exp-extended

Claude Heiland-Allen

2015–2019

Contents

1 exp-extended.cabal

```
name:                exp-extended
version:             0.2
synopsis:            floating point with extended exponent range
homepage:            https://code.mathr.co.uk/exp-extended
license:             BSD3
license-file:        LICENSE
author:              Claude Heiland-Allen
maintainer:          claude@mathr.co.uk
copyright:           2015,2016,2019 Claude Heiland-Allen
category:            Numeric
build-type:          Simple
cabal-version:       >=1.10
description:
    A small library to extend floating point types with a larger
    exponent, so that you can represent really huge or really tiny
    numbers without overflow to infinity or underflow to zero.
    .
    >> unExpExtended . log . exp .          expExtended' $ 1000
    > 1000.0
    >>                 log . exp                         $ 1000
    > Infinity
    >> unExpExtended . log . exp . negate . expExtended' $ 1000
    > -1000.0
    >>                 log . exp . negate                $ 1000
    > -Infinity
    .
    Version 0.2 has lighter dependencies.

library
  hs-source-dirs:      src
  exposed-modules:     Numeric.ExpExtended
                       Numeric.ExpExtended.Internal
  build-depends:       base >= 4.9 && < 4.14
  default-language:    Haskell2010
  other-extensions:    TypeFamilies
```

```
source-repository head
  type:     git
  location: https://code.mathr.co.uk/exp-extended.git

source-repository this
  type:     git
  location: https://code.mathr.co.uk/exp-extended.git
  tag:      v0.2
```

2 .gitignore

```
.cabal-sandbox
cabal.sandbox.config
dist
dist-newstyle
```

3 LICENSE

4 Setup.hs

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

5 src/Numeric/ExpExtended.hs

```
{-# LANGUAGE TypeFamilies #-}
-- | Extend floating point types with a larger exponent range.
module Numeric.ExpExtended
  ( ExpExtendable(..)
  , expExtended'
  , unExpExtendable
  , unExpExtended
  , EDouble
  , EFloat
  ) where

import Data.Ratio (numerator, denominator)

import Text.Show as T
import Text.Read as T

import Numeric

import Numeric.ExpExtended.Internal

getCache :: ExpExtendable a => proxy a -> Cache a
getCache _ = cache

getCache1 :: ExpExtendable a => (b -> proxy a) -> Cache a
getCache1 _ = cache

getCache2 :: ExpExtendable a => (c -> b -> proxy a) -> Cache a
getCache2 _ = cache

getCacheIn1 :: ExpExtendable a => (proxy a -> b) -> Cache a
getCacheIn1 _ = cache

-- | Does the extended value fit in the base type without over/underflow?
unExpExtendable :: ExpExtendable a => ExpExtended a -> Bool
unExpExtendable = self
  where
    c = getCacheIn1 self
    self x = withExpExtended x $ \_ e -> cRangeMin c < e && e < cRangeMax c

-- | Scale to the base type
--   (possibly overflowing to infinity or underflowing to zero).
unExpExtended :: ExpExtendable a => ExpExtended a -> a
unExpExtended x = withExpExtended x $ \m e -> scaleFloat e m

-- | Extend the exponent range while preserving the value.
--
--   > expExtended' x == expExtended x 0
expExtended' :: ExpExtendable a => a -> ExpExtended a
expExtended' x = expExtended x 0

-- | Extend floating point types with a larger exponent range.
--
--   Implementors need only implement:
--
--    * the 'ExpExtended' data type, isomorphic to a strict pair @(a, 'Int')@
```

```
--    * its constructor 'unsafeExpExtended'
--    * its destructor 'withExpExtended'
--
--   Using a data family allows the UNPACK optimisation.
class RealFloat a => ExpExtendable a where
  {-# MINIMAL unsafeExpExtended, withExpExtended #-}

  -- | Associated data.
  --
  --   Instances: 'Compensable', 'Enum', 'Eq', 'Floating', 'Fractional', 'Num', 'Ord', 'Precise', 'Read', 'Real', 'RealFloat', 'RealFrac', 'Show'
  data ExpExtended a

  -- | Deconstruct into basic value and exponent.
  withExpExtended :: ExpExtended a -> (a -> Int -> r) -> r

  -- | Construct from a basic value and an exponent, without checking the
  --   invariant.  Use 'expExtended' instead.
  unsafeExpExtended :: a -> Int -> ExpExtended a

  -- | Cache of magic values.  Stored once per instance to avoid recomputation.
  cache :: Cache a
  cache = cacheDefault

  -- | Construct from a basic value and an exponent, ensuring that the result
  --   establishes the internal invariant:
  --
  --   > m == significand m && ((m == 0 || isInfinite m || isNaN m) ==> e == 0)
  --
  --   Also handles overflow to infinity, and underflow to zero.
  expExtended :: a -> Int -> ExpExtended a
  expExtended = self
    where
      c = getCache2 self
      self m e
        | m == 0 = unsafeExpExtended m 0
        | isNaN m = unsafeExpExtended m 0
        | isInfinite m = unsafeExpExtended m 0
        | e > cSupExponent c || e' > maxExponent =
            unsafeExpExtended (signum m / 0) 0
        | e < cInfExponent c || e' < minExponent =
            unsafeExpExtended (signum m * 0) 0
        | otherwise = unsafeExpExtended m' e'
        where
          m' = significand m
          e'' = exponent m
          e' = e + e''

instance ExpExtendable Float where
  data ExpExtended Float = EF {-# UNPACK #-} !Float {-# UNPACK #-} !Int
  unsafeExpExtended = EF
  withExpExtended (EF m e) f = f m e

type EFloat = ExpExtended Float

instance ExpExtendable Double where
  data ExpExtended Double = ED {-# UNPACK #-} !Double {-# UNPACK #-} !Int
```

```
  unsafeExpExtended = ED
  withExpExtended (ED m e) f = f m e

type EDouble = ExpExtended Double


instance (ExpExtendable a, Show a) => Show (ExpExtended a) where
  showsPrec d m = withExpExtended m $ \a b -> showParen (d > 10)
    $ showString "expExtended "
    . T.showsPrec 11 a
    . showChar ' '
    . T.showsPrec 11 b

instance (ExpExtendable a, Read a) => Read (ExpExtended a) where
  readPrec = parens $ prec 10 $ do
    Ident "expExtended" <- lexP
    a <- step T.readPrec
    b <- step T.readPrec
    return $ expExtended a b

instance ExpExtendable a => Eq (ExpExtended a) where
  a == b = withExpExtended a $ \m1 e1 -> withExpExtended b $ \m2 e2 ->
    e1 == e2 && m1 == m2
  a /= b = withExpExtended a $ \m1 e1 -> withExpExtended b $ \m2 e2 ->
    e1 /= e2 || m1 /= m2

instance ExpExtendable a => Ord (ExpExtended a) where
  compare a b = withExpExtended a $ \m1 e1 -> withExpExtended b $ \m2 e2 ->
    case max e1 e2 of
      e | m1 == 0 -> compare 0 m2
        | m2 == 0 -> compare m1 0
        | otherwise -> scaleFloat (e1 - e) m1 `compare` scaleFloat (e2 - e) m2

instance ExpExtendable a => Num (ExpExtended a) where
  negate a = withExpExtended a $ \m e -> unsafeExpExtended (negate m) e
  a + b = withExpExtended a $ \m1 e1 -> withExpExtended b $ \m2 e2 ->
    case max e1 e2 of
      e | m1 == 0 -> b
        | m2 == 0 -> a
        | otherwise -> expExtended (scaleFloat (e1 - e) m1 + scaleFloat (e2 - e)↙
          ↳ m2) e
  a * b = withExpExtended a $ \m1 e1 -> withExpExtended b $ \m2 e2 ->
    expExtended (m1 * m2) (e1 + e2)
  abs a = withExpExtended a $ \m e -> unsafeExpExtended (abs m) e
  signum a = withExpExtended a $ \m _ -> expExtended (signum m) 0
  fromInteger = self
    where
      c = getCache1 self
      e = cDigits c
      self n = case fromInteger n of
        m | isInfinite m -> scaleFloat e (fromInteger (cDownShift c n e))
          | otherwise -> expExtended m 0

instance ExpExtendable a => Fractional (ExpExtended a) where
  recip a = withExpExtended a $ \m e -> expExtended (recip m) (negate e)
  fromRational q =
    let x = fromInteger (numerator q) / fromInteger (denominator q)
```

```
          p = toRational x
          d = q - p
          y = fromInteger (numerator d) / fromInteger (denominator d)
      in  x + y

instance ExpExtendable a => Real (ExpExtended a) where
  toRational = self
    where
      c = getCacheIn1 self
      self a = withExpExtended a $ \m e ->
        let q = toRational m in case compare e 0 of
          GT -> q * fromInteger (cRadixPower c e)
          EQ -> q
          LT -> q / fromInteger (cRadixPower c (negate e))

instance ExpExtendable a => RealFrac (ExpExtended a) where
  properFraction = self
    where
      c = getCache1 (snd . self)
      self a = withExpExtended a $ \m e -> case () of
        _ | e > cDigits c ->
              case properFraction (scaleFloat (cDigits c + 1) m) of
                (n, _) -> (fromInteger (cUpShift c n (e - cDigits c - 1)), 0)
          | e < 0 -> (0, a)
          | otherwise -> case properFraction (scaleFloat e m) of
              (n, m') -> (n, expExtended m' 0)

reduce :: RealFrac a => a -> a -> a
reduce p x = case properFraction (x / p) of
  (n, y) -> let _n :: Int ; _n = n in y * p

instance ExpExtendable a => Floating (ExpExtended a) where
  pi = expExtended pi 0

  exp = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e -> case () of
        _ | e == 0 -> expExtended (exp m) 0
          | cExpMin c <= e && e <= cExpMax c ->
              expExtended (exp (scaleFloat e m)) 0
          | e >= cExpSup c -> expExtended (if m > 0 then m / 0 else 0) 0
          | e < cExpInf c -> 1
          | otherwise -> expExtended (exp m) 0 ^ (cRadix c ^ e)

  log = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e ->
        expExtended (log m + fromIntegral e * cLogRadix c) 0

  sqrt = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e ->
        expExtended (sqrt (if even e then m else cRadix' c * m)) (e `div` 2)
```

```
225     sin = self
          where
            c = getCache1 self
            self x = let y = reduce (2 * pi) x in withExpExtended y $ \m e ->
              if e < cRangeMin c
230           then y
              else expExtended (sin (scaleFloat e m)) 0

        cos = self
          where
235         c = getCache1 self
            self x = withExpExtended (reduce (2 * pi) x) $ \m e ->
              if e < cRangeMin c
              then 1
              else expExtended (cos (scaleFloat e m)) 0
240
        tan = self
          where
            c = getCache1 self
            self x = let y = reduce pi x in withExpExtended y $ \m e ->
245           if e < cRangeMin c
              then y
              else expExtended (tan (scaleFloat e m)) 0

        asin = self
250       where
            c = getCache1 self
            self x = withExpExtended x $ \m e ->
              if e < cRangeMin c
              then x
255           else expExtended (asin (scaleFloat e m)) 0

        acos x = withExpExtended x $ \m e -> expExtended (acos (scaleFloat e m)) 0

        atan = self
260       where
            c = getCache1 self
            self x = withExpExtended x $ \m e -> case () of
              _ | e == 0 -> expExtended (atan m) 0
                | e <= cRangeMin c -> x
265             | m < 0 -> negate (atan (negate x))
                | e >= cRangeMax c -> pi/2 - atan (recip x)
                | otherwise -> expExtended (atan (scaleFloat e m)) 0

        sinh = self
270       where
            c = getCache1 self
            self x = withExpExtended x $ \m e -> case () of
              _ | e <= cRangeMin c -> x
                | cExpMin c <= e && e <= cExpMax c ->
275                 expExtended (sinh (scaleFloat e m)) 0
                | e >= cExpSup c -> expExtended (signum m / 0) 0
                | otherwise -> (exp x - exp (-x)) / 2

        cosh = self
280       where
            c = getCache1 self
```

```
      self x = withExpExtended x $ \m e -> case () of
        _ | e == 0 -> expExtended (cosh m) 0
          | cExpMin c <= e && e <= cExpMax c ->
              expExtended (cosh (scaleFloat e m)) 0
          | e >= cExpSup c -> expExtended (1/0) 0
          | otherwise -> (exp x + exp (-x)) / 2

  tanh = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e -> case () of
        _ | e == 0 -> expExtended (tanh m) 0
          | e <= cRangeMin c -> x
          | e < cRangeMax c -> expExtended (tanh (scaleFloat e m)) 0
          | otherwise -> signum x

  asinh = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e -> case () of
        _ | e == 0 -> expExtended (asinh m) 0
          | e <= cRangeMin c -> x
          | e < cRangeMax c -> expExtended (asinh (scaleFloat e m)) 0
          | m < 0 -> negate (asinh (negate x))
          | otherwise -> log x + log 2
              -- x + sqrt (x^2 + 1) == 2 * x for huge x and small precision

  acosh = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e -> case () of
        _ | e == 0 -> expExtended (acosh m) 0
          | e < cRangeMax c -> expExtended (acosh (scaleFloat e m)) 0
          | m < 0 -> acosh (negate x)
          | otherwise -> log x + log 2
              -- x + sqrt (x^2 - 1) == 2 * x for huge x and small precision

  atanh = self
    where
     c = getCache1 self
     self x = withExpExtended x $ \m e -> case () of
        _ | e == 0 -> expExtended (atanh m) 0
          | e <= cRangeMin c -> x
          | otherwise -> expExtended (atanh (scaleFloat e m)) 0

  log1p = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e -> case () of
        _ | e <= cRangeMin c -> x
          | e >= cRangeMax c -> log (1 + x)
          | otherwise -> expExtended (log1p (scaleFloat e m)) 0

  expm1 = self
    where
      c = getCache1 self
      self x = withExpExtended x $ \m e -> case () of
```

```
          _ | e <= cRangeMin c -> x
            | e >= cExpMax c -> exp x - 1
            | otherwise -> expExtended (expm1 (scaleFloat e m)) 0

    log1pexp = log1p . exp

    log1mexp x
      | x <= negate (log 2) = log1p (negate (exp x))
      | otherwise = log (negate (expm1 x))

instance ExpExtendable a => RealFloat (ExpExtended a) where
    floatRadix = self
      where
        c = getCacheIn1 self
        self _ = cRadix c

    floatDigits = self
      where
        c = getCacheIn1 self
        self _ = cDigits c

    floatRange _ = (minExponent, maxExponent)

    decodeFloat x = withExpExtended x $ \m e -> case decodeFloat m of
      (n, e') -> (n, e + e')
    encodeFloat n e = expExtended (encodeFloat n 0) e

    significand x = withExpExtended x $ \m _ -> expExtended m 0
    exponent x = withExpExtended x $ \_ e -> e

    scaleFloat = self
      where
        minExponentI = toInteger minExponent
        maxExponentI = toInteger maxExponent
        self 0 x = x
        self n x = withExpExtended x $ \m e -> case () of
          _ | m == 0 || isInfinite m || isNaN m -> x
            | e == 0 && minExponent <= n && n <= maxExponent ->
              unsafeExpExtended m n
            | minExponent <= n && n <= maxExponent &&
              minExponent <= ne && ne <= maxExponent ->
              unsafeExpExtended m ne
            | minExponentI <= neI && neI <= maxExponentI ->
              unsafeExpExtended m (fromInteger neI)
            | neI < minExponentI -> unsafeExpExtended (signum m * 0) 0
            | maxExponentI < neI -> unsafeExpExtended (signum m / 0) 0
            where
              ne = n + e
              neI = toInteger n + toInteger e

    isNaN x = withExpExtended x $ \m _ -> isNaN m
    isInfinite x = withExpExtended x $ \m _ -> isInfinite m
    isDenormalized _ = False
    isNegativeZero x = withExpExtended x $ \m _ -> isNegativeZero m
    isIEEE _ = False -- what does this really mean?

    atan2 y x = withExpExtended y $ \my ey -> withExpExtended x $ \mx ex ->
```

```
    case negate (max ey ex) of
      e | ex == 0 && ey == 0 -> expExtended (atan2 my mx) 0
        | ex == 0 -> expExtended (atan2 (scaleFloat ey my) mx) 0
        | ey == 0 -> expExtended (atan2 my (scaleFloat ex mx)) 0
        | otherwise -> expExtended (atan2 (scaleFloat e my) (scaleFloat e mx)) 0

instance ExpExtendable a => Enum (ExpExtended a) where
  -- based on 'compensated' (c) Edward Kmett 2013 (license: BSD3)
  -- <http://hackage.haskell.org/package/compensated-0.6.1/docs/src/Numeric-
      ↳ Compensated.html#line-436>
  succ a = a + 1
  pred a = a - 1
  toEnum = fromIntegral
  fromEnum = round
  enumFrom x = x : enumFrom (x + 1)
  enumFromThen x y = x : enumFromThen y (y - x + y)
  enumFromTo x y
    | x <= y = x : enumFromTo (x + 1) y
    | otherwise = []
  enumFromThenTo a b c
    | a <= b    = up a
    | otherwise = down a
    where
     delta = b - a
     up x | x <= c    = x : up (x + delta)
          | otherwise = []
     down x | c <= x    = x : down (x + delta)
            | otherwise = []

--  TODO: instances for storable, deepseq, unboxed vectors, ...
```

6 src/Numeric/ExpExtended/Internal.hs

```
-- | Internal stuff.
module Numeric.ExpExtended.Internal
  ( Cache(..)
  , cacheDefault
  , minExponent
  , maxExponent
  ) where

import Data.Bits (bit, shiftL, shiftR)

-- | Cache of useful magic values.
data Cache a = Cache
  { cRadix       :: !Integer
  -- ^ base 'floatRadix'
  , cDigits      :: !Int
  -- ^ base 'floatDigits'
  , cRangeMin    :: !Int
  -- ^ base 'fst' . 'floatRange'
  , cRangeMax    :: !Int
  -- ^ base 'snd' . 'floatRange'
  , cSupExponent :: !Int
  -- ^ magic for overflow checks
  --
  --   > (finite :: a) && supExponent <= e ==> maxExponent < exponent finite + e
```

```
    , cInfExponent :: !Int
    -- ^ magic for underflow checks
    --
    --   > (finite :: a) && e <= infExponent ==> exponent finite + e < minExponent
    , cUpShift     :: Integer -> Int -> Integer
    -- ^ radix-aware 'shiftL'
    , cDownShift   :: Integer -> Int -> Integer
    -- ^ radix-aware 'shiftR'
    , cRadixPower  :: Int -> Integer
    -- ^ radix-aware 'bit'
    , cExpMin      :: !Int
    -- ^ smaller than this exponent and base 'exp' is 1
    , cExpMax      :: !Int
    -- ^ larger  than this exponent and base 'exp' overflows to inf or 0
    , cExpInf      :: !Int
    -- ^ smaller than this exponent and extended 'exp' is 1
    , cExpSup      :: !Int
    -- ^ larger  than this exponent and extended 'exp' overflows to inf or 0
    , cLogRadix    :: !a
    -- ^ base 'log' . 'fromInteger' . 'floatRadix'
    , cRadix'      :: !a
    -- ^ base 'fromInteger' . 'floatRadix'
    }

-- | Calculate the magic values at a type.
cacheDefault :: RealFloat a => Cache a
cacheDefault = self
  where
    x = unProxy self
    self = Cache
           { cRadix = floatRadix x
           , cDigits = floatDigits x
           , cRangeMin = fst (floatRange x)
           , cRangeMax = snd (floatRange x)
           , cSupExponent = supExponentDefault x
           , cInfExponent = infExponentDefault x
           , cUpShift = case floatRadix x of
               2 -> shiftL
               d -> \n e -> n * (d ^ e)
           , cDownShift = case floatRadix x of
               2 -> shiftR
               d -> \n e -> n `div` (d ^ e)
           , cRadixPower = case floatRadix x of
               2 -> bit
               d -> \e -> d ^ e
           , cExpMin = expMinDefault x
           , cExpMax = expMaxDefault x
           , cExpInf = expInfDefault x
           , cExpSup = expSupDefault x
           , cLogRadix = log (fromIntegral (floatRadix x))
           , cRadix' = fromIntegral (floatRadix x)
           }

unProxy :: proxy a -> a
unProxy _ = undefined

margin :: Int
```

```
  -- for handling small changes in exponents in addition to large changes
  -- TODO prove this works properly
  margin = 2

  -- | Maximum exponent.
  --
  --   As big as possible without requiring more expensive overflow checks.
  maxExponent :: Int
  maxExponent = div maxBound 2 - margin

  -- | Minimum exponent.
  --
  --   As small as possible without requiring more expensive overflow checks.
  minExponent :: Int
  minExponent = div minBound 2 + margin

  {-
  b ^ maxExponent = exp (b^e)
  maxExponent * log b = b ^ e
  maxExponent * log b = exp (e * log b))
  log (maxExponent * log b) / log b = e
  -}
  expSupDefault :: RealFloat a => a -> Int
  expSupDefault m = floor $ logBase b (fromIntegral maxExponent * log b)
    where
      b :: Double
      b = fromIntegral (floatRadix m)

  expInfDefault :: RealFloat a => a -> Int
  expInfDefault m = negate (floatDigits m)

  expMaxDefault :: RealFloat a => a -> Int
  expMaxDefault m = floor $ logBase b (log (scaleFloat (snd (floatRange m)) (recip
      b)))
    where b = fromIntegral (floatRadix m) `asTypeOf` m

  expMinDefault :: RealFloat a => a -> Int
  expMinDefault m = negate (floatDigits m)


  -- Exponents larger than this will always overflow.
  -- Smaller exponents might still overflow, depending on the base exponent.
  supExponentDefault :: RealFloat a => a -> Int
  supExponentDefault x
    = maxExponent
    - fst (floatRange x)
    + floatDigits x

  -- Exponents smaller than this will always underflow.
  -- Larger exponents might still underflow, depending on the base exponent.
  infExponentDefault :: RealFloat a => a -> Int
  infExponentDefault x
    = minExponent
    - snd (floatRange x)
    - floatDigits x
```