Rational Types Module in Haskell

Dalton Lundy

June 22, 2018

```
module Fractions where
```

This Module is an indepth implimentation of rational number types written in literate haskell. The .lhs file for this is both compilable by latex and GHC.

In a strongly and statically typed language such as Haskell, the programmar can be extremely specific about the types of data that are being used by any given system. This gives one curious thoughts about advantages of certain types over others. Rationals types may provide better performance in certain systems than other traditional number types. For example, 1/3 is usually evaluated to something like this '0.33333333'; but with rational types, it is just (1,3). In this case in particular, rational types holds cleaner and smaller memory storage and also avoids rounding errors that may appear with furthur calcualtions.

Using record syntax will make things a bit easier for this type:

```
data Frac = Frac {

    numerator :: Int
    ,
    dominator :: Int

    }
```

Let's not use the contructor show method and create a prettier and more intuitive one.

```
instance Show Frac where
    show (Frac a b) = (show a) ++ "/" ++ (show b)
```

Now perhaps, the most important function in this module will be a function to simplify fractions:

```
fracSimplify (Frac _ 0) = Frac 0 0
fracSimplify (Frac 0 _) = Frac 0 0
fracSimplify (Frac a b) = if a < 0 && b < 0 then (fracSimplify (Frac (-a) (-b))) else
                           Frac (quot a gcd) (quot b gcd)
                               where gcd = euclid a b
                                     euclid x y
                                         | x < 0     =    euclid (-x) y
                                         | y < 0     =    euclid x (-y)
                                         | x == y    =    x
                                         | x < y     =    euclid x (y - x)
                                         | otherwise =    euclid (x - y) x
```

Next, since computers are dumb, we must prove that fractions are numbers too:

```
instance Eq Frac where

    (==) f1 f2 =
        case (fracSimplify f1, fracSimplify f2) of
            (Frac a b , Frac c d) -> a == c && b == d

instance Num Frac where


    (+) (Frac a b) (Frac 0 0)
        = Frac a b

    (+) (Frac 0 0) (Frac a b)
        = Frac a b

    (+) (Frac a b) (Frac x y)
        = if b == y then fracSimplify $! Frac (a+x) b else
                         fracSimplify $! Frac ((a*y) + (x * b)) (y*b)

    (*) (Frac a b) (Frac x y)
        = fracSimplify $! Frac (a*x) (b*y)

    (-) (Frac a b) (Frac x y)
        = fracSimplify $! (Frac a b) + (Frac (-1) 1) * (Frac x y)

    abs  (Frac a b)
        = Frac (abs a) (abs b)

    signum (Frac a b)
        =    if a == 0  || b == 0 then Frac 0 0    else
             if a*b < 0 then Frac (-1) 1 else
             Frac 1 1

    fromInteger n
         = Frac (fromIntegral n) 1
```

Here are some additional functions to help working with these types

```
divideFrac (Frac a b) (Frac x y)
    = fracSimplify $! (Frac a b) * (Frac y x)

intToFrac
    = \n -> Frac n 1

fracToFloat (Frac a b) = (fromIntegral  a) / (fromIntegral b)

recipFrac (Frac a b) = Frac b a
```

There is one problem with our rational numbers: since there is a limit to how big our ints can be, our rational numbers cannot be precise at large numbers. This is easy to fix. All we do is add an extra int and define a new proper fraction type; something you probably last heard about in elementary school.

```
data Prop = Prop {

    whole :: Int
    ,
    remainder :: Frac

    } deriving (Eq)


instance Show Prop where
    show (Prop n f) = (show n) ++ " " ++ (show f)


prop_simplify (Prop n (Frac 0 0)) = Prop n (Frac 0 0)

prop_simplify (Prop n (Frac a b)) = Prop (n + n') (fracSimplify $!  Frac (a - (n' * b)) b)
        where n' = quot a b



instance Num Prop where

    (+) (Prop n (Frac a b)) (Prop n' (Frac x y) )
         = prop_simplify $! Prop (n + n') ((Frac a b) + (Frac x y))

    (*) (Prop n (Frac a b)) (Prop n' (Frac x y) )
         = prop_simplify $! Prop (n * n') ((Frac a b) * (Frac x y))

    (-) (Prop n (Frac a b)) (Prop n' (Frac x y) )
         = prop_simplify $! Prop (n - n') ((Frac a b) - (Frac x y))

    abs (Prop n (Frac a b))
         = Prop (abs n) (abs (Frac a b))

    signum (Prop n (Frac a b))
         = Prop (signum n) (Frac 0 0)

    fromInteger n
        = Prop (fromIntegral n) (Frac 0 0)



divideProp p1 p2
    = fractToProp $! divideFrac (propToFrac p1) (propToFrac p2)

intToProp
    = \n -> Prop n  (Frac 0 0)

fractToProp
    = \f -> prop_simplify $! Prop 0 (fracSimplify f)

propToFrac
    = \(Prop n (Frac a b)) ->fracSimplify $!  Frac ((b*n)+a) b
```

However, since the ancient Greeks, we have known that irrational numbers easily ruin rational calculations. If it helps, here are some irrational approximations with the new Rational type.

```
piFrac = Frac 355 113

eFrac  = Frac 87 32
```