Part II

The Haskell 98 Libraries

https://doi.org/10.1017/S0956796803001412 Published online by Cambridge University Press

JFP **13** (1): 149–152, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803001412 Printed in the United Kingdom

Chapter 12

Rational Numbers

```
module  Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %
data  (Integral a)      => Ratio a = ...
type  Rational          =  Ratio Integer
(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approxRational          :: (RealFrac a) => a -> a -> Rational
instance  (Integral a)  => Eq         (Ratio a)  where ...
instance  (Integral a)  => Ord        (Ratio a)  where ...
instance  (Integral a)  => Num        (Ratio a)  where ...
instance  (Integral a)  => Real       (Ratio a)  where ...
instance  (Integral a)  => Fractional (Ratio a)  where ...
instance  (Integral a)  => RealFrac   (Ratio a)  where ...
instance  (Integral a)  => Enum       (Ratio a)  where ...
instance  (Read a,Integral a) => Read (Ratio a)  where ...
instance  (Integral a)  => Show       (Ratio a)  where ...
```

For each `Integral` type t, there is a type `Ratio` t of rational pairs with components of type t. The type name `Rational` is a synonym for `Ratio Integer`.

`Ratio` is an instance of classes `Eq`, `Ord`, `Num`, `Real`, `Fractional`, `RealFrac`, `Enum`, `Read`, and `Show`. In each case, the instance for `Ratio` t simply "lifts" the corresponding operations over t. If t is a bounded type, the results may be unpredictable; for example `Ratio Int` may give rise to integer overflow even for rational numbers of small absolute size.

The operator `(%)` forms the ratio of two integral numbers, reducing the fraction to terms with no common factor and such that the denominator is positive. The functions `numerator` and `denominator` extract the components of a ratio; these are in reduced form with a positive denominator. `Ratio` is an abstract type. For example, `12 % 8` is reduced to 3/2 and `12 % (-8)` is reduced to (-3)/2.

The `approxRational` function, applied to two real fractional numbers `x` and `epsilon`, returns the simplest rational number within the open interval $(\mathtt{x} - \mathtt{epsilon}, \mathtt{x} + \mathtt{epsilon})$. A rational number n/d in reduced form is said to be simpler than another n'/d' if $|n| \leq |n'|$ and $d \leq d'$. Note that it can be proved that any real interval contains a unique simplest rational.

12.1 Library `Ratio`

```
-- Standard functions on rational numbers
module  Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %

ratPrec = 7 :: Int

data  (Integral a)      => Ratio a = !a :% !a  deriving (Eq)
type  Rational          =  Ratio Integer

(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approxRational          :: (RealFrac a) => a -> a -> Rational

-- "reduce" is a subsidiary function used only in this module.
-- It normalises a ratio by dividing both numerator
-- and denominator by their greatest common divisor.
--
-- E.g., 12 `reduce` 8    ==  3 :%   2
--       12 `reduce` (-8) ==  3 :% (-2)

reduce _ 0              =  error "Ratio.% : zero denominator"
reduce x y              =  (x `quot` d) :% (y `quot` d)
                           where d = gcd x y

x % y                   =  reduce (x * signum y) (abs y)

numerator (x :% _)      =  x

denominator (_ :% y)    =  y

instance  (Integral a)  => Ord (Ratio a)  where
    (x:%y) <= (x':%y')  =  x * y' <= x' * y
    (x:%y) <  (x':%y')  =  x * y' <  x' * y
```

https://doi.org/10.1017/S0956796803001412 Published online by Cambridge University Press

```
instance  (Integral a)  => Num (Ratio a)  where
    (x:%y) + (x':%y')   =  reduce (x*y' + x'*y) (y*y')
    (x:%y) * (x':%y')   =  reduce (x * x') (y * y')
    negate (x:%y)       =  (-x) :% y
    abs (x:%y)          =  abs x :% y
    signum (x:%y)       =  signum x :% 1
    fromInteger x       =  fromInteger x :% 1

instance  (Integral a)  => Real (Ratio a)  where
    toRational (x:%y)   =  toInteger x :% toInteger y

instance  (Integral a)  => Fractional (Ratio a)  where
    (x:%y) / (x':%y')   =  (x*y') % (y*x')
    recip (x:%y)        =  y % x
    fromRational (x:%y) =  fromInteger x :% fromInteger y

instance  (Integral a)  => RealFrac (Ratio a)  where
    properFraction (x:%y) = (fromIntegral q, r:%y)
                            where (q,r) = quotRem x y

instance  (Integral a)  => Enum (Ratio a)  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate -- May overflow
    enumFrom         =  numericEnumFrom        -- These numericEnumXXX functions
    enumFromThen     =  numericEnumFromThen    -- are as defined in Prelude.hs
    enumFromTo       =  numericEnumFromTo      -- but not exported from it!
    enumFromThenTo   =  numericEnumFromThenTo

instance  (Read a, Integral a)  => Read (Ratio a)  where
    readsPrec p  =  readParen (p > ratPrec)
                              (\r -> [(x%y,u) | (x,s)   <- readsPrec
                                                           (ratPrec+1) r,
                                                ("%",t) <- lex s,
                                                (y,u)   <- readsPrec
                                                           (ratPrec+1) t ]

instance  (Integral a)  => Show (Ratio a)  where
    showsPrec p (x:%y)  =  showParen (p > ratPrec)
                               (showsPrec (ratPrec+1) x .
                                showString " % " .
                                showsPrec (ratPrec+1) y)
```

https://doi.org/10.1017/S0956796803001412 Published online by Cambridge University Press

```
approxRational x eps    =  simplest (x-eps) (x+eps)
        where simplest x y | y < x      =  simplest y x
                           | x == y     =  xr
                           | x > 0      =  simplest' n d n' d'
                           | y < 0      =  - simplest' (-n') d' (-n) d
                           | otherwise  =  0 :% 1
                                        where xr@(n:%d) = toRational x
                                              (n':%d')  = toRational y

              simplest' n d n' d'       -- assumes 0 < n%d < n'%d'
                        | r == 0     =  q :% 1
                        | q /= q'    =  (q+1) :% 1
                        | otherwise  =  (q*n''+d'') :% n''
                                     where (q,r)      =  quotRem n d
                                           (q',r')    =  quotRem n' d'
                                           (n'':%d'') =  simplest' d' r' d r
```

https://doi.org/10.1017/S0956796803001412 Published online by Cambridge University Press