Numbers in Prelude for Haskell’

Andrew U. Frank
Dept. of Geoinformation
Technical University Vienna
frank@geoinfo.tuwien.ac.at

Abstract

A more differentiated structure for the classes Num and related in the Haskell-98 prelude is proposed. It structures operations along the lines of algebraic structures but remains close to and compatible with the current prelude. It allows overloading of the regular arithmetic operations for instances where the corresponding axioms are valid.

1 Introduction

The current structure of the number classes Num, Real, Integral, Fractional, and Floating is very close to traditional programming languages and has served the Haskell community well. New applications – we use Haskell for example as a design tool and specification language for geographic information systems – are hindered by the current coarse structure and require a finer subdivision of operations in classes along the lines of their algebraic structure. It is desirable, to separate the additive, commutative group with (+) from the ring which includes also (*), so for example vectors can be made instances for an additive group and not automatically get a (*) as multiplication. We design and specify of parts of geographic information systems and use algebraic methods and Haskell. We push for reuse of well-known structures (like monoid, group, and lattice); the current prelude makes this impossible and we have replaced it with our homebrew version. Similar requests were voiced by more mathematically oriented groups, for example Mechveliani [Haskell and computer algebra].

The proposal here is conservative and very close to the current prelude; it is intended such that current code is not requiring any change. It separates the class number classes in Haskell (report 6.4) in xx classes:, namely Campgroup, Combing, Euclidean Ring, Field, and Floating. It further follows the rule that all operations are included in a class to achieve a uniform method of overloading.

2 Proposed structure in classes

Code to show the classes and default implementation for some operations, covering what is currently in the prelude:

```
import Prelude hiding ((++), Num (..), Integral (..), Fractional
(..),
                    abs, signum, gcd, lcm)

class Zeros z where
    zero :: z

class Ones o where
    one :: o
```

To be able to define some operations in classes, the constants for the units of the algebras must be available in instantiable classes, therefore the two classes *Zeros* and *Ones*.

```
class  Monoid s where
    infixr 5  ++
    (++) :: s -> s -> s
```

The class monoid (without a context of a unit) fixes one operation. For the definition of the commutative group, a unit (zero) would be necessary, but is not included to reduce change in existing code.

```
class  CommGroup a where
    infixl 6  +, -
    (+) :: a -> a -> a
    negate  :: a -> a
    --
    (-) :: a -> a -> a
    a - b = a + (negate b)
    subtract :: a -> a -> a
    subtract =  flip (-)
```

The class for groups with orders require no separate detailed instances, both operations can be derived and defined by default:

```
class (Zeros a, Ord a, CommGroup a) => OrdGroup a where
                -- context required to allow definitions
    abs::  a -> a
    abs a = if a > zero then a else negate a
    difference :: a -> a -> a
    difference a b = abs (a - b)
```

The class for rings does again not include the zero to avoid context.

```
class  CommRing a where
    infixl 7  *
    (*) :: a -> a -> a
    sqr :: a -> a
    sqr a = a * a
```

The operations *gcd* and *lcm*, which are currently in no class, move to EuclideanRing, where also *quot*, *rem*, *div* and *mod* are found. A separation in EuclideanRing as a superclass of GCDRing is possible, but I do not see much justification.

```
class (OrdGroup a, CommRing a) => EuclideanRing a where
    quot, rem, div, mod :: a -> a -> a
    gcd, lcm :: a -> a -> a
    ---
    gcd x y = if x == zero &&  y == zero
                        then error
                          "Prelude.gcd: gcd 0 0 is undefined"
              else gcd' (abs x) (abs y)
                         where  gcd' x y = if y == zero then x
                                           else gcd' y (x `rem`
y)
    lcm x y = if x == zero then zero else
                if y == zero then zero
                   else abs ((x `quot` (gcd x y)) * y)

    quot a b = i    where (i,f) = quotRem a b
    rem a b = f     where (i,f) = quotRem a b
    div a b = i     where (i,f) = divMod a b
    mod a b = f     where (i,f) = divMod a b

    divMod, quotRem :: a -> a -> (a,a)
```

Operations for increment and decrement (possibly renamed to *succ* and *pred*) and the determination of the sign of a number is in the class IntegralDomain:

```
class (Zeros a, Ones a,  Ord a, CommGroup a) => IntegralDomain a
where
    inc, dec :: a -> a
    inc a = one + a
    dec a = a - one
    signum ::  a -> a      -- result is   (-1, 0, 1)
    signum a = if a > zero then one else
                if a == zero then zero else(negate one)
```

The class field replaces Fraction:

```
class (CommRing a)  => Field a where
    infixl 7  /

    recip :: a -> a         -- reciprocal

    (/) :: a -> a -> a
    a / b = a * (recip b)
```

The current class Floating is not changed.

3 Tough Questions

3.1 How to determine units for a type

Mechveliani's proposal suggests additional example parameters for functions. E.g. an operation to find the zero for a data type with signature *zero :: a -> a*. I think that *zero :: a* achieves the same effect, possibly in connection with `*asTypeOf*`. (zero `asTypeOf` x)

3.2 Enforce algebraic structure

Mathematical definition of these algebraic structures demand additional rules. Should this be enforced? What are the rules that go with a class and the operation signs defined in it? In particular, is + and * always commutative or is commutativity added in a separate class (Mechveliani does this). The same question for the inclusion of units -

I go for commutativity of + and * and document this in the suggested names for the classes. If a program needs non-commutative operations, the programmer is free to add a new class and select other symbols for the operation. My goal is to structure the current prelude such that such additions become feasible, but are not forced upon those, who do not need them.

3.3 Including of context

The question of including context in the classes is similar. I have opted not to include more context than is strictly necessary for the proposed classes and not document the usually required context of classes like Eq,, Ord, etc. The compiler is checking with instances the presence of the required classes and instances later. Context at the class level seems mostly a documentation issue and precludes later use of a class when you cannot provide the regular Eq, Ord etc. (I want to avoid forcing instance of Eq which then say "_ == _ = *error "not implementable for type xxx"*"). Example: an implementation may allow the multiplication of functions, but defining Eq, Ord, or Show for functions may be impossible or unnecessary.

3.4 Seldom used operations

The 'mixed type' operations ^ and ^^ which compute special case of the power function are left out and could be carried forward from the current prelude; I would prefer to drop these operations to free some namespace for operations and use special instantiations of the general operation ** (which would then require an additional, separate class).

3.5 FromInteger, FromRational

These two operations are not related to algebraic operations but to the understanding of constants values in the program text or input – they are related to conversions, rounding and input/output and could be grouped with other conversions (to string, from string etc.). This requires a separate proposal.

4 Additional burden compared to current prelude

Most programmers will not see a difference. The currently exported operations for the base types remain unchanged. Only those programs, which define a new number-like class, have to instantiate more classes but not more operations than before. The splitting in several classes will often be beneficial – we have encountered numerous situations where addition is defined reasonably, but not a multiplication or an absolute value.

5 Additional library

The proposal is to change the current class structure but not to extend the prelude. An additional library could include additional classes. For example, algebraic modules and vector spaces with scalar multiplication ($*:: s -> a -> a), noncom mutative etc. I assume also that

Mechveliani's implementation could be built as additional classes and data types on top of this proposal.

6 Conclusion

It is possible to change the class structure for the numeric classes in the prelude to bring them in line with a more algebraic structure and allow more flexibility in the application code. The effect of this change on existing code is minimal and only programs where instances for these classes where defined, are affected.