variable-precision

Claude Heiland-Allen

2012–2018

Contents

1 CHANGES

```
v0.4     better sin, cos, tan
v0.3.1   use complex-generic
v0.2.1   fixed point
v0.2     generic IEEE-ish
v0.1.1   fixed for ghc-7.0.4
v0.1     initial release
```

2 fast/Numeric/VariablePrecision/Integer/Logarithm.hs

```
{-# LANGUAGE MagicHash #-}
module Numeric.VariablePrecision.Integer.Logarithm where

import GHC.Exts (Int(I#))
import GHC.Integer.Logarithms (integerLog2#)

integerLog2 :: Integer -> Int
integerLog2 n = I# (integerLog2# n)
```

3 .gitignore

```
dist
-
```

4 LICENSE

```
Copyright (c) 2012, Claude Heiland-Allen

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

    * Neither the name of  nor the names of other
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

5 Numeric/VariablePrecision/Algorithms.hs

```
{-# LANGUAGE BangPatterns #-}
{- |
Module      :  Numeric.VariablePrecision.Algorithms
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  BangPatterns

Implementations of various floating point algorithms.  Accuracy has not
been extensively verified, and termination has not been proven.

Everything assumes that 'floatRadix' is 2.  This is *not* checked.

Functions taking an @accuracy@ parameter may fail to terminate if
@accuracy@ is too small.  Accuracy is measured in least significant
bits, similarly to '(=~=)'.
```

```
20  In this documentation, /basic functionality/ denotes that methods used
    are from classes:

      * 'Num', 'Eq', 'Ord'.

25  Further, /basic RealFloat functionality/ denotes /basic functionality/
    with the addition of:

      * Anything in 'RealFloat' except for 'atan2'.

30  The intention behind the used functionality documentation is to help
    users decide when it is appropriate to use these generic implementations
    to implement instances.

    -}
35  module Numeric.VariablePrecision.Algorithms
      ( recodeFloat
      , viaDouble
      , (=~=)
      , genericRecip
40    , genericSqrt
      , genericExp
      , genericLog
      , genericLog'
      , genericLog2
45    , genericLog''
      , genericPi
      , genericSin
      , genericPositiveZero
      , genericNegativeZero
50    , genericPositiveInfinity
      , genericNegativeInfinity
      , genericNotANumber
      , sameSign
      ) where
55
    import Data.Bits (bit, shiftR)
    import Data.List (foldl')

    -- | Special values implemented using basic RealFloat functionality.
60  genericPositiveZero, genericNegativeZero, genericPositiveInfinity, ↙
      ↳ genericNegativeInfinity, genericNotANumber :: RealFloat a => a

    genericPositiveZero =  0

    genericNegativeZero = -0
65
    genericPositiveInfinity = result
      where
        result = encodeFloat m e
       m = bit (floatDigits (undefined `asTypeOf` result))
70      e = snd (floatRange  (undefined `asTypeOf` result))

    genericNegativeInfinity = result
      where
        result = encodeFloat (negate m) e
```

```
    m = bit (floatDigits (undefined `asTypeOf` result))
    e = snd (floatRange  (undefined `asTypeOf` result))

genericNotANumber = genericPositiveInfinity + genericNegativeInfinity


-- | Convert between generic 'RealFloat' types more efficiently than
--   'realToFrac'.  Tries hard to preserve special values like
--   infinities and negative zero, but any NaN payload is lost.
--
--   Uses only basic RealFloat functionality.
--
recodeFloat :: (RealFloat a, RealFloat b) => a -> b
recodeFloat !x
  | isNaN x               = genericNotANumber
  | isInfinite x && x > 0 = genericPositiveInfinity
  | isInfinite x && x < 0 = genericNegativeInfinity
  | isNegativeZero x      = genericNegativeZero
  | x == 0                = genericPositiveZero
  | otherwise = uncurry encodeFloat (decodeFloat x)


-- | Check if two numbers have the same sign.
--   May give a nonsense result if an argument is NaN.
sameSign :: (Ord a, Num a) => a -> a -> Bool
sameSign a b = compare 0 a == compare 0 b


-- | Approximate equality.
--   @(a =~= b) c@ when adding the difference to the larger in magnitude
--   changes at most @c@ least significant mantissa bits.
--
--   Uses only basic RealFloat functionality.
--
(=~=) :: RealFloat a => a -> a -> Int -> Bool
(=~=) !x !y !s
  | x == y = True
  | isNaN x && isNaN y = True
  | isNaN x || isNaN y = False
  | isInfinite x || isInfinite y = False
  | not (sameSign a b) = False
  | otherwise = abs (e - f) <= s && abs (x - y) <= encodeFloat 1 (s + (e `max` f
    ↳ ))
  where
    (a, e) = decodeFloat x
    (b, f) = decodeFloat y


-- | Compute a reciprocal using the Newton-Raphson division algorithm,
--   as described in
--   <http://en.wikipedia.org/wiki/Division_%28digital%29#Newton.E2.80.93
  ↳ Raphson_division>.
--
--   Uses only basic RealFloat functionality.
--
genericRecip :: RealFloat a => Int {- ^ accuracy -} -> a -> a
genericRecip accuracy y = recip' y
```

```
130     where
          recip' f0
            | isNaN f0 = f0
            | isInfinite f0 && f0 > 0 = genericPositiveZero
            | isInfinite f0 && f0 < 0 = genericNegativeZero
135         | isNegativeZero f0       = genericNegativeInfinity
            | f0 == 0                 = genericPositiveInfinity
            | f0 <  0 = negate . recip' . negate $ f0
            | otherwise = scaleFloat sh (go d s0 x0)
          where
140         x0 = k48 - k32 * d
            d = significand f0  -- in [0.5,1)
            sh = exponent d - exponent f0
          go !d !s !x
            | (x =~= x') accuracy = x'
145         | s == 0 = x'
            | otherwise = go d (s - 1) x'
          where
            x' = scaleFloat 1 x - d * x * x  -- x * (2 - d * x)
          -- an attempt to avoid recomputing per-type constants
150       p = floatDigits (undefined `asTypeOf` y)
          s0 = ceiling (logBase 2 (fromIntegral (p + 1) / logBase 2 17) :: Double) :: ↙
            ↳ Int
          k48 = recodeFloat (48/17 :: Double)
          k32 = recodeFloat (32/17 :: Double)

155
    -- | Compute a square root using Newton's method.
    --
    --   Uses basic RealFloat functionality and '(/)'.
    --
160 genericSqrt :: RealFloat a => Int {- ^ accuracy -} -> a -> a
    genericSqrt accuracy f0
      | f0 < 0 = genericNotANumber
      | f0 == 0 = f0  -- preserves negative zero
      | isNaN f0 = f0
165   | isInfinite f0 = f0
      | otherwise = go (viaDouble sqrt f)
      where
        e = exponent f0
        d = if even e then 2 else 1
170     s = e - d  -- even
        f = scaleFloat (negate s) f0  -- in [1,4)
        go !r =
          let r' = scaleFloat (-1) (r + f / r)
          in  if (r =~= r') accuracy then scaleFloat (s `shiftR` 1) r' else go r'
175

    -- | Compute an exponential using power series.
    --
    --   Uses basic RealFloat functionality, '(/)' and 'recip'.
180 --
    genericExp :: RealFloat a => Int {-^ accuracy -} -> a -> a
    genericExp accuracy x
      | isNaN x = x
      | isInfinite x && x < 0 = 0
185   | isInfinite x = x
```

```
    | x == 0 = 1
    | x <  0 = recip . genericExp accuracy . negate $ x
    | otherwise = go 0 1 1
    where
      go !s !xnnf{- x^n / n! -} !n
        | (s =~= s') accuracy = s'
        | otherwise  = go s' (xnnf * x / fromIntegral n) (n + 1 :: Int)
        where
          s' = s + xnnf


-- | Compute a logarithm.
--
--   See 'genericLog''' for algorithmic references.
--
--   Uses basic RealFloat functionality, 'sqrt' and 'recip'.
--
genericLog :: RealFloat a => Int {- ^ accuracy -} -> a -> a
genericLog accuracy = genericLog' accuracy (genericLog2 accuracy)


-- | Compute log 2.
--
--   See 'genericLog''' for algorithmic references.
--
--   Uses basic RealFloat functionality, 'sqrt' and 'recip'.
--
genericLog2 :: RealFloat a => Int {- ^ accuracy -} -> a
genericLog2 accuracy = negate (genericLog'' accuracy 0.5)


-- | Compute a logarithm using decomposition and a value for @log 2@.
--
--   See 'genericLog''' for algorithmic references.
--
--   Uses basic RealFloat functionality, 'sqrt', and 'recip'.
--
genericLog' :: RealFloat a => Int {- ^ accuracy -} -> a {- ^ log 2 -} -> a -> a
genericLog' accuracy ln2 x
  | isNaN x      = x
  | x == 0       = genericNegativeInfinity
  | x <  0       = genericNotANumber
  | isInfinite x = x
  | otherwise    = mln2 + genericLog'' accuracy s
  where
    m = exponent    x
    s = significand x
    mln2 -- micro-optimisation
      | m == 0 = 0
      | otherwise = fromIntegral m * ln2


-- | Compute a logarithm for a value in [0.5,1) using the AGM method
--   as described in section 7 of
--   /The Logarithmic Constant: log 2/
--   Xavier Gourdon and Pascal Sebah, May 18, 2010,
--   <http://numbers.computation.free.fr/Constants/Log2/log2.ps>.
```

```
--
--   The precondition is not checked.
--
--   Uses basic RealFloat functionality, 'sqrt', and 'recip'.
--
genericLog'' :: RealFloat a => Int {- ^ accuracy -} -> a {- ^ value in [0.5,1) -} -> a
genericLog'' accuracy x = result
  where
    result = go (-1) 1 (encodeFloat 1 m) 0 1 (scaleFloat m x) 0
    m2 = accuracy - floatDigits (undefined `asTypeOf` result)
    m = m2 `shiftR` 1
    small y = y == 0 || exponent y <= m2
    go !n !a !b !s !c !d !t
      | small ds && small dt = recip (1 - s') - recip (1 - t')
      | otherwise = go n' a' b' s' c' d' t'
      where
        a' = scaleFloat (-1) (a + b)
        c' = scaleFloat (-1) (c + d)
        b' = sqrt (a * b)
        d' = sqrt (c * d)
        ds = scaleFloat n (a * a - b * b)
        dt = scaleFloat n (c * c - d * d)
        t' = t + dt
        s' = s + ds
        n' = n + 1


-- | Compute pi using the method described in section 8 of
--   /Multiple-precision zero-finding methods and the complexity of elementary function evaluation/
--   Richard P Brent, 1975 (revised May 30, 2010),
--   <http://arxiv.org/abs/1004.3412>.
--
--   Uses basic RealFloat functionality, '(/)', and 'sqrt'.
--
genericPi :: RealFloat a => Int {- ^ accuracy -} -> a
--   Works ok up to around 600,000 bits (178,000 decimal digits) but after
--   that further increase to mantissa precision leads to problems.
--   Output compared against /Pi/ by Scott Hemphill <http://www.gutenberg.org/ebooks/50>.
genericPi accuracy = result
  where
    sqr x = x * x
    result = go 1 (sqrt 0.5) 0.25 0 1
    go !a !b !t !k !p
      | (p =~= p') accuracy = p'
      | otherwise = go a' b' t' k' p'
      where
        a' = scaleFloat (-1) (a + b)
        b' = sqrt (a * b)
        t' = t - scaleFloat k (sqr (a' - a))
        k' = k + 1
        p' = scaleFloat (-2) (sqr (a + b) / t)


-- | Compute 'sin' using the method described in section 3 of
```

```
    --   /Efficient multiple-precision evaluation of elementary functions/
    --   David M Smith, 1989,
    --   <http://digitalcommons.lmu.edu/math_fac/1/>
300 --
    --   Requires a value for pi.
    --
    --   Uses basic RealFloat functionality, '(/)', and sqrt.
    genericSin :: RealFloat a => Int {-^ accuracy -} -> a {-^ pi -} -> a {-^ x -} ->↙
        ↳ a
305 genericSin accuracy pi x0 = reduced taylor x0
      where
        sqr y = y * y
        t :: Double
        t = fromIntegral (floatDigits x0 - accuracy)
310     k :: Int
        k = round (t / 3)
        three = 3 ^ k
        up 0 !y = y
        up n !y = up (n - 1) (y * (3 - scaleFloat 2 (sqr y)))
315     reduced f y
          | y == 0      = y
          | y <  0      = (negate . reduced f . negate) y
          | y <= pi / 4 = (up k . f . (/ three)) y
          | y <= pi / 2 = (sqrt . (1 -) . sqr . reduced f . (pi / 2 -)) y
320       | y <= pi     = (reduced f . (pi -)) y
          | y <= pi * 2 = (negate . reduced f . (pi * 2 -)) y
          | otherwise = (reduced f . subtract (pi * 2)) y
        taylor y = (sum' . reverse . takeWhile ((> threshold) . abs) . go 1) y
          where
325         threshold = scaleFloat (negate (floatDigits y * 2)) y
            x2 = sqr y
            go !n !xnf = xnf : go n' xnf'
              where
                n' = n + 1
330             xnf' = negate (x2 * xnf / fromInt (2 * n + 1))
        fromInt :: Num a => Int -> a
        fromInt = fromIntegral

    sum' :: Num a => [a] -> a
335 sum' = foldl' (+) 0

    -- | Lift a function from Double to generic 'RealFloat' types.
    viaDouble :: (RealFloat a, RealFloat b) => (Double -> Double) -> a -> b
    viaDouble f = recodeFloat . f . recodeFloat
340

    -- FIXME everything assumes that floatRadix is 2 without checking
```

6 Numeric/VariablePrecision/Aliases.hs

```
    {-|
    Module      :  Numeric.VariablePrecision.Aliases
    Copyright   :  (c) Claude Heiland-Allen 2012
    License     :  BSD3
  5
    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
```

```
    Portability :  portable

    Aliases for 'recodeFloat' and 'recodeComplex' with specialized types.

    Aliases for commonly desired types.

    -}
    module Numeric.VariablePrecision.Aliases
      ( toFloat, fromFloat, toDouble, fromDouble
      , toComplexFloat, fromComplexFloat, toComplexDouble, fromComplexDouble
      , F, X, C, CF, CX
      , F8, F16, F24, F32, F40, F48, F53
      , f8, f16, f24, f32, f40, f48, f53
      , X8, X16, X24, X32, X40, X48, X53
      , x8, x16, x24, x32, x40, x48, x53
      , CF8, CF16, CF24, CF32, CF40, CF48, CF53
      , cf8, cf16, cf24, cf32, cf40, cf48, cf53
      , CX8, CX16, CX24, CX32, CX40, CX48, CX53
      , cx8, cx16, cx24, cx32, cx40, cx48, cx53
      , module TypeLevel.NaturalNumber
      , module TypeLevel.NaturalNumber.ExtraNumbers
      ) where

    import TypeLevel.NaturalNumber (N8, n8)
    import TypeLevel.NaturalNumber.ExtraNumbers
      (N16, n16, N24, n24, N32, n32, N40, n40, N48, n48, N53, n53)

    import Numeric.VariablePrecision.Float (VFloat)
    import Numeric.VariablePrecision.Fixed (VFixed)
    import Numeric.VariablePrecision.Complex (VComplex, recodeComplex, toComplex, ↙
        ↳ fromComplex)
    import Numeric.VariablePrecision.Algorithms (recodeFloat)

    import Data.Complex.Generic (Complex)

    -- | Convert to a Float from the same precision.
    toFloat :: F24 -> Float
    toFloat = recodeFloat

    -- | Convert from a Float to the same precision.
    fromFloat :: Float -> F24
    fromFloat = recodeFloat

    -- | Convert to a Double from the same precision.
    toDouble :: F53 -> Double
    toDouble = recodeFloat

    -- | Convert from a Double to the same precision.
    fromDouble :: Double -> F53
    fromDouble = recodeFloat

    -- | Convert to a Float from the same precision.
    toComplexFloat :: CF24 -> Complex Float
    toComplexFloat = recodeComplex . toComplex

    -- | Convert from a Float to the same precision.
    fromComplexFloat :: Complex Float -> CF24
```

```
fromComplexFloat = fromComplex . recodeComplex

-- | Convert to a Double from the same precision.
toComplexDouble :: CF53 -> Complex Double
toComplexDouble = recodeComplex . toComplex

-- | Convert from a Double to the same precision.
fromComplexDouble :: Complex Double -> CF53
fromComplexDouble = fromComplex . recodeComplex

type F = VFloat
type X = VFixed
type C = VComplex
type CF = C F
type CX = C X

type F8  = F N8  ; f8  :: F8  ; f8  = 0
type F16 = F N16 ; f16 :: F16 ; f16 = 0
type F24 = F N24 ; f24 :: F24 ; f24 = 0
type F32 = F N32 ; f32 :: F32 ; f32 = 0
type F40 = F N40 ; f40 :: F40 ; f40 = 0
type F48 = F N48 ; f48 :: F48 ; f48 = 0
type F53 = F N53 ; f53 :: F53 ; f53 = 0

type X8  = X N8  ; x8  :: X8  ; x8  = 0
type X16 = X N16 ; x16 :: X16 ; x16 = 0
type X24 = X N24 ; x24 :: X24 ; x24 = 0
type X32 = X N32 ; x32 :: X32 ; x32 = 0
type X40 = X N40 ; x40 :: X40 ; x40 = 0
type X48 = X N48 ; x48 :: X48 ; x48 = 0
type X53 = X N53 ; x53 :: X53 ; x53 = 0

type CF8  = CF N8  ; cf8  :: CF8  ; cf8  = 0
type CF16 = CF N16 ; cf16 :: CF16 ; cf16 = 0
type CF24 = CF N24 ; cf24 :: CF24 ; cf24 = 0
type CF32 = CF N32 ; cf32 :: CF32 ; cf32 = 0
type CF40 = CF N40 ; cf40 :: CF40 ; cf40 = 0
type CF48 = CF N48 ; cf48 :: CF48 ; cf48 = 0
type CF53 = CF N53 ; cf53 :: CF53 ; cf53 = 0

type CX8  = CX N8  ; cx8  :: CX8  ; cx8  = 0
type CX16 = CX N16 ; cx16 :: CX16 ; cx16 = 0
type CX24 = CX N24 ; cx24 :: CX24 ; cx24 = 0
type CX32 = CX N32 ; cx32 :: CX32 ; cx32 = 0
type CX40 = CX N40 ; cx40 :: CX40 ; cx40 = 0
type CX48 = CX N48 ; cx48 :: CX48 ; cx48 = 0
type CX53 = CX N53 ; cx53 :: CX53 ; cx53 = 0
```

7 Numeric/VariablePrecision/Complex.hs

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving, StandaloneDeriving,
    FlexibleInstances, FlexibleContexts, MultiParamTypeClasses, Rank2Types,
    UndecidableInstances #-}
{- |
Module      :  Numeric.VariablePrecision.Complex
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3
```

```
    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
    Portability :  DeriveDataTypeable, GeneralizedNewtypeDeriving, ↙
        ↳ StandaloneDeriving, FlexibleInstances, FlexibleContexts, ↙
        ↳ MultiParamTypeClasses, Rank2Types, UndecidableInstances
10
    Complex numbers with variable precision.

    -}
    module Numeric.VariablePrecision.Complex
15    ( module Data.Complex.Generic
      , VComplex()
      , toComplex
      , fromComplex
      , withComplex
20    , recodeComplex
      , scaleComplex
      , scaleVComplex
      , toComplexDFloat
      , toComplexDFixed
25    , fromComplexDFloat
      , fromComplexDFixed
      , withComplexDFloat
      , withComplexDFixed
      ) where
30
    import Data.Complex.Generic
    import Data.Complex.Generic.Default

    import Numeric.VariablePrecision.Fixed
35  import Numeric.VariablePrecision.Float
    import Numeric.VariablePrecision.Precision
    import Numeric.VariablePrecision.Algorithms (recodeFloat)

    -- | Newtype wrapper around 'Complex' so that instances can be written
40  --   for 'HasPrecision' and 'VariablePrecision'.
    newtype VComplex t p = FromComplex
      { -- | Convert 'VComplex' to 'Complex'.
        toComplex :: Complex (t p)
      }
45    deriving (Eq)

    deriving instance Num (Complex (t p)) => Num (VComplex t p)
    deriving instance Fractional (Complex (t p)) => Fractional (VComplex t p)
    deriving instance Floating (Complex (t p)) => Floating (VComplex t p)
50
    -- | Convert 'Complex' to 'VComplex'.
    fromComplex :: Complex (t p) -> VComplex t p
    fromComplex = FromComplex

55  instance NaturalNumber p => Num (Complex (VFloat p)) where
      (+) = addDefault
      (-) = subDefault
      (*) = mulDefault
      negate = negateDefault
60    fromInteger = fromIntegerDefault
```

```
    abs = absDefault
    signum = signumDefault

instance NaturalNumber p => Num (Complex (VFixed p)) where
  (+) = addDefault
  (-) = subDefault
  (*) = mulDefault
  negate = negateDefault
  fromInteger = fromIntegerDefault
  abs = error "Num.abs: unimplemented for Complex VFixed"
  signum = error "Num.signum: unimplemented for Complex VFixed"

instance NaturalNumber p => Fractional (Complex (VFloat p)) where
  (/) = divDefaultRF
  fromRational = fromRationalDefault

instance NaturalNumber p => Fractional (Complex (VFixed p)) where
  (/) = divDefault
  fromRational = fromRationalDefault

instance NaturalNumber p => Floating (Complex (VFloat p)) where
  pi = piDefault
  exp = expDefault
  log = logDefault
  sqrt = sqrtDefault
  sin = sinDefault
  cos = cosDefault
  tan = tanDefault
  sinh = sinhDefault
  cosh = coshDefault
  tanh = tanhDefault
  asin = asinDefault
  acos = acosDefault
  atan = atanDefault
  asinh = asinhDefault
  acosh = acoshDefault
  atanh = atanhDefault

instance NaturalNumber p => ComplexRect (Complex (VFloat p)) (VFloat p) where
  mkRect x y = (x :+ y)
  rect (x :+ y) = (x, y)
  real = realDefault
  imag = imagDefault
  realPart = realPartDefault
  imagPart = imagPartDefault
  conjugate = conjugateDefault
  magnitudeSquared = magnitudeSquaredDefault
  sqr = sqrDefaultRF
  (.*) = rmulDefault
  (*.) = mulrDefault

instance NaturalNumber p => ComplexRect (Complex (VFixed p)) (VFixed p) where
  mkRect x y = (x :+ y)
  rect (x :+ y) = (x, y)
  real = realDefault
  imag = imagDefault
  realPart = realPartDefault
```

```
    imagPart = imagPartDefault
    conjugate = conjugateDefault
    magnitudeSquared = magnitudeSquaredDefault
    sqr = sqrDefault -- FIXME
    (.*) = rmulDefault
    (*.) = mulrDefault

instance NaturalNumber p => ComplexPolar (Complex (VFloat p)) (VFloat p) where
    mkPolar = mkPolarDefault
    cis = cisDefault
    polar = polarDefault
    magnitude = magnitudeDefaultRF
    phase = phaseDefaultRF

instance NaturalNumber p => ComplexRect (VComplex VFloat p) (VFloat p) where
    mkRect x y = FromComplex (x :+ y)
    rect = rect . toComplex
    real = realDefault
    imag = imagDefault
    realPart = realPartDefault
    imagPart = imagPartDefault
    conjugate = conjugateDefault
    magnitudeSquared = magnitudeSquaredDefault
    sqr = sqrDefaultRF
    (.*) = rmulDefault
    (*.) = mulrDefault

instance NaturalNumber p => ComplexRect (VComplex VFixed p) (VFixed p) where
    mkRect x y = FromComplex (x :+ y)
    rect = rect . toComplex
    real = realDefault
    imag = imagDefault
    realPart = realPartDefault
    imagPart = imagPartDefault
    conjugate = conjugateDefault
    magnitudeSquared = magnitudeSquaredDefault
    sqr = sqrDefault -- FIXME
    (.*) = rmulDefault
    (*.) = mulrDefault

instance NaturalNumber p => ComplexPolar (VComplex VFloat p) (VFloat p) where
    mkPolar = mkPolarDefault
    cis = cisDefault
    polar = polarDefault
    magnitude = magnitudeDefaultRF
    phase = phaseDefaultRF

instance HasPrecision (VComplex VFloat)

instance HasPrecision (VComplex VFixed)

instance VariablePrecision (VComplex VFloat) where
    adjustPrecision = withComplex (fmap adjustPrecision)

instance VariablePrecision (VComplex VFixed) where
    adjustPrecision = withComplex (fmap adjustPrecision)
```

```
175 instance Normed (VComplex VFloat) where
      norm1 z = abs (realPart z) + abs (imagPart z)
      norm2 = magnitude
      norm2Squared = magnitudeSquared
      normInfinity z = abs (realPart z) `max` abs (imagPart z)
180
    instance NaturalNumber p => Show (VComplex VFloat p) where
      showsPrec p = showsPrec p . toComplex

    instance NaturalNumber p => Show (VComplex VFixed p) where
185   showsPrec p = showsPrec p . toComplex

    instance NaturalNumber p => Read (VComplex VFloat p) where
      readsPrec p = map (first fromComplex) . readsPrec p
        where first f (a, b) = (f a, b)
190
    instance NaturalNumber p => Read (VComplex VFixed p) where
      readsPrec p = map (first fromComplex) . readsPrec p
        where first f (a, b) = (f a, b)

195 -- | Lift an operation on 'X.Complex' to one on 'VComplex'.
    withComplex :: (Complex (t p) -> Complex (t q)) -> (VComplex t p -> VComplex t q↙
        ↳ )
    withComplex f = fromComplex . f . toComplex

    -- | Much like 'mapComplex' 'recodeFloat'.
200 recodeComplex :: (RealFloat a, RealFloat b) => Complex a -> Complex b
    recodeComplex = fmap recodeFloat

    -- | Much like 'withComplex' 'scaleComplex''.
    scaleVComplex :: NaturalNumber p => Int -> VComplex VFloat p -> VComplex VFloat ↙
        ↳ p
205 scaleVComplex = withComplex . scaleComplex

    -- | Much like 'mapComplex' 'scaleFloat'.
    scaleComplex :: RealFloat r => Int -> Complex r -> Complex r
    scaleComplex = fmap . scaleFloat
210
    -- | Freeze a 'VComplex VFloat'.
    toComplexDFloat :: NaturalNumber p => VComplex VFloat p -> Complex DFloat
    toComplexDFloat = fmap toDFloat . toComplex

215 -- | Freeze a 'VComplex VFixed'.
    toComplexDFixed :: NaturalNumber p => VComplex VFixed p -> Complex DFixed
    toComplexDFixed = fmap toDFixed . toComplex

    -- | Thaw a 'Complex DFloat'.  Results in 'Nothing' on precision mismatch.
220 fromComplexDFloat :: NaturalNumber p => Complex DFloat -> Maybe (VComplex VFloat↙
        ↳  p)
    fromComplexDFloat (x :+ y) = do
      r <- fromDFloat x
      i <- fromDFloat y
      return (r .+ i)
225
    -- | Thaw a 'Complex DFixed'.  Results in 'Nothing' on precision mismatch.
    fromComplexDFixed :: NaturalNumber p => Complex DFixed -> Maybe (VComplex VFixed↙
        ↳  p)
```

```
fromComplexDFixed (x :+ y) = do
  r <- fromDFixed x
  i <- fromDFixed y
  return (r .+ i)

-- | Thaw a 'Complex DFloat' to its natural precision.  'Nothing' is passed on
--   precision mismatch between real and imaginary parts.
withComplexDFloat :: Complex DFloat -> (forall p . NaturalNumber p => Maybe (↙
    ↳ VComplex VFloat p) -> r) -> r
withComplexDFloat (x :+ y) f = withDFloat x $ \r -> f $ do
  i <- fromDFloat y
  return (r .+ i)

-- | Thaw a 'Complex DFixed' to its natural precision.  'Nothing' is passed on
--   precision mismatch between real and imaginary parts.
withComplexDFixed :: Complex DFixed -> (forall p . NaturalNumber p => Maybe (↙
    ↳ VComplex VFixed p) -> r) -> r
withComplexDFixed (x :+ y) f = withDFixed x $ \r -> f $ do
  i <- fromDFixed y
  return (r .+ i)
```

8 Numeric/VariablePrecision/Fixed.hs

```
{-# LANGUAGE BangPatterns, DeriveDataTypeable, Rank2Types, MonoLocalBinds #-}
{- |
Module      :  Numeric.VariablePrecision.Fixed
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  BangPatterns, DeriveDataTypeable, Rank2Types, MonoLocalBinds

Variable precision software fixed point based on @Integer@.

Accuracy has not been extensively verified.

Example:

> reifyPrecision 1000 $ \prec ->
>   show $ auto (355 :: VFixed N15) / 113 `atPrecision` prec

-}
module Numeric.VariablePrecision.Fixed
  ( VFixed()
  , DFixed(..)
  , toDFixed
  , fromDFixed
  , withDFixed
  ) where

import Data.Data (Data())
import Data.Typeable (Typeable())

import Data.Bits (bit, shiftL, shiftR)
import Data.Ratio ((%), numerator, denominator)
```

```
35  import Numeric (readSigned, readFloat)
    import Text.FShow.Raw (BinDecode(..), binDecFormat, FormatStyle(Generic))

    import Numeric.VariablePrecision.Precision
    import Numeric.VariablePrecision.Precision.Reify
40

    -- | A software implementation of fixed point arithmetic, using an
    --   'Integer' adjusted to @p@ bits after the binary point.
    newtype VFixed p = F Integer
45    deriving (Data, Typeable)


    instance HasPrecision VFixed where
      -- default implementation
50

    instance VariablePrecision VFixed where
      adjustPrecision = self
        where
55        self (F x)
            | s > 0 = F (x `shiftL` s)
            | s < 0 = F (x `shiftR` negate s)
            | otherwise = F x
          s = naturalNumberAsInt (undefined `asPrecOut` self) - naturalNumberAsInt (↙
              ↳ undefined `asPrecIn` self)
60        asPrecIn :: p -> (t p -> t q) -> p
          asPrecIn _ _ = undefined
          asPrecOut :: q -> (t p -> t q) -> q
          asPrecOut _ _ = undefined

65
    instance NaturalNumber p => BinDecode (VFixed p) where
      decode l@(F x) = (x, negate . fromIntegral . precision $ l)
      showDigits l = 2 + floor ((1 + fromIntegral (precision l)) * logBase 10 2 :: ↙
          ↳ Double)

70
    instance NaturalNumber p => Show (VFixed p) where

      show = binDecFormat (Generic (Just (-5, 5))) Nothing

75
    instance NaturalNumber p => Read (VFixed p) where

      readsPrec _ = readSigned readFloat -- FIXME ignores precedence?

80
    instance NaturalNumber p => Eq (VFixed p) where F x == F y = x == y


    instance NaturalNumber p => Ord (VFixed p) where F x `compare` F y = x `compare`↙
        ↳ y
85

    instance NaturalNumber p => Num (VFixed p) where
      F x + F y = F $  x + y
```

```
  F x - F y = F $  x - y
  l@(F x) * F y = F $ (x * y) `shiftR` fromIntegral (precision l)
  negate (F x) = F (negate x)
  abs (F x) = F (abs x)
  signum (F x)
    | x > 0 = 1
    | x < 0 = -1
    | otherwise = 0
  fromInteger x = let r = F (x `shiftL` fromIntegral (precision r)) in r


instance NaturalNumber p => Fractional (VFixed p) where
  l@(F x) / F y = F ((x `shiftL` fromIntegral (precision l)) `quot` y)
  fromRational x =
    let r = F ((numerator x `shiftL` fromIntegral (precision r)) `quot` ↙
        ↳ denominator x) in r


instance NaturalNumber p => Real (VFixed p) where
  toRational l@(F x) = x % (bit . fromIntegral . precision) l


instance NaturalNumber p => RealFrac (VFixed p) where
  properFraction f@(F x) = w (fromIntegral n, g)
    where
      w (a, b)
        | a < 0 && b /= 0 = (a + 1, b - 1)
        | otherwise = (a, b)
      n = x `shiftR` p
      p = fromIntegral (precision f)
      g = F (x - (n `shiftL` p))


-- | A concrete format suitable for storage or wire transmission.
data DFixed = DFixed{ dxPrecision :: !Word, dxMantissa :: !Integer }
  deriving (Eq, Ord, Read, Show, Data, Typeable)


-- | Freeze a 'VFixed'.
toDFixed :: NaturalNumber p => VFixed p -> DFixed
toDFixed f@(F m) = DFixed (precision f) m


-- | Thaw a 'DFixed'.  Results in 'Nothing' on precision mismatch.
fromDFixed :: NaturalNumber p => DFixed -> Maybe (VFixed p)
fromDFixed d@(DFixed _ m)
  | dxPrecision d == precision result = Just result
  | otherwise = Nothing
  where
    result = F m  --  -XMonoLocalBinds


-- | Thaw a 'DFixed' to its natural precision.
withDFixed :: DFixed -> (forall p . NaturalNumber p => VFixed p -> r) -> r
withDFixed (DFixed p m) f = reifyPrecision p $ \prec -> f (F m `atPrecision` ↙
    ↳ prec)
```

9 Numeric/VariablePrecision/Float.hs

```
{-# LANGUAGE BangPatterns, DeriveDataTypeable, Rank2Types #-}
{- |
Module      :  Numeric.VariablePrecision.Float
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  BangPatterns, DeriveDataTypeable, Rank2Types

Variable precision software floating point based on @(Integer, Int)@ as
used by 'decodeFloat'.  Supports infinities and NaN, but not negative
zero or denormalization.

Accuracy has not been extensively verified, and termination of numerical
algorithms has not been proven.

-}
module Numeric.VariablePrecision.Float
  ( VFloat()
  , Normed(norm1, norm2, norm2Squared, normInfinity)
  , effectivePrecisionWith
  , effectivePrecision
  , (-@?)
  , DFloat(..)
  , toDFloat
  , fromDFloat
  , withDFloat
  ) where

import Data.Data (Data())
import Data.Typeable (Typeable())

import Data.Bits (bit, shiftL, shiftR)
import Data.Ratio ((%), numerator, denominator)

import GHC.Float (showSignedFloat)
import Numeric (readSigned, readFloat)
import Text.FShow.RealFloat (DispFloat(), FShow(fshowsPrec), fshowFloat)

import Numeric.VariablePrecision.Algorithms
import Numeric.VariablePrecision.Precision
import Numeric.VariablePrecision.Precision.Reify
import Numeric.VariablePrecision.Integer.Logarithm


-- | A software implementation of floating point arithmetic, using a strict
--   pair of 'Integer' and 'Int', scaled similarly to 'decodeFloat', along
--   with additional values representing:
--
--     * positive infinity (@1/0@),
--
--     * negative infinity (@-1/0@),
--
--     * not a number (@0/0@).
```

```
--
--   The 'Floating' instance so far only implements algorithms for:
--
--     * 'pi',
--
--     * 'sqrt',
--
--     * 'exp',
--
--     * 'log',
--
--     * 'sin', 'cos', 'tan'.
--
--   These 'Floating' methods transit via 'Double' and so have limited
--   precision:
--
--     * 'asin', 'acos', 'atan',
--
--     * 'sinh', 'cosh', 'tanh',
--
--     * 'asinh', 'acosh', 'atanh'.
--
--   'floatRange' is arbitrarily limited to mitigate the problems that
--   occur when enormous integers might be needed during some number
--   type conversions (worst case consequence: program abort in gmp).
--
data VFloat p
  = F !Integer !Int
    -- invariant: matches decodeFloat spec
    -- if unsure, use encodeVFloat which maintains the invariant
    -- if sure, use checkVFloat which checks the invariant
    -- only construct with bare F when absolutely sure
  | FZero   -- FIXME add negative zero
  | FPosInf
  | FNegInf
  | FNaN    -- FIXME add payload
  deriving (Data, Typeable)

encodeVFloat :: NaturalNumber p => VFloat p -> Integer -> Int -> VFloat p
encodeVFloat witness = self
    where
      b = fromIntegral $ precision (undefined `asTypeOf` witness)
      b' = b - 1
      self 0 !_ = FZero
      self m  e = checkVFloat "encodeFloat" $ encodeFloat'' (m > 0) m' (e - sh)↙
          ↳ l
        where
          absm = abs m
          m' = absm `shift` sh
          e2 = integerLog2 absm
          sh = b - e2
          l = integerLog2 m'
      encodeFloat'' !s' !m' !e' !l
        | m' <= 0 = failed -- FIXME
        | b' == l = F (if s' then m' else negate m') e'
        | b' <  l = {-# SCC "encodeFloat''.shiftR" #-} encodeFloat'' s' (m' `↙
            ↳ shiftR` 1) (e' + 1) (l - 1)
```

```
          | b' > l = {-# SCC "encodeFloat''.shiftL" #-} encodeFloat'' s' (m' `↙
              ↳ shiftL` 1) (e' - 1) (l + 1)
          | otherwise = failed -- FIXME
          where
            failed = error $ "Numeric.VariablePrecision.Float.encodeVFloat: ↙
                ↳ internal error (please report this bug): "
                          ++ show (b, b', l, s', m', e')


instance NaturalNumber p => DispFloat (VFloat p) where


instance NaturalNumber p => FShow (VFloat p) where

  fshowsPrec p = showSignedFloat fshowFloat p


instance NaturalNumber p => Show (VFloat p) where

  showsPrec = fshowsPrec


instance NaturalNumber p => Read (VFloat p) where

  readsPrec _ = readSigned readFloat -- FIXME ignores precedence, NaN/Inf fail?


instance HasPrecision VFloat


minimumExponent, maximumExponent :: Int
minimumExponent = negate (bit 20)
maximumExponent =          bit 20

asTypeIn :: (a -> b) -> a
asTypeIn _ = undefined

asTypeOut :: (a -> b) -> b
asTypeOut _ = undefined

asTypeOut2 :: (a -> b -> c) -> c
asTypeOut2 _ = undefined


instance VariablePrecision VFloat where

  adjustPrecision = self
    where
      p = asTypeIn  self
      q = asTypeOut self
      np = floatDigits p
      nq = floatDigits q
      n = nq - np
      self FZero    = FZero
      self FPosInf  = FPosInf
      self FNegInf  = FNegInf
      self FNaN     = FNaN
```

```
        self (F m e)
          | n >  0 = encodeVFloat q (m `shiftL` n) (e - n)
          | n == 0 = encodeVFloat q m e
          | n <  0 = encodeVFloat q (m `shiftR` negate n) (e + negate n)
          | otherwise = unreachable

instance Eq (VFloat p) where

  FZero   == FZero   = True
  F a b   == F x y   = a == x && b == y
  FPosInf == FPosInf = True
  FNegInf == FNegInf = True
  -- everything else including NaN
  _       == _       = False

  a       /= x       = not (a == x)


instance Ord (VFloat p) where

  FZero   <  FZero   = False
  FZero   <  F x _   = 0 < x
  F a _   <  FZero   = a < 0
  F a b   <  F x y
    | a > 0 && x > 0 && b <  y = True
    | a > 0 && x > 0 && b == y = a < x
    | a > 0 && x > 0 && b >  y = False
    | a > 0 && x < 0           = False
    | a < 0 && x > 0           = True
    | a < 0 && x < 0 && b <  y = False
    | a < 0 && x < 0 && b == y = a < x
    | a < 0 && x < 0 && b >  y = True
    | otherwise = unreachable
  FNaN    <  _       = False
  _       <  FNaN    = False
  FPosInf <  _       = False
  _       <  FPosInf = True
  _       <  FNegInf = False
  FNegInf <  _       = True

  a       >  x       = x < a

  a       <= x       = a < x || a == x

  a       >= x       = a > x || a == x

  min a@FNaN !_ = a
  min !_ x@FNaN = x
  min a x
    | a <= x    = a
    | otherwise = x

  max a@FNaN !_ = a
  max !_ x@FNaN = x
  max a x
    | a >= x    = a
```

```
      | otherwise = x

225   -- 'compare' uses default implementation in Ord


instance NaturalNumber p => Num (VFloat p) where

230   f@(F a b) + F x y
        | b >  y = encodeVFloat f (a + (x `shiftR` (b - y))) b
        | b == y = encodeVFloat f (a + x) b
        | b <  y = encodeVFloat f ((a `shiftR` (y - b)) + x) y
        | otherwise = unreachable
235   a@FNaN  + _       = a
      _       + x@FNaN  = x
      FZero   + x       = x
      a       + FZero   = a
      FPosInf + FNegInf = FNaN
240   FNegInf + FPosInf = FNaN
      FPosInf + _       = FPosInf
      _       + FPosInf = FPosInf
      FNegInf + _       = FNegInf
      _       + FNegInf = FNegInf
245
      f@(F a b) - F x y
        | b >  y = encodeVFloat f (a - (x `shiftR` (b - y))) b
        | b == y = encodeVFloat f (a - x) b
        | b <  y = encodeVFloat f ((a `shiftR` (y - b)) - x) y
250     | otherwise = unreachable
      a@FNaN  - _       = a
      _       - x@FNaN  = x
      FZero   - x       = negate x
      a       - FZero   = a
255   FPosInf - FPosInf = FNaN
      FNegInf - FNegInf = FNaN
      FPosInf - _       = FPosInf
      _       - FPosInf = FNegInf
      FNegInf - _       = FNegInf
260   _       - FNegInf = FPosInf

      negate (F a b) = checkVFloat "negate" $ F (negate a) b
      negate FZero   = FZero
      negate FPosInf = FNegInf
265   negate FNegInf = FPosInf
      negate a@FNaN  = a

      abs !a
        | a < 0     = negate a
270     | otherwise = a

      signum !a
        | a < 0     = -1
        | a > 0     =  1
275     | otherwise =  a

      f@(F a b) * F x y   = encodeVFloat f ((a * x) `shiftR` (k - 1)) (b + y + k - ↙
          ↳ 1) where k = fromIntegral $ precision f
      a@FNaN    * _       = a
```

```
    _          * x@FNaN  = x
    FZero      * FPosInf = FNaN
    FZero      * FNegInf = FNaN
    FZero      * _       = FZero
    FPosInf    * FZero   = FNaN
    FNegInf    * FZero   = FNaN
    _          * FZero   = FZero
    a          * x
      | sameSign a x     = FPosInf
      | otherwise        = FNegInf

    fromInteger !i = encodeFloat i 0


instance NaturalNumber p => Real (VFloat p) where

    toRational FZero = 0
    toRational (F m e)
      | e >  0 = fromInteger (m `shiftL` e)
      | e == 0 = fromInteger m
      | e <  0 = m % bit (negate e)
      | otherwise = unreachable
    toRational FPosInf =  1 % 0
    toRational FNegInf = (-1) % 0
    toRational FNaN    =  0 % 0


instance NaturalNumber p => Fractional (VFloat p) where

    f@(F _ _) / g@(F _ _) = f * recip g
    a@FNaN    / _         = a
    _         / x@FNaN    = x
    FPosInf   / FPosInf = FNaN
    FPosInf   / FNegInf = FNaN
    FNegInf   / FPosInf = FNaN
    FNegInf   / FNegInf = FNaN
    _         / FPosInf = FZero
    _         / FNegInf = FZero
    a         / FZero
      | a > 0           = FPosInf
      | a < 0           = FNegInf
      | otherwise       = FNaN
    FZero     / _       = FZero
    a         / x
      | a `sameSign` x  = FPosInf
      | otherwise       = FNegInf

    recip a@FNaN = a
    recip FZero    = FPosInf
    recip FPosInf = FZero
    recip FNegInf = FZero
    recip f@(F m e) = encodeVFloat f (bit k `quot` m) (negate (k + e)) where k = 2↙
        ↳  * fromIntegral (precision f)

    fromRational r = fromInteger (numerator r) / fromInteger (denominator r) -- ↙
        ↳ FIXME accuracy
```

```
335 instance NaturalNumber p => RealFrac (VFloat p) where

      properFraction = self
        where
          p = fromIntegral $ precision (asTypeIn self)
340       self FZero = (0, FZero)
          self me@(F m e)
            | e >= 0 = (fromInteger m, FZero)
            | e < negate p = (0, me)
            | otherwise = (fromInteger n', f')
345         where
              n = m `shiftR` (negate e)
              d = encodeVFloat (asTypeIn self) (n `shiftL` (negate e)) e
              f = me - d
              (n', f')
350             | (m >= 0) == (f >= 0) = (n, f)
                | otherwise = (n + 1, f - 1)
          self f = (error $ "Numeric.VariablePrecision.Float.properFraction: not ↙
            ↳ finite: " ++ show f, f)

      -- 'truncate' uses default implementation in RealFrac
355
      -- 'floor' uses default implementation in RealFrac

      -- 'ceiling' uses default implementation in RealFrac

360   -- 'round' uses default implementation in RealFrac


    instance NaturalNumber p => RealFloat (VFloat p) where

365   floatRadix _ = 2

      floatDigits = self
        where
          prec = fromIntegral $ precision (asTypeIn self)
370       self = const prec

      floatRange = const (minimumExponent, maximumExponent) -- FIXME arbitrary
      -- this floatRange is somewhat arbitrary, but toInteger gives integers
      -- with up to around (precision + maxExponent) bits, the value here
375   -- gives rise to potentially more than 300k decimal digits...

      isNaN FNaN = True
      isNaN _    = False

380   isInfinite FPosInf = True
      isInfinite FNegInf = True
      isInfinite _       = False

      isDenormalized _ = False
385
      isNegativeZero _ = False

      isIEEE _ = False -- FIXME what does this mean?
```

```
390     decodeFloat FZero    = (0, 0)
        decodeFloat (F m e) = (m, e)
        decodeFloat f = error $ "Numeric.VariablePrecision.Float.decodeFloat: not ↙
            ↳ finite: " ++ show f

        encodeFloat = self
395       where
            self = encodeVFloat (undefined `asTypeOf` asTypeOut2 self)

        exponent = self
          where
400         prec = fromIntegral $ precision (asTypeIn self)
            self FZero    = 0
            self (F _ e) = e + prec
            self f = error $ "Numeric.VariablePrecision.Float.exponent: not finite: " ↙
                ↳ ++ show f

405     significand = self
          where
            prec = fromIntegral $ precision (asTypeIn self)
            e = negate prec
            self (F m _) = checkVFloat "significand" $ F m e
410         self f = f

        scaleFloat n (F m e) = checkVFloat "scaleFloat" $ F m (e + n)
        scaleFloat _ f = f

415     -- 'atan2' uses default implementation in RealFloat


    shift :: Integer -> Int -> Integer
    shift !n !k
420   | k >  0 = n `shiftL` k
      | k == 0 = n
      | k <  0 = n `shiftR` (negate k)
      | otherwise = unreachable

425
    instance NaturalNumber p => Floating (VFloat p) where -- FIXME

        pi    = genericPi 2

430     sqrt = genericSqrt 2

        exp   = genericExp 2

        log   = self
435       where
            log2 = genericLog2 2
            self = genericLog' 2 log2

        -- '(**)' uses default implementation in Floating
440
        -- 'logBase' uses default implementation in Floating

        sin = genericSin 2 pi
```

```
445     cos x = sin (x + pi/2)

        tan x = sin x / cos x

        sinh = viaDouble sinh -- FIXME
450
        cosh = viaDouble cosh -- FIXME

        tanh = viaDouble tanh -- FIXME

455     asin = viaDouble asin -- FIXME

        acos = viaDouble acos -- FIXME

        atan = viaDouble atan -- FIXME
460
        asinh = viaDouble asinh -- FIXME

        acosh = viaDouble acosh -- FIXME

465     atanh = viaDouble atanh -- FIXME


    -- despite the name, using this is vital for correct behaviour
    -- because it properly handles underflow and overflow as well as
470 -- checking that the invariant for F holds
    checkVFloat :: NaturalNumber p => String -> VFloat p -> VFloat p
    checkVFloat = self
      where
        prec = fromIntegral $ precision (asTypeOut2 self)
475     prec' = prec - 1
        elo = minimumExponent
        ehi = maximumExponent
        self s x@(F m e)
          | not mok   = error $ "Numeric.VariablePrecision.Float.checkVFloat." ++ s ↙
            ↳ ++ ": internal error (please report this bug): " ++ show ((m, am, lm↙
            ↳ , prec, prec', mok), (elo, e, ehi, eok))
480       | eok       = x
          | e < elo   = FZero   -- underflow
          | m > 0     = FPosInf -- overflow
          | m < 0     = FNegInf -- overflow
          | otherwise = unreachable
485       where
            eok = elo <= e  && e  <= ehi
            mok = lm == prec'
            lm = integerLog2 am
            am  = abs m
490     self _ x = x


    -- | A selection of norms.
    class HasPrecision t => Normed t where
495   norm1         :: NaturalNumber p => t p -> VFloat p
      norm2         :: NaturalNumber p => t p -> VFloat p
      norm2Squared :: NaturalNumber p => t p -> VFloat p
      normInfinity :: NaturalNumber p => t p -> VFloat p
```

```
500
instance Normed VFloat where
  norm1 = abs
  norm2 = abs
  norm2Squared x = x * x
505   normInfinity = abs


-- | A measure of meaningful precision in the difference of two
--   finite non-zero values.
510 --
--   Values of very different magnitude have little meaningful
--   difference, because @a + b `approxEq` a@ when @|a| >> |b|@.
--
--   Very close values have little meaningful difference,
515 --   because @a + (a - b) `approxEq` a@ as @|a| >> |a - b|@.
--
--   'effectivePrecisionWith' attempts to quantify this.
--
effectivePrecisionWith :: (Num t, RealFloat r) => (t -> r) {- ^ norm -} -> t -> ↙
    ↳ t -> Int
520 effectivePrecisionWith n i j
  | t a && t b && t c = p - (d `max` (e - d))
  | otherwise = 0
  where
    t k = k > 0 && not (isInfinite k)
525   d = (x `max` y) - z
    e = abs (x - y) `min` p
    p = floatDigits a
    x = exponent a
    y = exponent b
530   z = exponent c
    a = n i
    b = n j
    c = n (i - j)

535
-- | Much like 'effectivePrecisionWith' combined with 'normInfinity'.
effectivePrecision :: (NaturalNumber p, HasPrecision t, Normed t, Num (t p)) => ↙
    ↳ t p -> t p -> Int
effectivePrecision = effectivePrecisionWith normInfinity
infix 6 `effectivePrecision`
540

-- | An alias for 'effectivePrecision'.
(-@?) :: (NaturalNumber p, HasPrecision t, Normed t, Num (t p)) => t p -> t p ->↙
    ↳  Int
(-@?) = effectivePrecision
545 infix 6 -@?


unreachable :: a
unreachable = error "Numeric.VariablePrecision.Float: internal error (please ↙
    ↳ report this bug): unreachable code was reached"
550

-- | A concrete format suitable for storage or wire transmission.
```

```
data DFloat
  = DFloat            { dPrecision :: !Word, dMantissa :: !Integer, dExponent :: !Int }
  | DZero             { dPrecision :: !Word }
  | DPositiveInfinity { dPrecision :: !Word }
  | DNegativeInfinity { dPrecision :: !Word }
  | DNotANumber       { dPrecision :: !Word }
  deriving (Eq, Ord, Read, Show, Data, Typeable)

-- | Freeze a 'VFloat'.
toDFloat :: NaturalNumber p => VFloat p -> DFloat
toDFloat f@(F m e) = DFloat            (precision f) m e
toDFloat f@FZero   = DZero             (precision f)
toDFloat f@FPosInf = DPositiveInfinity (precision f)
toDFloat f@FNegInf = DNegativeInfinity (precision f)
toDFloat f@FNaN    = DNotANumber       (precision f)

-- | Thaw a 'DFloat'.  Results in 'Nothing' on precision mismatch.
fromDFloat :: NaturalNumber p => DFloat -> Maybe (VFloat p)
fromDFloat d
  | dPrecision d == precision result = Just result
  | otherwise = Nothing
  where
    result = case d of
      DFloat _ m e -> encodeVFloat undefined m e
      DZero _ -> FZero
      DPositiveInfinity _ -> FPosInf
      DNegativeInfinity _ -> FNegInf
      DNotANumber _ -> FNaN

-- | Thaw a 'DFloat' to its natural precision.
withDFloat :: DFloat -> (forall p . NaturalNumber p => VFloat p -> r) -> r
withDFloat (DFloat p m e) f = reifyPrecision p $ \prec -> f (encodeVFloat undefined m e `atPrecision` prec)
withDFloat d f = unsafeWithDFloat d f

-- | Thaw a 'DFloat' without guaranteeing a well-formed 'VFloat' value.
--   Possibly slightly faster.
unsafeWithDFloat :: DFloat -> (forall p . NaturalNumber p => VFloat p -> r) -> r
unsafeWithDFloat (DFloat        p m e) f = reifyPrecision p $ \prec -> f (F m e `atPrecision` prec)
unsafeWithDFloat (DZero             p) f = reifyPrecision p $ \prec -> f (FZero `atPrecision` prec)
unsafeWithDFloat (DPositiveInfinity p) f = reifyPrecision p $ \prec -> f (FPosInf `atPrecision` prec)
unsafeWithDFloat (DNegativeInfinity p) f = reifyPrecision p $ \prec -> f (FNegInf `atPrecision` prec)
unsafeWithDFloat (DNotANumber       p) f = reifyPrecision p $ \prec -> f (FNaN `atPrecision` prec)
```

10 Numeric/VariablePrecision.hs

```
{- |
Module      :  Numeric.VariablePrecision
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3
```

```
    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
    Portability :  portable

    Convenience module.

    -}
    module Numeric.VariablePrecision
      ( module Numeric.VariablePrecision.Float
      , module Numeric.VariablePrecision.Fixed
      , module Numeric.VariablePrecision.Complex
      , module Numeric.VariablePrecision.Precision
      , module Numeric.VariablePrecision.Precision.Reify
      , module Numeric.VariablePrecision.Aliases
      , module Numeric.VariablePrecision.Algorithms
      ) where

    import Numeric.VariablePrecision.Float
    import Numeric.VariablePrecision.Fixed
    import Numeric.VariablePrecision.Complex
    import Numeric.VariablePrecision.Precision
    import Numeric.VariablePrecision.Precision.Reify
    import Numeric.VariablePrecision.Aliases
    import Numeric.VariablePrecision.Algorithms
```

11 Numeric/VariablePrecision/Precision.hs

```
    {- |
    Module      :  Numeric.VariablePrecision.Precision
    Copyright   :  (c) Claude Heiland-Allen 2012
    License     :  BSD3

    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
    Portability :  portable

    Classes for types with precision represented by a type-level natural
    number, and variable precision types.

    Note that performance may be (even) slow(er) with some versions of the
    type-level-natural-number package.

    -}
    module Numeric.VariablePrecision.Precision
      ( HasPrecision(precisionOf)
      , precision
      , atPrecision
      , atPrecisionOf
      , (.@)
      , VariablePrecision(adjustPrecision)
      , auto
      , withPrecision
      , withPrecisionOf
      , (.@~)
      , module TypeLevel.NaturalNumber
      , module Data.Word
      ) where
```

```
import TypeLevel.NaturalNumber
  ( NaturalNumber(..), Zero, SuccessorTo, n0, successorTo )
import Data.Word (Word)

-- | A class for types with precision.
--   The methods must not evaluate their arguments, and their results
--   must not be evaluated.
--   Minimal complete definition: (none).
class HasPrecision t where
  precisionOf :: NaturalNumber p => t p -> p
  precisionOf _ = undefined

-- | Much like 'naturalNumberAsInt' combined with 'precisionOf'.
precision :: (NaturalNumber p, HasPrecision t) => t p -> Word
precision = fromIntegral . naturalNumberAsInt . precisionOf

-- | Much like 'const' with a restricted type.
atPrecision :: (NaturalNumber p, HasPrecision t) => t p -> p -> t p
atPrecision = const

-- | Much like 'const' with a restricted type.
--   Precedence between '<' and '+'.
atPrecisionOf :: (HasPrecision t, HasPrecision s) => t p -> s p -> t p
atPrecisionOf = const
-- where _ = precisionOf t `asTypeOf` precisionOf s
infixl 5 `atPrecisionOf`

-- | An alias for 'atPrecisionOf'.
--   Precedence between '<' and '+'.
(.@) :: (HasPrecision t , HasPrecision s) => t p -> s p -> t p
(.@) = atPrecisionOf
infixl 5 .@

-- | A class for types with adjustable precision.
--   Minimal complete definition: 'adjustPrecision'.
class HasPrecision t => VariablePrecision t where
  -- | Adjust the precision of a value preserving as much accuracy as
  --   possible.
  adjustPrecision :: (NaturalNumber p, NaturalNumber q) => t p -> t q

-- | Synonym for 'adjustPrecision'.
auto :: (VariablePrecision t, NaturalNumber p, NaturalNumber q) => t p -> t q
auto = adjustPrecision

-- | Much like 'adjustPrecision' combined with 'atPrecision'.
withPrecision :: (NaturalNumber p, NaturalNumber q, VariablePrecision t) => t p ↙
    ↳ -> q -> t q
withPrecision s q = adjustPrecision s `atPrecision` q
```

```
-- | Much like 'withPrecision' combined with 'precisionOf'.
--   Precedence between '<' and '+'.
withPrecisionOf :: (NaturalNumber p, NaturalNumber q, VariablePrecision t, ↙
    ↳ HasPrecision s) => t p -> s q -> t q
withPrecisionOf s w = s `withPrecision` precisionOf w
infixl 5 `withPrecisionOf`


-- | An alias for 'withPrecisionOf'.
--   Precedence between '<' and '+'.
(.@~) :: (NaturalNumber p, NaturalNumber q, VariablePrecision t, HasPrecision s)↙
    ↳ => t p -> s q -> t q
(.@~) = withPrecisionOf
infixl 5 .@~
```

12 Numeric/VariablePrecision/Precision/Reify.hs

```
{-# LANGUAGE Rank2Types #-}
{- |
Module      :  Numeric.VariablePrecision.Precision.Reify
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  Rank2Types

Reify from value-level to type-level using Rank2Types.

-}

module Numeric.VariablePrecision.Precision.Reify
  ( reifyPrecision
  , withReifiedPrecision
  , (.@$)
  ) where

import Numeric.VariablePrecision.Precision
  ( VariablePrecision, withPrecision, Word
  , NaturalNumber, n0, successorTo
  )

-- | Reify a precision from value-level to type-level.
reifyPrecision :: Word -> (forall p . NaturalNumber p => p -> a) -> a
-- Implemented as described in an email from Gregory Grosswhite
-- <http://markmail.org/message/55iuty6axeljj2do>
reifyPrecision = go n0
  where
    go :: NaturalNumber q => q -> Word -> (forall p . NaturalNumber p => p -> a)↙
        ↳ -> a
    go n i f
      | i == 0 = f n
      | otherwise = go (successorTo n) (i - 1) f

-- | Much like 'reifyPrecision' combined with 'withPrecision'.
withReifiedPrecision
  :: (VariablePrecision t, NaturalNumber p)
```

```
40    => t p {-^ original value -}
      -> Word {- ^ new precision -}
      -> (forall q. NaturalNumber q => t q -> a) {-^ operation -}
      -> a
    withReifiedPrecision x i f = reifyPrecision i (f . withPrecision x)
45  infixl 1 `withReifiedPrecision`

    -- | An alias for 'withReifiedPrecision'.
    (.@$)
      :: (VariablePrecision t, NaturalNumber p)
50    => t p {-^ original value -}
      -> Word {- ^ new precision -}
      -> (forall q. NaturalNumber q => t q -> a) {-^ operation -}
      -> a
    (.@$) = withReifiedPrecision
55  infixl 1 .@$
```

13 pure/Numeric/VariablePrecision/Integer/Logarithm.hs

```
{-# LANGUAGE BangPatterns #-}
module Numeric.VariablePrecision.Integer.Logarithm where

import Data.Bits (shiftL)
5
integerLog2 :: Integer -> Int
integerLog2 n
  | n > 0 = go (-1) 1
  | otherwise = error $ "integerLog2: non-positive argument: " ++ show n
10  where
    go !l !b
      | n < b = l
      | otherwise = go (l + 1) (b `shiftL` 1)
```

14 README

```
Users of ghc-7.0.4 might require -fcontext-stack=100
This flag can also be :set within ghci.
```

15 Setup.hs

```
import Distribution.Simple
main = defaultMain
```

16 THANKS

```
Robert P Munafo -- testing with ghc-7.0.4 resulted in fixes
  5318e8aac2fb75a9ecf9a3b661e911f610aa1784
    support old base with RealFloat required for Complex
  591fac0e024a41c2c31cde909385d36376e5a715
5     support ghc-7.0.4 by increasing context stack size
  f7c89c0503b3038dc0057e0dc029db0dd98f2b0d
    document ghc-7.0.4 requiring increasing context stack size
```

17 TODO

```
Numeric.VariablePrecision.Precision
        examples of usage in documentation

Numeric.VariablePrecision.Precision.Reify
        examples of usage in documentation

Numeric.VariablePrecision.Float
        readsPrec ignores precedence
        check accuracy of fromRational
        check accuracy of (+), (-), (*)
        check accuracy of (/), recip
        proper (a)(sin|cos|tan)(h) (bad: viaDouble)
        profile and optimise space and time
        consider rounding modes
        consider mixed-precision operations
        consider IEEE -0 semantics

TypeLevel.NaturalNumber.ExtraNumbers
        submit upstream and remove if accepted
```

18 TypeLevel/NaturalNumber/ExtraNumbers.hs

```
{- |
Module      :  TypeLevel.NaturalNumber.ExtraNumbers
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  stable
Portability :  portable

Boilerplate definitions generated by:

> flip mapM_ [16..53] $ \p -> let s = show p in
>   putStrLn $ "type N" ++ s ++ " = SuccessorTo N" ++ show (p - 1) ++
>     " ; n" ++ s ++ " :: N" ++ s ++ " ; n" ++ s ++ " = undefined"

-}
module TypeLevel.NaturalNumber.ExtraNumbers where

import TypeLevel.NaturalNumber (N15, SuccessorTo)

type N16 = SuccessorTo N15 ; n16 :: N16 ; n16 = undefined
type N17 = SuccessorTo N16 ; n17 :: N17 ; n17 = undefined
type N18 = SuccessorTo N17 ; n18 :: N18 ; n18 = undefined
type N19 = SuccessorTo N18 ; n19 :: N19 ; n19 = undefined
type N20 = SuccessorTo N19 ; n20 :: N20 ; n20 = undefined
type N21 = SuccessorTo N20 ; n21 :: N21 ; n21 = undefined
type N22 = SuccessorTo N21 ; n22 :: N22 ; n22 = undefined
type N23 = SuccessorTo N22 ; n23 :: N23 ; n23 = undefined
type N24 = SuccessorTo N23 ; n24 :: N24 ; n24 = undefined
type N25 = SuccessorTo N24 ; n25 :: N25 ; n25 = undefined
type N26 = SuccessorTo N25 ; n26 :: N26 ; n26 = undefined
type N27 = SuccessorTo N26 ; n27 :: N27 ; n27 = undefined
type N28 = SuccessorTo N27 ; n28 :: N28 ; n28 = undefined
type N29 = SuccessorTo N28 ; n29 :: N29 ; n29 = undefined
type N30 = SuccessorTo N29 ; n30 :: N30 ; n30 = undefined
```

```
type N31 = SuccessorTo N30 ; n31 :: N31 ; n31 = undefined
type N32 = SuccessorTo N31 ; n32 :: N32 ; n32 = undefined
type N33 = SuccessorTo N32 ; n33 :: N33 ; n33 = undefined
type N34 = SuccessorTo N33 ; n34 :: N34 ; n34 = undefined
type N35 = SuccessorTo N34 ; n35 :: N35 ; n35 = undefined
type N36 = SuccessorTo N35 ; n36 :: N36 ; n36 = undefined
type N37 = SuccessorTo N36 ; n37 :: N37 ; n37 = undefined
type N38 = SuccessorTo N37 ; n38 :: N38 ; n38 = undefined
type N39 = SuccessorTo N38 ; n39 :: N39 ; n39 = undefined
type N40 = SuccessorTo N39 ; n40 :: N40 ; n40 = undefined
type N41 = SuccessorTo N40 ; n41 :: N41 ; n41 = undefined
type N42 = SuccessorTo N41 ; n42 :: N42 ; n42 = undefined
type N43 = SuccessorTo N42 ; n43 :: N43 ; n43 = undefined
type N44 = SuccessorTo N43 ; n44 :: N44 ; n44 = undefined
type N45 = SuccessorTo N44 ; n45 :: N45 ; n45 = undefined
type N46 = SuccessorTo N45 ; n46 :: N46 ; n46 = undefined
type N47 = SuccessorTo N46 ; n47 :: N47 ; n47 = undefined
type N48 = SuccessorTo N47 ; n48 :: N48 ; n48 = undefined
type N49 = SuccessorTo N48 ; n49 :: N49 ; n49 = undefined
type N50 = SuccessorTo N49 ; n50 :: N50 ; n50 = undefined
type N51 = SuccessorTo N50 ; n51 :: N51 ; n51 = undefined
type N52 = SuccessorTo N51 ; n52 :: N52 ; n52 = undefined
type N53 = SuccessorTo N52 ; n53 :: N53 ; n53 = undefined
```

19 variable-precision.cabal

```
Name:                variable-precision
Version:             0.4
Synopsis:            variable-precision floating point
Description:
  Software floating point with type-tagged variable mantissa precision,
  implemented using a strict pair of 'Integer' and 'Int' scaled alike
  to 'decodeFloat'.  Version 0.4 adds more number-type-agnostic numerical
  algorithms ('sin', 'cos', 'tan').
  .
  Instances of the usual numeric type classes are provided, along with
  additional operators (with carefully chosen fixities) to coerce,
  adjust and reify precisions.
  .
  The intention with this library is to be relatively simple but still
  useful, refer to the documentation for caveats concerning accuracy and
  assorted ill-behaviour.
  .
  Usage with ghc(i)-7.0.4 might require @-fcontext-stack=100@.

Homepage:            https://code.mathr.co.uk/variable-precision
License:             BSD3
License-file:        LICENSE
Author:              Claude Heiland-Allen
Maintainer:          claude@mathr.co.uk
Copyright:           (c) 2012 Claude Heiland-Allen
Category:            Math
Build-type:          Simple

Cabal-version:       >=1.6

Extra-source-files:
```

```
    CHANGES
    README
    THANKS
    TODO
    pure/Numeric/VariablePrecision/Integer/Logarithm.hs
    fast/Numeric/VariablePrecision/Integer/Logarithm.hs

Flag fast
  Description:        Enable optimisations requiring recent integer-gmp
  Default:            True

Library
  Exposed-modules:
    Numeric.VariablePrecision
    Numeric.VariablePrecision.Algorithms
    Numeric.VariablePrecision.Fixed
    Numeric.VariablePrecision.Float
    Numeric.VariablePrecision.Complex
    Numeric.VariablePrecision.Precision
    Numeric.VariablePrecision.Precision.Reify
    Numeric.VariablePrecision.Aliases
    TypeLevel.NaturalNumber.ExtraNumbers
  Other-modules:
    Numeric.VariablePrecision.Integer.Logarithm
  Build-depends:
    base >= 3 && < 6,
    complex-generic >= 0.1.1 && < 0.2,
    floatshow >= 0.2 && < 0.3,
    type-level-natural-number >= 1 && < 2
  if (!flag(fast))
    HS-source-dirs: . pure
  if ( flag(fast))
    HS-source-dirs: . fast
    Build-depends: integer-gmp >= 0.4
  GHC-Options:         -Wall -fcontext-stack=100
  GHC-Prof-Options:    -prof -auto-all -caf-all

source-repository head
  type:     git
  location: https://code.mathr.co.uk/variable-precision.git

source-repository this
  type:     git
  location: https://code.mathr.co.uk/variable-precision.git
  tag:      v0.4
```