is straightforward: `succ n s = n s . s`, the predecessor is not so easy to derive, and Church himself had some doubts.

We define a special successor which acts on pairs of objects according to the rule `sp s (x,any) = (s x,x)`. This successor applied N times to $(zero, zero)$ gives obviously something like $(N, N-1)$ and we may recover the predecessor, from which we construct the subtraction. Of course, the complexity of subtraction is simply horrible, but the evaluation of this complexity is an interesting didactic exercise:

```
pred n s z = pr where
  (_,pr) = n sp (z,z)
  sp (x,_) = (s x,x)
```

All this can be done without algebraic type classes, with specific operations names such as `add` or `mult` substituted for `(+)` and `(*)`. It is possible to define the exponentiation, whose simplicity is shocking:

```
n1^n2 = n2 n1
```

The explanation how it works takes some time, but the students have a splendid occasion to realise that the Set theory expression B^A for the set of all applications from A to B is *not* just a symbolic notation! Moreover, after three years of studies they have seen plenty of recursion examples, from factorial to the Ackerman function, but this was the first time they found the recursion hidden in a functional combinator without the classical recursive structure, terminal clause, etc., and yet conceptually rather simple.

As our primary concern here was to show the application of the functional combinators, and not the structuring of the polymorphism, this example was elaborated in CAML, not in Gofer. With the system classes there are some interesting problems. One *cannot* define abstract, polymorphic numerals represented by concrete objects, for example:

```
ch0 = addgroupZero :: Chnum a
```

because all top level definitions must be resolved, and the system complains that `addgroupZero` is still ambiguous. Of course we can restrict our domain to, say `Chnum String` where the *zero* is the empty string, and the *successor* simply concatenates `"*"` with its argument. But then we had other problems, which are addressed in the Conclusions.

4 Infinite Data Types

4.1 Lazy Manipulation of Power Series

All textbooks on lazy programming, and the packages distributed with Haskell present many lazy streams such as the list of all positive integers, the Fibonacci sequence, or all the primes constructed with the aid of the Eratosthenes sieve. The construction of cyclic data structures has been also discussed in the literature ([5]). We found thus a useful and a little less worked domain of lazy infinite power series.

An *effective and simple* coding of an algorithm dealing with such series is not entirely trivial. The algorithms are usually dominated by the administration of the truncation trivia. In fact, if one implements the algorithms discussed in [4], one sees mainly summing loops and the evaluation of the bounds of these loops, which becomes quite boring. In our approach an univariate power series $u_0 + u_1x + u_2x^2 + u_3x^3 \ldots$ will be represented by the sequence:

```
u0 :> u1 :> u2 :> u3    ...
```

where `:>` is an infix, right associative constructor:

```
data Series a = a :> (Series a)
-- No termination clause!!
```

We could use normal lists, but we have introduced a specific datatype for the following reasons:

– The above definition precludes all attempts to construct explicit *finite* objects. This is a useful debugging aid and a challenge for those students whose first reaction is: "this perversion will never work!".
– We didn't want to overload normal lists with too specific algebraic structure. As we wanted to use the classical comprehension notation, we declared the constructor `:>` as an instance of the `Functor` class, so that we could use such functionals as `map`. We have also overloaded such functionals as `fold` and `zip`.

Our main idea was to show that infinite lazy lists treated as "normal" data, as "first class citizens" simplify enormously the algorithms. The class system served here uniquely for bookkeeping and syntax simplification.

Addition (or subtraction) of series is defined within the `AddGroup` class as `u + v = zipWith (+) u v` where in our case the series fusion might be defined as:

```
zipWith op (u0:>uq) (v0:>vq) = (u0 `op` v0) :> zipWith op uq vq
```

The multiplication is defined recursively. If the series $U = U_0 + x \cdot \overline{U}, \quad V = V_0 + x \cdot \overline{V}$, then we find that $U \cdot V = U_0 \cdot V_0 + x \cdot (U \cdot \overline{V} + V_0 \cdot \overline{U})$. The Gofer translation is trivial. The head of the solution is given immediately, and in order to get the tail we need only the head of the recursive application. We see immediately that the series form an algebra: the definition above is placed in the class `Monoid`, as all internal multiplications, but its context demands that a series be also an instance of the `Module`. So we have a nice cross-referencing structure:

```
instance Monoid a => Module a (Series a) where
   x #* s = map (x *) s

instance (Monoid a, AddGroup (Series a),
          Module a (Series a)) =>
  Monoid (Series a) where
   u@(u0:>uq)*(v0:>vq) = u0*v0 :> (v0 #* uq) + vq*u
```

The division uses the same principle, if $W = U/V$, then $U = V \cdot W$, or $U_0 + x \cdot \overline{U} = V_0 \cdot W_0 + x \cdot (W_0 \cdot \overline{V} + V \cdot \overline{W}$, which after rearranging gives us:

```
instance (Group a) => Group (Series a) where
   (u0:>uq)/v@(v0:>vq) = w0 :> (uq - w0 #* vq)/v
                      where w0 = u0/v0
```

where we see that we don't even need a procedure to divide a series by a coefficient. This example shows once more some mild limitations of the system used. If the scalars of a given Module belong to a Group, and thus to a Field, the Module is a Linear Space. We can declare it explicitly together with the default division as:

```
class (Group a, Module a b) => LinSpace a b where
  (/#) :: b -> a -> b
  x /# y = recip y #* x
```

but nobody will deduce for us that a declared series belongs to a Linear Space, provided that the coefficients admit the division. We must do it by hand, although the instance declaration is empty – we use only the inferred defaults.

Other operations on power series are equally easy to code (compare with [4]). If $W = U^\alpha$, then after the differentiation of both sides we get $W' = \alpha \cdot U^{\alpha-1}U'$, or $W = \alpha \int W \cdot U'/U$. It is the lazy integration which gives sense to this propagating recursion:

```
integ :: (Group a, Ring a) => a -> (Series a) -> (Series a)
integ c0 u = c0 :> zipWith comp2 u intS
  where   comp2 x y = x / (fromInteger y)
          intS = intSeq 1 where intSeq n = n :> intSeq (n+1)
```

It takes some time to master this technique and to appreciate the fact that the definition: $W = \text{Const} + \int f(W)$ is not just a specification, or an equation, but an *algorithm*. It suffices to know the Const to be able to generate the next term and the whole series. The definition above is equivalent to the obvious identity for any series f: $f_n = \frac{f'_{n-1}}{n}$.

Other elementary functions are coded in the same way, for example $W = \exp(U) = \int W \cdot U'$, etc.

If the series fulfills a more complicated, non-linear equation, the lazy approach influences also the construction of the Newton algorithm. Again, instead of coding a loop broken by some convergence criteria, we construct shamelessly an infinite list of infinite iterants. For example, if $W = \sqrt{U}$, then we get $[W^{(0)}, W^{(1)}, \ldots, W^{(n)} \ldots]$, where $W^{(n+1)} = \frac{1}{2}\left(W^{(n)} + U/W^{(n)}\right)$. The construction of this stream is quite simple, the standard prelude function:

```
iterate f x = x : iterate f (f x)
```

does the job, for example to get a square root of y in the domain of rational series, we define:

```
sqRS y = iterate (\x->(1%2)#*(x + y/x))
                 (fromInteger 1) :: [RatSeries]
```

The convergence in this case means obviously the increasing number of correct terms. How to present the final answer to the "end user" of this algorithm? At this moment most students fall into their bad habits, and claim that we must give explicitly the number of terms wanted. So, we restate our religious credo: no, you should generate *one* stream, where the number of correct terms is *exactly* equal to the number of terms looked at. The final solution is based on the observation that the number of correct terms doubles with every iteration $W^{(n)}$. So, we neglect the zeroth (initial) iterant, extract one term from the first series, two terms from the second (after having skipped the first), next four terms from the third after having skipped three, then eight, etc. This exercise is a little impure, as it requires finite lists, their concatenation and reconversion into series, but it is still quite elegant and coded in two lines. The infinite list which convolutes all the stream of series `strS` is given by:

```
convit 1 (tail strS) where
  convit n (x:q) = take n (drop (n-1) x ++ convit (2*n) q
```

The composition and reversal of series is usually considered to be a serious programming challenge. But laziness is a virtue here, and the final codes are again three-liners. Let $U(x) = U_0 + U_1x + U_2x^2 + \ldots$, and $V(x) = V_1x + V_2x^2 + \ldots$, as usual. The free term must be absent from V. We want to find $W = U(V)$. The solution is nothing more than the ordinary, but infinite Horner scheme:

$$U(V) = U_0 + x(V_1 + V_2x + \ldots) \times (U_1 + x(V_1 + V_2x + \ldots) \times (U2 + x(\ldots))) \quad (1)$$

or, horribly enough

```
sercomp u (must_be_zero:>vq) = cmv u where
        cmv (u0:>uq) = u0:>(vq * cmv uq)
```

The reverse of a given series is the solution of the following problem. Given

$$z = t + V_2t^2 + V_3t^3 + \ldots, \qquad \text{find} \quad t = z + W_2z^2 + W_3z^3 + \ldots \tag{2}$$

The suggestion that might be offered to students is to reduce this problem to a composition of series. This is readily done if we note that an auxiliary series p defined by $t = z(1 - zp)$ fulfills the identity:

$$p = (1 - zp)^2 \left(V_2 + V_3z(1 - zp) + V_4z^2(1 - zp)^2 + \ldots\right) \tag{3}$$

```
serinverse (_zero :> _one :> vt) = t  where
   t = fromInteger 0 :> m
   m = fromInteger 1 :> negate (m*m) * sercomp vt t
```

The approach presented above is easily generalized; one might try to find the reciprocal, or solve polynomial equation in the series domain using the approach of Kung and Traub[6]. It is extremely easy to construct formal Padé approximants applying the continuous fraction expansion to the field of series, and reconstructing the rational form. It is possible also to apply lazy streams to the generation of graphs from their partition function[7], but this is a topic too distant from the aim of this conference.

4.2 Partition Generating Function

Another interesting example of the extrapolating recursion is the generation of the *number of partitions of a given integer* N the number of inequivalent non-negative integer solutions of the equation $\sum_{k=1}^{N} x_k = N$ (known also as the number of Ferrer graphs, or Young diagrams for a given N). For example:

$$5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1 \tag{4}$$

i.e. 7 solutions. The generating function for these numbers is well known, but not very easy to handle:

$$Z(x) = \prod_{n=1}^{\infty} \frac{1}{1 - x^n}. \tag{5}$$

Computing a finite approximation to it by standard iterative methods and getting the list
$[1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56, 77, \ldots]$ is simply unwieldy. But we can rewrite this as an open recurrence:

$$Z(x) = Z_1(x), \qquad \text{where} \quad Z_m(x) = \frac{1}{1 - x^m} Z_{m+1}(x). \tag{6}$$

After rewriting Z_m as $Z_m(x) = Z_{m+1}(x) + x^n Z_m(x)$ and after introducing $B_m(x)$ such that $Z_m(x) = 1 + x^m B_m(x)$, we have the final receipt:

$$B_m(x) = 1 + x\left(B_{m+1} + x^{m-1}B_m(x)\right), \tag{7}$$

which gives us the following efficient program:

```
partgen :: IntSeries
partgen = one :> w one where
  one = ringOne
  w n = one :> w (n+1) + byxn ringZero (n-1) (w n)
```

where `byxn` is a function which multiplies a series by x^n (adds n zeros at the beginning).

5 Some Geometry

5.1 Concrete and Abstract Vectors

Defining vectors as triplets (x, y, z), or some tagged data structures is not very interesting, even if with our type class system it is nice to have a syntactically simple way of adding vectors or multiplying them by scalars. We have discussed already some issues related to the construction of the multi-parametric class `LinSpace a b`, where `a` is the type of scalars, and `b` – vectors.

We may play some time with vectors, define scalar `(*.)` and vector `(/\)` products, and even introduce some simplistic tensors. But our main purpose was to show how to use the functional formalism to generate some graphic object, and more specifically – some three-dimensional surfaces created by generalized sweeps or extrusions. We began by restricting our vectors to the type `Vec` which was just the type of floating triplets, and then we have shown how to extend the standard operations to functional objects which described parametrized displacements of vectors:

```
type Path = Float -> Vec   -- The Float means "time"
instance AddGroup Path where
  (f + g) s = f s + g s
  (f - g) s = f s - g s
instance (Module Float) Path where
  (a #* g) s = a #* g s
```

etc. A generalized sweep is an operation which consists in taking a curve, thus a `Path` object, and to transform it by a parametrized transformation, such as rotations or translations, or combinations thereof. It was thus natural to define:

```
type Vtransf = Vec -> Vec
instance Monoid Vtransf where
   groupOne = id
   f * g = f . g
```

Given a transformation `f` of a vector, it is extended to a path by a simple convolution: `trf f g s = f (g s)`, or `trf = (.)`. If this transformation is additionally parametrized, if the function acting upon a vector has a form `f t`, then the induced transformation of paths has a form:

```
transf f t g s = f t (g s)
```

which can be reduced to a combinator `trfpath f = (.) . f`, a form truly detested by most students. So, they are demanded to construct some simple paths, such as a horizontal unit circle, or any parametrized straight line $\mathbf{x} = \mathbf{x}_0 + \mathbf{u}t$:

```
circ ang = (cos ang, sin ang, 0.0)
strline x0 u t = x0 + t#*u
```

We pass then to transformations, for example a parametrized (by a scalar) translation along a given vector `u`:

```
transl u = (+) . (#* u)
```

or, if you prefer, `transl u t x = t#*u + x`. Note that we are doing everything not to use any concrete coordinate representation, when it is possible. So when we define a rotation, we do not care about its general matrix form, but try to reason relative to objects defining our "scene". This is coherent with our main didactic philosophy, that the functional abstractions should be based always on concretes. We define thus a rotation around a normalized axis `n` by an angle ϕ by splitting a vector into its parallel and perpendicular components, and performing the 2-dimensional rotation:

```
rotv n phi x = let parl = (n *. x) #* n
                   perp = x-parl;  tr3 = x/\n
               in parl + cos phi #* perp
                       - sin phi #* tr3
```

5.2 Construction of Parametric Surfaces

We are now ready for a more complex manipulation of vector objects, for example, if we wish to apply two parametrized transformations `f1` and then `f2` at the same time, such as a translation combined with a rotation, we combine it:

```
ovrlap f2 f1 t = f2 t . f1 t
```

Finally we construct our sweeping surface:

```
sweepsurf trf g s t = trf t g s
```

whose combinatoric form `sweepsurf = (flip .) . flip` should be reserved to amateurs.

The remaining task is to take some curve, to design our favourite transformation, and to produce the surface. But the standard versions of Gofer have no graphical output, so we seize the opportunity to convey to our students sometimes forgotten information that modern, multitasking, windowed environments encourage the cooperation between heterogeneous applications. We choose two lists – intervals of the parameters s and t: `intervT` and `intervS`, and we make a grid:

```
grid intervS intervT surf =
  [[surf s t | s <- intervS] | t <- intervT]
```

This grid is lazy, so it may be quite large. It is transformed then into a many-line output string, and piped into Gnuplot or any other drawing program, or saved in a file and processed off-line. For example if we rotate an oblique straight line around the z axis, and if we add to this rotation an oscillating variation of scale, we have to program the following:

```
scale phi v = (1.0 + 0.3*sin(8.0*phi))#*v
mytransf = trfpath (ovrlap scale (rotv (0.0,0.0,1.0)))
line t = strline (1.0,1.0,0.0) normalize (1.0,0.0,1.0) t
mysurf s = sweepsurf mytransf line s
```

and the result is presented on Fig. 1.

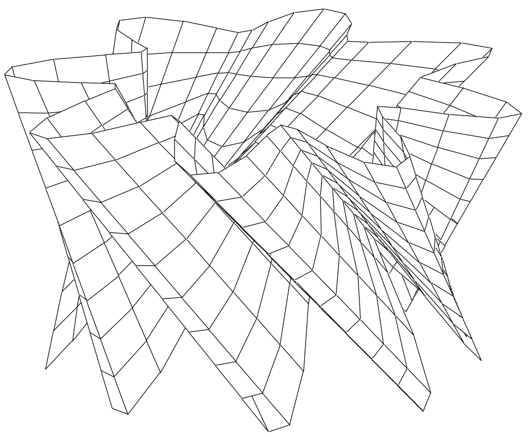

Fig. 1. An example of a parametric surface

Another type of sweep surfaces are *tubes*, constructed by a translation of the generating curve along a trajectory together with a rotation which keeps the original alignment of the generator with respect to the tangent vector of the trajectory. Of course we can compose this translation and rotation with some other transformations. The size of this paper makes it impossible to include more beautiful and very complicated 3D drawings produced by very short programs.

This exercise touches the problem of computing the tangent vector to a path, and permits to develop another branch of our functional mathematics – the construction of differential structures, which cannot be discussed here.

6 Conclusions

We tried to demonstrate that modern polymorphic functional languages may be sensibly used to model and implement quite abstract mathematical structures in a way which is transparent enough to be accepted by mathematically oriented undergraduate students.

Programs in functional languages such as Haskell or Miranda, thanks to their lack of syntactic overhead are coded fast. Higher order functions and lazy evaluation are capable to produce little programming miracles. The inheritance and genericity offered by Haskell seem to be more adequate for that kind of mathematical programming than the approaches based on object-oriented languages, although we would like very much to compare them with, say, the C++ or CLOS protagonists.

We found that the type classes system may be used to define statically the hierarchy of mathematical domains, but that there are some flaws therein. We missed some meta-class system which would permit, for example, the simultaneous construction of several Galois fields parametrized by different characteristics and order; we had problems with the Church numerals: if their polymorphic type is restricted, one cannot attribute any sensible type to the exponentiation operator, as the objects are self-applicable. We had problems with ambiguous types while trying to define metric (or normed) vector spaces: the type of a perfectly reasonable function which normalized a vector dividing it by its norm, did not fit into the class system.

Such observations are addressed rather to readers interested in the programming tool building, and not only in the pedagogical process, but they seem important. The typed, lazy programming languages are excellent tools to teach the constructive approach to mathematics. But the type classes are *not* mathematical domains. Our plans for the future include the construction of a different class system, perhaps better adapted to the construction of abstract mathematics.

References

1. D. Guntz, M. Monagan, *Introduction to Gauss*, Sigsam Bulletin **28**, no. 2, (1994), pp 3 – 19.
2. P. Hudak, S. Peyton Jones, P. Wadler et al., *Report on the programming language Haskell, (Version 1.3)*, Technical report Yale University/Glasgow University, (1995).
3. Mark P. Jones, *Gofer, Functional Programming Environment*, (1991).
4. Donald E. Knuth, *The Art of Computer Programming, Vol 2 / Seminumerical Algorithms*, Addison-Wesley, Reading, (1981).
5. Lloyd Allison, *Circular Programs and Self-referential Structures*, Software — Practice and Experience, Vol. **19**(2), (1989), pp. 99 – 109.

6. H. T. Kung, J. F. Traub, JACM **25** (1978), pp. 245–260.
7. Jerzy Karczmarczuk, *Lazy Functional Programming and Manipulation of Perturbation Series*, Proc. III International Workshop on Software Engineering for High Energy Physics, (1993), pp. 571–581.