ruff

Claude Heiland-Allen

2011–2015

Contents

1 extra/island-database-csv.hs

```
module Main (main) where

import Fractal.Mandelbrot

import Control.Monad (forM_, when)
import Data.Ratio ((%), numerator, denominator)

isIsland :: AngledInternalAddress -> Bool
isIsland = null . snd . splitAddress

main :: IO ()
main = do
  putStrLn "# period,island,address,numLo,denLo,numHi,denHi,orient,size,real,↙
      ↪ imag"
```

```
  putStrLn "1,True,1,0,1,1,1,3.141592653589793,0.75,0.0,0.0"
  forM_ [1..] $ \p -> do
    let den = 2 ^ p - 1
    forM_ [1 .. (den + 1) `div` 2 - 1] $ \num -> do
      let r = ExternalAngle (num % den)
          Just q = anglePeriod r
          Just ad = angledInternalAddress r
          Just (ExternalAnglePair s@(ExternalAngle elo) (ExternalAngle ehi)) = ↙
              ↳ externalAngles ad
      when (q == p && isIsland ad && r == s) $ do
        addressToAtom ad $ \ma -> case ma of
          Nothing ->
            putStrLn
              $  show (toInteger p) ++ ",True,"
              ++ prettyAddress ad ++ ","
              ++ show (numerator elo) ++ ","
              ++ show (denominator elo) ++ ","
              ++ show (numerator ehi) ++ ","
              ++ show (denominator ehi) ++ ",?,?,?,?"
          Just a  ->
            putStrLn
              $  show (toInteger p) ++ ",True,"
              ++ prettyAddress ad ++ ","
              ++ show (numerator elo) ++ ","
              ++ show (denominator elo) ++ ","
              ++ show (numerator ehi) ++ ","
              ++ show (denominator ehi) ++ ","
              ++ show (atomOrientation a) ++ ","
              ++ show (atomSize a) ++ ","
              ++ show (realPart (atomNucleus a)) ++ ","
              ++ show (if abs (imagPart (atomNucleus a)) < auto (atomSize a / ↙
                  ↳ 16) then 0 else imagPart (atomNucleus a))
```

2 extra/raytrace2.hs

```
{-# LANGUAGE RankNTypes #-}
module Main (main) where

import Fractal.Mandelbrot
import Fractal.Mandelbrot.Numeric.RayTraceForward4ConvergeP (convergeRaysP)

import Control.Monad (forM_, replicateM_, when)
import Data.IORef (IORef, newIORef, readIORef, writeIORef)
import Data.Maybe (fromJust)
import Data.Ratio ((%), numerator, denominator)
import Data.Time.Clock (getCurrentTime, diffUTCTime, NominalDiffTime)
import System.IO (hPutStrLn, stderr, hFlush, stdout)
import System.Random (randomRIO)

import Debug.Trace (traceShow)

modifyIORef'' :: IORef (Double, Double) -> ((Double, Double) -> (Double, Double)↙
    ↳ ) -> IO ()
modifyIORef'' r f = do
  p <- readIORef r
  let q@(x, y) = f p
  x `seq` y `seq` writeIORef r q
```

```
time :: IO r -> IO (Double, r)
time f = do
  t0 <- getCurrentTime
  r <- f
  t1 <- getCurrentTime
  return (s $ t1 `diffUTCTime` t0, r)
  where
    s = realToFrac :: NominalDiffTime -> Double

eprint :: String -> IO ()
eprint = hPutStrLn stderr

isIsland = null . snd . splitAddress

prettyAngledInternalAddress (Angled (Period p) (InternalAngle r) a) = show p ++ (if r == 0.5 then " " else " " ++ show (numerator r) ++ "/" ++ show (denominator r) ++ " ") ++ prettyAngledInternalAddress a
prettyAngledInternalAddress (Unangled (Period p)) = show p

main = do
  forM_ (iterate (2 *) 128) $ \p -> do
    stats <- newIORef (0, 0)
    let den = 2 ^ p - 1
    replicateM_ 2 $ do
      num <- randomRIO (1, den - 1)
      let r = ExternalAngle (num % den)
          Just q = anglePeriod r
          Just ad = angledInternalAddress r
          Just ea = externalAngles ad
      when (q == p && isIsland ad) $ do
        putStrLn (prettyAngledInternalAddress ad)
        (tP, (dP, eP)) <- time (convergeRaysP ea $ \c e -> c .@$ (precision c * 2) $ \c' -> convergeAtom p c' (\(Just x) -> return (toDAtom x, toDFloat e)))
        withDAtom dP print
        (t,  (d , e )) <- time (convergeRays  ea $ \c e -> c .@$ (precision c * 2) $ \c' -> convergeAtom p c' (\(Just x) -> return (toDAtom x, toDFloat e)))
        withDAtom d print
        withDAtom d $ \(Just a) -> withDAtom dP $ \(Just aP) -> if check a aP
          then modifyIORef'' stats (\(t0, tP0) -> (t0 + t, tP0 + tP))
          else putStrLn "FAIL"
    (t, tP) <- readIORef stats
    putStrLn (show (fromIntegral p :: Integer) ++ " " ++ show (t / tP))
    hFlush stdout

check a0 a1
  =  atomShape a0 == atomShape a1
  && atomPeriod a0 == atomPeriod a1
  && 0.5 < s && s < 2
  && magnitude d < auto s / 16
  where
    s = atomSize a0 / atomSize a1
    d = atomNucleus a0 - auto (atomNucleus a1)
```

3 extra/raytrace.hs

```
{-# LANGUAGE RankNTypes #-}
module Main (main) where

import Fractal.Mandelbrot
import Fractal.Mandelbrot.Numeric.RayTraceForward4ConvergeP (convergeRaysP)

import Control.Monad (forM_, replicateM_, when)
import Data.IORef (IORef, newIORef, readIORef, writeIORef)
import Data.Maybe (fromJust)
import Data.Ratio ((%))
import Data.Time.Clock (getCurrentTime, diffUTCTime, NominalDiffTime)
import System.IO (hPutStrLn, stderr, hFlush, stdout)
import System.Random (randomRIO)

modifyIORef'' :: IORef (Double, Double) -> ((Double, Double) -> (Double, Double)) -> IO ()
modifyIORef'' r f = do
  p <- readIORef r
  let q@(x, y) = f p
  x `seq` y `seq` writeIORef r q

time :: ((forall n . NaturalNumber n => CF n -> F n -> IO String) -> IO String) -> (forall n . NaturalNumber n => CF n -> IO String) -> IO String
time f g = do
  t0 <- getCurrentTime
  f (\c _ -> do
    t1 <- getCurrentTime
    r  <- g c
    t2 <- getCurrentTime
    return (show (s $ t1 `diffUTCTime` t0, s $ t2 `diffUTCTime` t1, s $ t2 `diffUTCTime` t0, r)))
  where
    s = realToFrac :: NominalDiffTime -> Double

eprint :: String -> IO ()
eprint = hPutStrLn stderr

main = do
  forM_ [2 ..] $ \p -> do
    stats <- newIORef (0, 0)
    let den = 2 ^ p - 1
    replicateM_ 10 $ do
      num <- randomRIO (1, den - 1)
      let r = ExternalAngle (num % den)
      when (anglePeriod r == Just p) $ do
        let Just ea = externalAngles =<< angledInternalAddress r
        eprint (show r)
        s  <- time (convergeRays  ea) (\c -> convergeAtom p c (\x -> return (show x)))
        eprint s
        sP <- time (convergeRaysP ea) (\c -> convergeAtom p c (\x -> return (show x)))
        eprint sP
        let (t10, t21, t20, s') = read s :: (Double, Double, Double, String)
            Just a = read s' :: Maybe (Atom N53)
```

```
            (t10P, t21P, t20P, s'P) = read sP :: (Double, Double, Double, String↙
                ↳ )
            Just aP = read s'P :: Maybe (Atom N53)
        if check a aP then modifyIORef'' stats (\(t, tP) -> (t + t20, tP + t20P)↙
            ↳ ) else print a >> print aP
    (t, tP) <- readIORef stats
    putStrLn (show (fromIntegral p :: Integer) ++ " " ++ show (t / tP))
    hFlush stdout

check a0 a1
  =  atomShape a0 == atomShape a1
  && atomPeriod a0 == atomPeriod a1
  && 0.5 < s && s < 2
  && magnitude d < auto s / 16
  where
    s = atomSize a0 / atomSize a1
    d = atomNucleus a0 - atomNucleus a1
```

4 Fractal/Mandelbrot.hs

```
{- |
Module      :  Fractal.Mandelbrot
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Convenience module importing everything.

-}
module Fractal.Mandelbrot
  ( module Fractal.Mandelbrot.Symbolic
  , module Fractal.Mandelbrot.Numeric
  , module Fractal.Mandelbrot.Utils
  , module Numeric.Rounded
  ) where

import Fractal.Mandelbrot.Symbolic
import Fractal.Mandelbrot.Numeric
import Fractal.Mandelbrot.Utils
import Numeric.Rounded
```

5 Fractal/Mandelbrot/Numeric/Atom.hs

```
{-# LANGUAGE BangPatterns, DataKinds, DeriveDataTypeable, PolyKinds, RankNTypes,↙
    ↳ ScopedTypeVariables, StandaloneDeriving #-}
{- |
Module      :  Fractal.Mandelbrot.Numeric.Atom
Copyright   :  (c) Claude Heiland-Allen 2011,2012,2014
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  BangPatterns, DataKinds, DeriveDataTypeable, PolyKinds, ↙
    ↳ RankNTypes, ScopedTypeVariables, StandaloneDeriving
```

```
10
    Mu-atom refinement.

    -}
    module Fractal.Mandelbrot.Numeric.Atom
15    ( Atom(..)
      , AtomShape(Cardioid, Circular)
      , addressShape
      , canonicalizeAtom
      , convergeAtom
20    , convergeNucleusBond
      , convergeNucleus
      , convergeBond
      , convergeInternal
      , analyseShape
25    , atomDomainSize
      , atomScaling
      , atomScale
    {-
      , DAtom
30    , toDAtom
      , fromDAtom
      , withDAtom
    -}
      ) where
35
    import Data.Data (Data)
    import Data.Typeable (Typeable)

    import Data.List (inits)
40  import Data.Maybe (isNothing, fromJust)

    {-
    import Numeric.VariablePrecision
      ( VariablePrecision, NaturalNumber, Zero
45    , F, adjustPrecision, (-@?), precision, (.@$)
      , CF, (.*), magnitudeSquared, sqr, phase, cis, scaleVComplex
      , toComplexDouble, fromComplexDouble, F24
      , DFloat, Complex, fromComplexDFloat, toComplexDFloat, withComplexDFloat
      )
50  -}

    import Fractal.Mandelbrot.Symbolic.ExternalAngle (Period)
    import Fractal.Mandelbrot.Symbolic.InternalAngle (InternalAngle)
    import Fractal.Mandelbrot.Symbolic.AngledInternalAddress (AngledInternalAddress,↙
        ↳ splitAddress)
55
    import Fractal.Mandelbrot.Utils
    import Data.Proxy
    import Data.Complex
    import Numeric.Rounded
60
    type F p = Rounded TowardNearest p
    type CF p = Complex (Rounded TowardNearest p)

    -- | An atom is a particular hyperbolic component, expressed in terms of
65  --   its concrete location in the complex plane.
```

```
data Atom (p :: *) =
  Atom
  { atomNucleus :: CF p
  , atomSize    :: Rounded TowardNearest 24
  , atomOrientation :: Double
  , atomPeriod  :: Period
  , atomShape :: AtomShape
  }
  deriving (Eq, Show, Typeable)
--deriving instance Precision p => Show (Atom p)

-- | The shape of an atom.
data AtomShape = Cardioid | Circular
  deriving (Eq, Ord, Enum, Bounded, Read, Show, Data, Typeable)

-- | The shape of the atom corresponding to an address.
addressShape :: AngledInternalAddress {- ^ address -} -> AtomShape
addressShape addr = case splitAddress addr of
  (_, []) -> Cardioid
  _ -> Circular


(.@$) :: forall (p :: *) r . (Precision p) => CF p -> Int -> (forall (q :: *) . ↙
    ↳ (Precision q) => CF q -> r) -> r
(.@$) c p f = reifyPrecision p (\prox ->f (c `asComplexType` prox))
  where
    asComplexType :: (Precision s, Precision t) => CF s -> Proxy t -> CF t
    asComplexType x _ = recodeComplex x

-- | Given an over-precise atom, canonicalize it to its sanest precision.
canonicalizeAtom :: forall (p :: *) a . Precision p => Atom p -> (forall r . (↙
    ↳ Precision r) => Atom r -> a) -> a
canonicalizeAtom a f = atomNucleus a .@$ p $ \n -> f a{ atomNucleus = n}
  where
    p = ceiling (24 - logBase 2 (atomSize a))

-- | Given the period and approximate nucleus location, refine this
--   estimate to an atom description, with incrementally increased
--   precision until the size is known to sufficiently useful effective
--   precision.
convergeAtom :: forall (p :: *) a . (Precision p) => Period {- ^ period -} -> CF↙
    ↳  p {- ^ nucleus estimate -} -> (forall (r :: *) . (Precision r) => Maybe (↙
    ↳ Atom r) -> a) -> a
convergeAtom period c f = convergeNucleusBond period c $ \mnb -> case mnb of
  Just (nucleus, bond) ->
    let delta = bond - nucleus
    in  case analyseShape period nucleus of
       Just shape -> canonicalizeAtom Atom
        { atomNucleus = nucleus
        , atomSize = sqrt (recodeFloat (magnitudeSquared delta))
        , atomOrientation = phase . recodeComplex $ delta
        , atomPeriod = period
        , atomShape = shape
        } (\x -> f (Just x))
       Nothing -> reifyPrecision 0 (\p -> f (Nothing `asMayAtom` p))
  Nothing -> reifyPrecision 0 (\p -> f (Nothing `asMayAtom` p))
  where
```

```
    asMayAtom :: Maybe (Atom q) -> Proxy q -> Maybe (Atom q)
    asMayAtom m _ = m

-- | Given the period and approximate nucleus location, successively
--   refine this estimate to the true nucleus and 1/2 bond point, with
--   incrementally increased precision until the difference between the
--   nucleus and bond is known to a sufficiently useful effective
--   precision.
convergeNucleusBond :: forall (p :: *) a . (Precision p) => Period {- ^ period -} -> CF p {- ^ nucleus estimate -} -> (forall (r :: *) . (Precision r) => Maybe (CF r, CF r) -> a) -> a
convergeNucleusBond p c f = do
  case (do
    n <- convergeNucleus p c
    b <- convergeBond p n (1/2)
    return (n, b)) of
    Nothing -> reifyPrecision 0 (\q -> f (Nothing `asMayPair` q))
    mnb@(Just (n, b)) ->
      if magnitudeSquared (n - b) > encodeFloat 1 (2 * (accuracy - fromIntegral (precision (realPart c))))
        then f mnb
        else n .@$ (precision (realPart n) * 2) $ \n' -> convergeNucleusBond p n' f
  where
    accuracy = 16 -- desired meaningful number of bits of delta
    asMayPair :: Maybe (CF q, CF q) -> Proxy q -> Maybe (CF q, CF q)
    asMayPair m _ = m


-- | Given the period and approximate location, successively refine
--   this estimate to a nucleus.
--
--   The algorithm is based on Robert Munafo's page
--   /Newton-Raphson method/
--   <http://mrob.com/pub/muency/newtonraphsonmethod.html>.
--
convergeNucleus :: Precision p => Period {- ^ period -} -> CF p {- ^ nucleus estimate -} -> Maybe (CF p)
convergeNucleus p c0 = go c0
  where
    accuracy = 4 -- converged when delta changed at most this many least significant bits
    go !c = step p 0 0
      where
        er = 65536
        er2 = er * er
        huge z = not (magnitudeSquared z < er2)
        step !q !z !d
          | huge z = Nothing
          | q == 0 = case c - z / d of
              c' | huge c' -> Nothing
                 | (c -@? c') < accuracy -> Just c'
                 | otherwise -> go c'
          | otherwise = step (q - 1) (sqr z + c) (scaleComplex 1 (z * d) + 1)

-- | Find a bond point to an atom at a particular internal angle.
convergeBond :: Precision p => Period {- ^ period -} -> CF p {- ^ nucleus -} ->
```

```
    ↪ InternalAngle {- ^ angle -} -> Maybe (CF p)
convergeBond p c a = convergeInternal p c 1 a

-- | Find an internal point within an atom.  The supplied radius should
--   be between 0 and 1.
convergeInternal :: Precision p => Period {- ^ period -} -> CF p {- ^ nucleus -}↙
    ↪  -> F p {- ^ radius -} -> InternalAngle {- ^ angle -} -> Maybe (CF p)
convergeInternal p c0 r0 a0 = go c0 c0
  where
    accuracy = 8 -- converged when delta changed at most this many least ↙
        ↪ significant bits
    b0 = (r0:+0) * (recodeComplex . cis $ (2 * pi * realToFrac a0 :: Double))
    go !z1 !c1 = step p z1 1 0 0 0 c1 z1
    er = 65536
    er2 = er * er
    huge z = not (magnitudeSquared z < er2)
    next !dz !dc !dm !c1 !z1
      | dm == 0 = Nothing
      | huge z1 = Nothing
      | huge c1 = Nothing
      | accurate = Just c1'
      | otherwise = go z1' c1'
      where
        accurate = accurateZ && accurateC
        accurateZ = deltaZ < encodeFloat 1 (2 * (accuracy - floatDigits deltaZ))
        accurateC = deltaC < encodeFloat 1 (2 * (accuracy - floatDigits deltaC))
        deltaZ = magnitudeSquared (z1' - z1)
        deltaC = magnitudeSquared (c1' - c1)
        z1' = z1 + dz / dm
        c1' = c1 + dc / dm
    step !q !a !b !c !d !e !c1 !z1
      | q == 0 = next d0 d1 m c1 z1
      | otherwise = step (q - 1) (sqr a + c1) (scaleComplex 1 (a * b)) (↙
          ↪ scaleComplex 1 (sqr b + a * c)) (scaleComplex 1 (a * d) + 1) (↙
          ↪ scaleComplex 1 (a * e + b * d)) c1 z1
      where
        y0 = z1 - a
        y1 = b0 - b
        b1 = b - 1
        m = b1 * e - d * c
        d0 = y0 * e - d * y1
        d1 = b1 * y1 - y0 * c

-- | Given a period and nucleus, determine the shape of the corresponding
--   atom.  Might explode with division by zero if the precision of the
--   nucleus isn't sufficient.
analyseShape :: Precision p => Period {- ^ period -} -> CF p {- ^ nucleus -} -> ↙
    ↪ Maybe AtomShape
analyseShape p n
  | failed = Nothing
  | ma > mi * threshold = Just Cardioid
  | otherwise = Just Circular
  where
    failed = isNothing mdeltas || mi == 0
    threshold = 2 * 2  -- empirical guess, 1 == perfect circle
    ma = maximum deltas
    mi = minimum deltas
```

```
    deltas = fromJust mdeltas
    mdeltas = sequence
      [ (magnitudeSquared . (n -)) `fmap` convergeBond p n a
      | a <- [1/17, 1/3, 1/2, 2/3, 16/17 ] -- somewhat arbitrary
      ]

-- | Given a period and nucleus, estimate the size of its atom domain.
--   <http://mathr.co.uk/blog/2013-12-10_atom_domain_size_estimation.html>
atomDomainSize :: Precision p => Period {- ^ period -} -> CF p {- ^ nucleus -} ↙
    ↳ -> F p
atomDomainSize p c = go c 1 1 (magnitude c)
  where
    go z dc q mq
      | q == p    = mq / magnitude dc
      | mp < mq   = go z' dc' q' mp
      | otherwise = go z' dc' q' mq
      where
        z' = z * z + c
        dc' = 2 * z * dc + 1
        q' = q + 1
        mp = magnitude z

atomScaling :: Precision p => Period {- ^ period -} -> CF p {- ^ nucleus -} -> (↙
    ↳ CF p, CF p)
atomScaling p c = (beta, lambda)
  where
    lambda = product lambdas
    lambdas = drop 1 . take (fromIntegral p) . map (* 2) . iterate (\z -> z * z ↙
        ↳ + c) $ 0
    beta = sum . map (recip . product) . inits $ lambdas

atomScale :: Precision p => Period {- ^ period -} -> CF p {- ^ nucleus -} -> CF ↙
    ↳ p
atomScale p c = let (beta, lambda) = atomScaling p c in beta * lambda * lambda

{-
-- | Serializable form of Atom.
data DAtom =
  DAtom
  { datomNucleus :: Complex DFloat
  , datomSize    :: F24
  , datomOrientation :: Double
  , datomPeriod  :: Period
  , datomShape :: AtomShape
  }
  deriving (Eq, Read, Show, Typeable)

-- | Freeze an atom to serializable form.
toDAtom :: Precision p => Atom p -> DAtom
toDAtom a = DAtom
  { datomNucleus = toComplexDFloat (atomNucleus a)
  , datomSize = atomSize a
  , datomOrientation = atomOrientation a
  , datomPeriod = atomPeriod a
  , datomShape = atomShape a
  }
```

```
-- | Attempt to thaw an atom, returns Nothing on precision mismatch.
fromDAtom :: Precision p => DAtom -> Maybe (Atom p)
fromDAtom d = do
  n <- fromComplexDFloat (datomNucleus d)
  return Atom
    { atomNucleus = n
    , atomSize = datomSize d
    , atomOrientation = datomOrientation d
    , atomPeriod = datomPeriod d
    , atomShape = datomShape d
    }

-- | Thaw an atom to its natural precision.
withDAtom :: DAtom -> (forall p . Precision p => Maybe (Atom p) -> r) -> r
withDAtom d f = withComplexDFloat (datomNucleus d) $ f . fmap (\n -> Atom
    { atomNucleus = n
    , atomSize = datomSize d
    , atomOrientation = datomOrientation d
    , atomPeriod = datomPeriod d
    , atomShape = datomShape d
    })
-}
```

6 Fractal/Mandelbrot/Numeric.hs

```
{- |
Module      :  Fractal.Mandelbrot.Numeric
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Numeric algorithms for the Mandelbrot set.

-}
module Fractal.Mandelbrot.Numeric
  ( module Fractal.Mandelbrot.Numeric.Atom
  , module Fractal.Mandelbrot.Numeric.RayTraceForward
  , module Fractal.Mandelbrot.Numeric.RayTraceForward4
  , module Fractal.Mandelbrot.Numeric.RayTraceForward4Converge
  , module Fractal.Mandelbrot.Numeric.RayTraceReverse
  ) where

import Fractal.Mandelbrot.Numeric.Atom
import Fractal.Mandelbrot.Numeric.RayTraceForward
import Fractal.Mandelbrot.Numeric.RayTraceForward4
import Fractal.Mandelbrot.Numeric.RayTraceForward4Converge
import Fractal.Mandelbrot.Numeric.RayTraceReverse
```

7 Fractal/Mandelbrot/Numeric/RayTraceForward4Converge.hs

```
{-# LANGUAGE DataKinds, FlexibleContexts, RankNTypes #-}
{- |
Module      :  Fractal.Mandelbrot.Numeric.RayTraceForward4Converge
```

```
    Copyright   :  (c) Claude Heiland-Allen 2012,2014
 5  License     :  BSD3

    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
    Portability :  DataKinds, FlexibleContexts, RankNTypes
10
    Heuristics to detect convergence of 'RayTraceForward4'.

    The general idea is as follows:

15    * an island of size D is surrounded by an atom period domain of size
        proportional to sqrt D;

      * the core of the Newton's method basin of convergence is roughly the
        same as the period domain;
20
      * when all ray endpoints have atom period domain equal to the target
        period switch from tracing rays to Newton's method, using their
        midpoint as initial estimate.

25  Examples:

    > convergeRays (ExternalAnglePair (11 / 75) (51264 / 349525)) (\c r -> show (c, ↙
        ↳ r))

    -}
30  module Fractal.Mandelbrot.Numeric.RayTraceForward4Converge
      ( convergeRays
      , raysConverged
      , addressToAtom
      ) where
35
    {-
    import Numeric.VariablePrecision
      ( NaturalNumber, precision, adjustPrecision, (.@$)
      , F, CF, magnitudeSquared, magnitude, scaleVComplex
40    , N24
      )
    -}

    import Data.Complex
45  import Data.Proxy
    import Numeric.Rounded
    import Fractal.Mandelbrot.Utils

    import Fractal.Mandelbrot.Symbolic.ExternalAngle
50    ( ExternalAnglePair(ExternalAnglePair)
      , tunePair
      , anglePeriod
      , Period
      )
55  import Fractal.Mandelbrot.Numeric.RayTraceForward4
      ( RayTraceForward4Callback(RayTraceForward4Callback)
      , rayTraceForward4
      , rayTraceForwardStart4
      )
```

```
60  import Fractal.Mandelbrot.Symbolic.AngledInternalAddress
      ( AngledInternalAddress
      , externalAngles
      , addressPeriod
      )
65  import Fractal.Mandelbrot.Numeric.Atom
      ( Atom
      , convergeAtom
      )

70  type F p = Rounded TowardNearest p
    type CF p = Complex (Rounded TowardNearest p)

    -- | Given a ray pair converging at the root of a hyperbolic component,
    --   find an estimate of its center and location using 'RayTraceForward4'.
75  convergeRays
      :: ExternalAnglePair {- ^ root angles -}
      -> (forall r . Precision r => CF r -> F r -> a) {- ^ callback -}
      -> a
    convergeRays o@(ExternalAnglePair ol oh) f =
80    let ExternalAnglePair il ih = tunePair h o
          h = ExternalAnglePair (1/3) (2/3)
          Just p = anglePeriod ol
          callback = RayTraceForward4Callback $ \continue zs -> case raysConverged p↙
              ↳ f zs of
            Nothing -> rayTraceForward4 continue callback
85          Just j -> j
      in  rayTraceForward4 (rayTraceForwardStart4 (ol, oh, il, ih)) callback

    -- | Lift convergence heuristics to multi-precision input.
    raysConverged
90    :: (Precision p, Precision q, Precision s, Precision t)
      => Period
      -> (forall r . Precision r => CF r -> F r -> a)
      -> (CF p, CF q, CF s, CF t)
      -> Maybe a
95  raysConverged p f = upConvert4 (raysConverged' p f)

    -- | Convergence heuristics.
    raysConverged'
      :: Precision r
100   => Period
      -> (CF r -> F r -> a)
      -> (CF r, CF r, CF r, CF r)
      -> Maybe a
    raysConverged' p f (a0, a1, b0, b1)
105   | converged = Just (f c (magnitude (a - b)))
      | otherwise = Nothing
      where
        a = scaleComplex (-1) (a0 + a1)
        b = scaleComplex (-1) (b0 + b1)
110     c = scaleComplex (-1) (a  + b )
        converged = all ((p ==) . atomPeriodDomain p) [a0,a1,b0,b1]

    -- | Compute atom period domain for a point.
    atomPeriodDomain
115   :: Precision r
```

```
  => Period {- ^ maximum iterations -}
  -> CF r {- ^ point -}
  -> Period
atomPeriodDomain p c = go 0 0 (-1) 1e10
  where
    go q z mq md
      | q == p = mq
      | d < md = go q' z' q' d
      | otherwise = go q' z' mq md
      where
        q' = q + 1
        z' = z * z + c
        d = magnitudeSquared z'

-- | Increase precision of all to the maximum precision of all.
upConvert4
  :: (Precision p, Precision q, Precision s, Precision t)
  => (forall r . Precision r => (CF r, CF r, CF r, CF r) -> a)
  -> (CF p, CF q, CF s, CF t)-> a
upConvert4 f (a, b, c, d)
  | p >= m = f (a , b', c', d')
  | q >= m = f (a', b , c', d')
  | s >= m = f (a', b', c , d')
  | t >= m = f (a', b', c', d )
  where
    m = maximum [p, q, s, t]
    p = precision (realPart a)
    q = precision (realPart b)
    s = precision (realPart c)
    t = precision (realPart d)
    a', b', c', d' :: Precision r => CF r
    a' = recodeComplex a
    b' = recodeComplex b
    c' = recodeComplex c
    d' = recodeComplex d

-- | Find the atom for an angled internal address.
addressToAtom
  :: AngledInternalAddress {- ^ address -}
  -> (forall r . Precision r => Maybe (Atom r) -> a) {- ^ callback -}
  -> a
addressToAtom a f = -- FIXME split to parent island and walk through children
  let p = addressPeriod a
  in  case externalAngles a of
        Nothing -> reifyPrecision 0 (\p -> f (Nothing `asMayAtom` p))
        Just es -> convergeRays es $ \c _ ->
          (c .@$ (precision (realPart c) * 2)) (\c' ->
            convergeAtom p c' f)
  where
    asMayAtom :: Maybe (Atom q) -> Proxy q -> Maybe (Atom q)
    asMayAtom m _ = m

(.@$) :: forall p r . (Precision p) => CF p -> Int -> (forall q . (Precision q) ↙
  ↳ => CF q -> r) -> r
(.@$) c p f = reifyPrecision p (\prox ->f (c `asComplexType` prox))
  where
    asComplexType :: (Precision s, Precision t) => CF s -> Proxy t -> CF t
```

```
    asComplexType x _ = recodeComplex x
```

8 Fractal/Mandelbrot/Numeric/RayTraceForward4.hs

```
{-# LANGUAGE DataKinds, FlexibleContexts, Rank2Types #-}
{- |
Module      :  Fractal.Mandelbrot.Numeric.RayTraceForward4
Copyright   :  (c) Claude Heiland-Allen 2012,2014
License     :  BSD3

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

Tracing four external rays simultaneously (one pair landing at the root
and the other at the 1/2 bond point) can be used to locate a particular
hyperbolic component.

Example:

> main :: IO ()
> main = do
>   let callback = RayTraceForward4Callback $ \continue zs@(z1, z2, z3, z4) ->
>         show zs : rayTraceForward4 continue callback
>       ray angle = rayTraceForward4 (rayTraceForwardStart4 (ol, oh, il, ih)) ↙
    ↳ callback
>         where Just o@(ExternalAnglePair ol oh) = externalAngles =<< ↙
    ↳ angledInternalAddress angle
>               ExternalAnglePair il ih = tunePair h o
>               h = ExternalAnglePair (1/3) (2/3)
>   putStrLn . unlines . take 256 . ray $ 1/71

-}
module Fractal.Mandelbrot.Numeric.RayTraceForward4
  ( RayTraceForward4Callback(RayTraceForward4Callback)
  , RayTraceForward4(rayTraceForward4)
  , rayTraceForwardStart4
  ) where

--import Control.Parallel (par, pseq)
{-
import Numeric.VariablePrecision
  ( NaturalNumber
  , CF
  )
-}

import Fractal.Mandelbrot.Symbolic.ExternalAngle (ExternalAngle)
import Fractal.Mandelbrot.Numeric.RayTraceForward (RayTraceForwardCallback(↙
    ↳ RayTraceForwardCallback), RayTraceForward(rayTraceForward), ↙
    ↳ rayTraceForwardStart)

import Data.Complex
import Numeric.Rounded

type CF p = Complex (Rounded TowardNearest p)
```

```
-- | For each four points on the rays, a callback gets passed the current
--   coordinates and aspect ratio, along with a continuation that can be
--   used to get more points.
data RayTraceForward4Callback a =
  RayTraceForward4Callback
  { rayTraceForward4Callback
      :: (Precision p, Precision q, Precision s, Precision t)
      => RayTraceForward4 a
      -> (CF p, CF q, CF s, CF t)
      -> a
  }

-- | Step along the four rays by one point each, calling the callback.
data RayTraceForward4 a =
  RayTraceForward4
  { rayTraceForward4 :: RayTraceForward4Callback a -> a }

-- | Initialize ray tracing for an quadruple of external angles.
--
--   Precondition:
--     @(ol, oh, il, ih)@ forms two ray pairs, with @(ol, oh)@ landing
--     at the root cusp and @(il, ih)@ landing at the 1/2 bond point
--     of the same hyperbolic component.
rayTraceForwardStart4
  :: (ExternalAngle, ExternalAngle, ExternalAngle, ExternalAngle)
  -> RayTraceForward4 a
rayTraceForwardStart4 (ol, oh, il, ih) =
  RayTraceForward4
  { rayTraceForward4 = rayTraceForwardStep4
      ( rayTraceForwardStart ol
      , rayTraceForwardStart oh
      , rayTraceForwardStart il
      , rayTraceForwardStart ih
      )
  }

rayTraceForwardStep4
  :: (RayTraceForward a, RayTraceForward a, RayTraceForward a, RayTraceForward a↙
    ↳ )
  -> RayTraceForward4Callback a
  -> a
rayTraceForwardStep4 (s4ol, s4oh, s4il, s4ih) callback =
  rayTraceForward s4ol $ RayTraceForwardCallback $ \col zol _ ->
  rayTraceForward s4oh $ RayTraceForwardCallback $ \coh zoh _ ->
  rayTraceForward s4il $ RayTraceForwardCallback $ \cil zil _ ->
  rayTraceForward s4ih $ RayTraceForwardCallback $ \cih zih _ ->
--  rayTraceForward4Callback callback (col `par` coh `par` cil `par` cih `par` ↙
    ↳ zol `par` zoh `par` zil `par` zih `pseq` RayTraceForward4
  rayTraceForward4Callback callback RayTraceForward4
    { rayTraceForward4 = rayTraceForwardStep4 (col, coh, cil, cih) }
    (zol, zoh, zil, zih)
```

9 Fractal/Mandelbrot/Numeric/RayTraceForward.hs

```
{-# LANGUAGE BangPatterns, DataKinds, FlexibleContexts, Rank2Types, ↙
    ↳ TypeOperators #-}
{- |
```

```
Module      :  Fractal.Mandelbrot.Numeric.RayTraceForward
Copyright   :  (c) Claude Heiland-Allen 2011,2012,2014
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  BangPatterns, DataKinds, FlexibleContexts, Rank2Types, ↙
    ↳ TypeOperators

Tracing external rays inwards from infinity with adaptive precision.

Example usage:

> main :: IO ()
> main = do
>   let callback = RayTraceForwardCallback $ \continue (x :+ y) _e ->
>         (show x ++ " " ++ show y) : rayTraceForward continue callback
>       ray angle = rayTraceForward (rayTraceForwardStart angle) callback
>   mapM_ (putStrLn . unlines . take 256 . ray) [0, 1/64 .. 63/64]

The algorithm is based on Tomoki Kawahira's paper
/An algorithm to draw external rays of the Mandelbrot set/
<http://www.math.nagoya-u.ac.jp/~kawahira/programs/mandel-exray.pdf>.

-}
module Fractal.Mandelbrot.Numeric.RayTraceForward
  ( RayTraceForwardCallback(RayTraceForwardCallback)
  , RayTraceForward(rayTraceForward)
  , rayTraceForwardStart
  ) where

{-
import Numeric.VariablePrecision
  ( NaturalNumber, SuccessorTo
  , VariablePrecision(adjustPrecision), (.@~)
  , F, CF, magnitudeSquared, mkPolar
  , fromComplexFloat, fromComplexDouble, sqr, scaleVComplex
  , cf8
  )
-}

import Data.Proxy
import GHC.TypeLits
import Data.Complex
import Numeric.Rounded
import Fractal.Mandelbrot.Utils
import Fractal.Mandelbrot.Symbolic.ExternalAngle (ExternalAngle, doubleAngle)


-- | For each point on the ray, a callback gets passed the current
--   coordinates and epsilon, along with a continuation that can be
--   used to get more points.
data RayTraceForwardCallback a = RayTraceForwardCallback{ ↙
    ↳ rayTraceForwardCallback :: Precision p => RayTraceForward a -> Complex (↙
    ↳ Rounded TowardNearest p) -> Rounded TowardNearest p -> a }

-- | Step along the ray by one point, calling the callback.
```

```
data RayTraceForward a = RayTraceForward { rayTraceForward :: ↙
    ↳ RayTraceForwardCallback a -> a }

-- | Initialize ray tracing for an external angle.
rayTraceForwardStart :: ExternalAngle -> RayTraceForward a
rayTraceForwardStart e = RayTraceForward{ rayTraceForward = rayTraceForwardStep ↙
    ↳ (RayTraceForwardContext e 4 4 4 65536 1 (fromComplexDouble (mkPolar 65536 ↙
    ↳ (realToFrac e)) :: Complex (Rounded TowardNearest Double)) 0 0) }

data RayTraceForwardContext p =
  RayTraceForwardContext
  { rtfcAngle :: !ExternalAngle
  , rtfcSharpness :: !Int -- number of steps to take within each dwell band
  , rtfcPrecision :: !Int -- enough bits must be available to represent the ↙
    ↳ delta with this much effective precision
  , rtfcAccuracy  :: !Int -- scales epsilon relative to the length of the last ↙
    ↳ step
  , rtfcEscapeR :: !Double
  , rtfcEpsilon2 :: !(Rounded TowardNearest p)
  , rtfcLastC :: !(Complex (Rounded TowardNearest p))
  , rtfcStepK :: !Int
  , rtfcStepJ :: !Int
  }

rayTraceForwardStep :: Precision q => RayTraceForwardContext q -> ↙
    ↳ RayTraceForwardCallback a -> a
rayTraceForwardStep rtfc0 go = step rtfc0 go
  where
    step :: Precision q => RayTraceForwardContext q -> RayTraceForwardCallback a↙
        ↳  -> a
    step rtfc g
      | rtfcStepJ rtfc >= rtfcSharpness rtfc = step rtfc
          { rtfcAngle = doubleAngle (rtfcAngle rtfc)
          , rtfcStepK = rtfcStepK rtfc + 1
          , rtfcStepJ = 0
          } g
      | otherwise = newton (rtfcLastC rtfc) (rtfcEpsilon2 rtfc) 0 g
      where
        limit = 64 -- FIXME arbitrary Newton iteration count limit
        r0 = rtfcEscapeR rtfc ** ((1/2) ** (fromIntegral (rtfcStepJ rtfc + 1) / ↙
            ↳ fromIntegral (rtfcSharpness rtfc)))
        t0 = fromComplexDouble $ mkPolar r0 (2 * pi * realToFrac (rtfcAngle rtfc↙
            ↳ ))
        newton :: Precision p => Complex (Rounded TowardNearest p) -> Rounded ↙
            ↳ TowardNearest p -> Int -> RayTraceForwardCallback a -> a
        newton !z !e2 !p f
          | enoughBits && converged =
              let lastC = recodeComplex (rtfcLastC rtfc)
                  eps' = scaleFloat (negate (2 * rtfcAccuracy rtfc)) (↙
                      ↳ magnitudeSquared (z' - lastC))
              in  rayTraceForwardCallback f (RayTraceForward (↙
                  ↳ rayTraceForwardStep rtfc
                    { rtfcStepJ = rtfcStepJ rtfc + 1
                    , rtfcLastC = z'
                    , rtfcEpsilon2 = eps'
                    })) z' eps'
          | enoughBits && p < limit = newton z' e2 (p + 1) f
```

```
          | otherwise = reifyPrecision (1 + precision (realPart z)) (\q ->
             newton (bumpPrecisionC z' q) (bumpPrecision e2 q) 0 f)
          where
105         enoughBits = negate (exponent e2) < 2 * (floatDigits e2 - rtfcPrecision rtfc)
            converged  = delta < e2
            delta = magnitudeSquared (z' - z)
            d = (cc - recodeComplex t0) / dd
            z' = z - d
110         (cc, dd) = ncnd (rtfcStepK rtfc + 1)
            ncnd 1 = (z, 1)
            ncnd i = let (!nc, !nd) = ncnd (i - 1) in (sqr nc + z, scaleComplex 1 (nc * nd) + 1)

--bumpPrecision :: (VariablePrecision t, NaturalNumber p) => t p -> t (SuccessorTo p)
115 bumpPrecision :: (Precision p, Precision q) => Rounded TowardNearest p -> Proxy q -> Rounded TowardNearest q -- + 1
bumpPrecision x _ = recodeFloat x
bumpPrecisionC :: (Precision p, Precision q) => Complex (Rounded TowardNearest p) -> Proxy q -> Complex (Rounded TowardNearest q) -- + 1
bumpPrecisionC x _ = recodeComplex x
```

10 Fractal/Mandelbrot/Numeric/RayTraceReverse.hs

```
{-# LANGUAGE BangPatterns, DataKinds, Rank2Types #-}
{- |
Module      :  Fractal.Mandelbrot.Numeric.RayTraceReverse
Copyright   :  (c) Claude Heiland-Allen 2011,2012,2014
 5 License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  BangPatterns, DataKinds, Rank2Types
10
Exterior parameters near the boundary can be traced outwards to compute
external angles.

Example usage:
15
> main :: IO ()
> main = do
>   let callback bs = RayTraceReverseCallback $ \m -> case m of
>         Nothing -> []
20 >         Just (continue, m') -> case m' of
>           Nothing -> let r = fromBits bs % (bit (length bs) - 1) in
>             "DONE" : ("\t" ++ show r) : ("\t" ++ show (fromRational r)) : rayTraceReverse continue (callback bs)
>           Just (c, m'') -> case m'' of
>             Nothing    -> (" \t" ++ show c) : rayTraceReverse continue (callback        bs )
25 >             Just False -> ("0\t" ++ show c) : rayTraceReverse continue (callback (False:bs))
>             Just True  -> ("1\t" ++ show c) : rayTraceReverse continue (callback (True :bs))
>       ray c = rayTraceReverse (rayTraceReverseStart c) (callback [])
>   putStr . unlines . ray . fromComplex fromDouble . read . head =<< getArgs
```

```
-}

module Fractal.Mandelbrot.Numeric.RayTraceReverse
  ( RayTraceReverse(rayTraceReverse)
  , RayTraceReverseCallback(RayTraceReverseCallback)
  , rayTraceReverseStart
  ) where

import Fractal.Mandelbrot.Utils (sqr, magnitudeSquared, toComplexDouble, ↙
    ↳ fromComplexDouble, recodeComplex, scaleComplex)

{-
import Numeric.VariablePrecision
  ( NaturalNumber, adjustPrecision, F
  , Complex, CF, realPart, magnitudeSquared, sqr
  , scaleVComplex, fromComplexDouble, toComplexDouble, phase, mkPolar, ↙
    ↳ fromDouble
  )
-}
import Data.Complex
import Numeric.Rounded

-- | For each point on the ray, a callback gets passed the current
--   coordinate and a bool whenever a dwell boundary is crossed
--   along with a continuation that can be used to get more points
--   if the ray hasn't passed the escape radius.
data RayTraceReverseCallback a = RayTraceReverseCallback{ ↙
    ↳ rayTraceReverseCallback :: Precision p => Maybe (RayTraceReverse a, Maybe ↙
    ↳ (Complex (Rounded TowardNearest p), Maybe Bool)) -> a }

-- | Step along the ray by one point, calling the callback.
data RayTraceReverse  a = RayTraceReverse { rayTraceReverse  :: ↙
    ↳ RayTraceReverseCallback a -> a }

-- | Initialize ray tracing for a point.
rayTraceReverseStart :: Precision p => Complex (Rounded TowardNearest p) -> ↙
    ↳ RayTraceReverse a
rayTraceReverseStart c = RayTraceReverse{ rayTraceReverse = rayTraceReverseStep ↙
    ↳ (RayTraceReverseContext 8 4 er eps2 c n z d) }
  where
    er = 1024
    er2 = er * er
    eps2 = encodeFloat 1 (2 * (4 - floatDigits (realPart c)))
    (n, z) = iter er2 c 0 0
    d = dwell er2 n z

data RayTraceReverseContext p =
  RayTraceReverseContext
  { rtrcSharpness :: !Int -- number of steps to take within each dwell band
  --  , rtrcPrecision :: !Int -- enough bits must be available to represent the ↙
      ↳ delta with this much effective precision  FIXME implement incremental ↙
      ↳ precision reduction
  , rtrcAccuracy  :: !Int -- scales epsilon relative to the length of the last ↙
      ↳ step
  , rtrcEscapeR :: !Double
  , rtrcEpsilon2 :: !(Rounded TowardNearest p)
```

```
  , rtrcLastC :: !(Complex (Rounded TowardNearest p))
  , rtrcLastN :: !Integer
  , rtrcLastZ :: !(Complex Double)
  , rtrcLastD :: !Double
  }

iter :: Precision p => Double -> Complex (Rounded TowardNearest p) -> Integer ->↙
    ↳ Complex (Rounded TowardNearest p) -> (Integer, Complex Double)
iter er2 c = go
  where
    go !n !z
      | magnitudeSquared z' > er2 = (n, z')
      | otherwise = go (n + 1) (sqr z + c)
      where z' = toComplexDouble z

dwell :: Double -> Integer -> Complex Double -> Double
dwell er2 n z = fromIntegral n - logBase 2 (log (magnitudeSquared z) / log er2)

iterd :: Precision p => Complex (Rounded TowardNearest p) -> Integer -> Complex ↙
    ↳ (Rounded TowardNearest p) -> Complex (Rounded TowardNearest p) -> (Complex↙
    ↳ (Rounded TowardNearest p), Complex (Rounded TowardNearest p))
iterd !c !m !z !dz
  | m == 0 = (z, dz)
  | otherwise = iterd c (m - 1) (sqr z + c) (scaleComplex 1 (z * dz) + 1)

rayTraceReverseStep :: Precision p => RayTraceReverseContext p -> ↙
    ↳ RayTraceReverseCallback a -> a
rayTraceReverseStep rtrc go
  | escaped  = rayTraceReverseCallback go (Just (stop, Nothing `asTypeOf` Just ↙
      ↳ (c, Nothing)))
  | otherwise = rayTraceReverseCallback go (Just (continue, Just (newC, newB)))
  where
    stop     = RayTraceReverse{ rayTraceReverse = \go' -> ↙
        ↳ rayTraceReverseCallback go' (Nothing `asTypeOf` Just (undefined, Just ↙
        ↳ (c, Nothing))) }
    continue = RayTraceReverse
      { rayTraceReverse = rayTraceReverseStep rtrc
        { rtrcEpsilon2 = scaleFloat (negate (2 * rtrcAccuracy rtrc)) (↙
            ↳ magnitudeSquared (newC - c))
        , rtrcLastC = newC
        , rtrcLastN = newN
        , rtrcLastZ = newZ
        , rtrcLastD = newD
        }
      }
    er2 = er * er
    er = rtrcEscapeR rtrc
    eps2 = rtrcEpsilon2 rtrc
    c = rtrcLastC rtrc
    n = rtrcLastN rtrc
    z = rtrcLastZ rtrc
    d = rtrcLastD rtrc
    d' = d - 1 / fromIntegral (rtrcSharpness rtrc)
    m = ceiling d'
    r = er ** (2 ** (fromIntegral m - d'))
    a = phase z / (2 * pi)
    t = a - fromIntegral (floor a :: Int)
```

```
      k0 = recodeComplex . fromComplexDouble $ mkPolar r (2 * pi *  t     )
      k1 = recodeComplex . fromComplexDouble $ mkPolar r (    pi *  t     )
      k2 = recodeComplex . fromComplexDouble $ mkPolar r (    pi * (t + 1))
      step !k !cc =
        let (f, df) = iterd cc m 0 0
            dc = (f - k) / df
            cc' = cc - dc
        in  cc : if not (magnitudeSquared (cc' - cc) > eps2) then [] else step k cc'
      steps k = step k c
      c0 = last (steps k0)
      (c1, c2) = last (steps k1 `zip` steps k2)
      dc1 = magnitudeSquared (c1 - c)
      dc2 = magnitudeSquared (c2 - c)
      (n0, z0) = iter er2 c0 0 0
      (n1, z1) = iter er2 c1 0 0
      (n2, z2) = iter er2 c2 0 0
      d0 = dwell er2 n0 z0
      d1 = dwell er2 n1 z1
      d2 = dwell er2 n2 z2
      (newC, newN, newZ, newD, newB)
        | m == n    = (c0, n0, z0, d0, Nothing)
        | dc1 < dc2 = (c1, n1, z1, d1, if n1 == n then Nothing else Just False)
        | dc2 < dc1 = (c2, n2, z2, d2, if n2 == n then Nothing else Just True )
        | otherwise = error $ show (dc1, dc2, m, n, n1, n2)
      escaped = m <= 0
```

11 Fractal/Mandelbrot/Symbolic/AngledInternalAddress.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{- |
Module      :  Fractal.Mandelbrot.Symbolic.AngledInternalAddress
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Implementation of ideas from Dierk Schleicher's paper
/Internal Addresses Of The Mandelbrot Set And Galois Groups Of Polynomials (version of February 5, 2008)/
<http://arxiv.org/abs/math/9411238v2>.

-}
module Fractal.Mandelbrot.Symbolic.AngledInternalAddress
  ( AngledInternalAddress(Unangled, Angled)
  , angledInternalAddress
  , angledFromList
  , angledToList
  , externalAngles
  , stripAngles
  , splitAddress
  , joinAddress
  , addressPeriod
  , visibleComponents
  ) where
```

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

import Data.List (genericDrop, genericIndex, genericLength, genericReplicate)
import Data.Ratio ((%))

import Fractal.Mandelbrot.Symbolic.ExternalAngle (anglePeriod, doubleAngle, ↙
    ↳ ExternalAngle(ExternalAngle), ExternalAnglePair(ExternalAnglePair), ↙
    ↳ BinaryAnglePair(), toBinaryAnglePair, fromBinaryAnglePair, Period(Period),↙
    ↳  tuneBinaryPair, wrapAngle)
import Fractal.Mandelbrot.Symbolic.InternalAngle (InternalAngle(InternalAngle), ↙
    ↳ (*/), FareyAngle(FareyAngle), fareyAngle)
import Fractal.Mandelbrot.Symbolic.KneadingSequence (Kneading, kneading, ↙
    ↳ kneadingPeriod, unwrapKneading)
import Fractal.Mandelbrot.Symbolic.InternalAddress (InternalAddress(↙
    ↳ InternalAddress), internalAddress, internalToList)
import Fractal.Mandelbrot.Utils (chunkWith2, mod_, strictlyWithin)


rho :: Kneading -> Integer -> Integer
rho v r | r >= 1 && fmap (Period r`mod`) (kneadingPeriod v) /= Just 0 = ((1 + r)↙
    ↳  +) . genericLength . takeWhile id . zipWith (==) vs . genericDrop r $ vs
        | otherwise = rho v (r + 1)
  where
    vs = unwrapKneading v

orbit :: (a -> a) -> a -> [a]
orbit = iterate

-- | Angled internal addresses have angles between each integer in an
--   internal address.
data AngledInternalAddress
  = Unangled Period
  | Angled Period InternalAngle AngledInternalAddress
  deriving (Read, Show, Eq, Ord, Data, Typeable)

-- | Builds a valid 'AngledInternalAddress' from a list, checking the
--   precondition that only the last 'Maybe Angle' should be 'Nothing',
--   and the 'Integer' must be strictly increasing.
angledFromList :: [(Period, Maybe InternalAngle)] -> Maybe AngledInternalAddress
angledFromList = fromList' 0
  where
    fromList' x [(n, Nothing)] | n > x = Just (Unangled n)
    fromList' x ((n, Just r) : xs) | n > x && 0 < r && r < 1 = Angled n r `fmap`↙
        ↳  fromList' n xs
    fromList' _ _ = Nothing

unsafeAngledFromList :: [(Period, Maybe InternalAngle)] -> AngledInternalAddress
unsafeAngledFromList = fromList' 0
  where
    fromList' x [(n, Nothing)] | n > x = Unangled n
    fromList' x ((n, Just r) : xs) | n > x && 0 < r && r < 1 = Angled n r (↙
        ↳ fromList' n xs)
    fromList' _ _ = error "Fractal.Mandelbrot.Address.unsafeAngledFromList"

-- | Convert an 'AngledInternalAddress' to a list.
```

```
angledToList :: AngledInternalAddress -> [(Period, Maybe InternalAngle)]
angledToList (Unangled n) = [(n, Nothing)]
angledToList (Angled n r a) = (n, Just r) : angledToList a

denominators :: InternalAddress -> Kneading -> [Integer]
denominators a v = denominators' (internalToList a)
  where
    denominators' (Period s0:ss@(Period s1:_)) =
      let rr = s1 `mod_` s0
      in  (((s1 - rr) `div` s0) + if s0 `elem` takeWhile (<= s0) (orbit p rr) ↙
          ↳ then 1 else 2) : denominators' ss
    denominators' _ = []
    p = rho v

numerators :: ExternalAngle -> InternalAddress -> [Integer] -> [Integer]
numerators r a qs = zipWith num (internalToList a) qs
  where
    num s q = genericLength . filter (<= r) . map (genericIndex rs) $ [0 .. q - ↙
        ↳ 2]
      where
        rs = iterate (foldr (.) id . genericReplicate s $ doubleAngle) r

-- | The angled internal address corresponding to an external angle.
angledInternalAddress :: ExternalAngle -> Maybe AngledInternalAddress
angledInternalAddress r0 = do
  let r = wrapAngle r0
      k = kneading r
  i <- internalAddress k
  let d = denominators i k
      n = numerators r i d
  return . unsafeAngledFromList . zip (internalToList i) . (++ [Nothing]) . map ↙
      ↳ (Just . InternalAngle) . zipWith (%) n $ d

-- | Split an angled internal address at the last island.
splitAddress :: AngledInternalAddress -> (AngledInternalAddress, [InternalAngle↙
    ↳ ])
splitAddress a =
  let (ps0, rs0) = unzip $ angledToList a
      ps1 = reverse ps0
      rs1 = reverse (Nothing : init rs0)
      prs1 = zip ps1 rs1
      f ((p, Just r):qrs@((q, _):_)) acc
        | p == q */ r = f qrs (r : acc)
      f prs acc = g prs acc
      g prs acc =
        let (ps2, rs2) = unzip prs
            ps3 = reverse ps2
            rs3 = reverse (Nothing : init rs2)
            prs3 = zip ps3 rs3
            aa = unsafeAngledFromList prs3
        in  (aa, acc)
  in  f prs1 []

-- | The inverse of 'splitAddress'.
joinAddress :: AngledInternalAddress -> [InternalAngle] -> AngledInternalAddress
joinAddress (Unangled p) [] = Unangled p
joinAddress (Unangled p) (r:rs) = Angled p r (joinAddress (Unangled $ p */ r) rs↙
```

```
    ↪ )
joinAddress (Angled p r a) rs = Angled p r (joinAddress a rs)

-- | The period of an angled internal address.
addressPeriod :: AngledInternalAddress -> Period
addressPeriod (Unangled p) = p
addressPeriod (Angled _ _ a) = addressPeriod a

-- | Discard angle information from an internal address.
stripAngles :: AngledInternalAddress -> InternalAddress
stripAngles = InternalAddress . map fst . angledToList

-- | The pair of external angles whose rays land at the root of the
--   hyperbolic component described by the angled internal address.
externalAngles :: AngledInternalAddress -> Maybe ExternalAnglePair
externalAngles = externalAngles' 1 (toBinaryAnglePair (ExternalAnglePair 0 1))

externalAngles' :: Period -> BinaryAnglePair -> AngledInternalAddress -> Maybe ↙
    ↪ ExternalAnglePair
externalAngles' p0 lohi (Unangled p)
  | p0 /= p = case visibleComponents (fromBinaryAnglePair lohi) p of
      [lh] -> Just lh
      _ -> Nothing
  | otherwise = Just (fromBinaryAnglePair lohi)
externalAngles' p0 lohi a0@(Angled p r a)
  | p0 /= p = case visibleComponents (fromBinaryAnglePair lohi) p of
      [lh] -> externalAngles' p (toBinaryAnglePair lh) a0
      _ -> Nothing
  | otherwise = do
      let FareyAngle _ lh = fareyAngle r
      externalAngles' (p */ r) (if p > 1 then tuneBinaryPair lh lohi else lh) a

-- | The visible components in the wake.
visibleComponents :: ExternalAnglePair -> Period -> [ExternalAnglePair]
visibleComponents (ExternalAnglePair lo hi) q =
  let gaps (l, h) n
        | n == 0 = [(l, h)]
        | n > 0 = let gs = gaps (l, h) (n - 1)
                      cs = candidates n gs
                  in  accumulate cs gs
      -- deliberately unhandled case
      candidates n gs =
        let den = 2 ^ n - 1
        in  [ r
            | (l, h) <- gs
            , num <- [ ceiling (l * fromInteger den)
                     .. floor (h * fromInteger den) ]
            , let r = ExternalAngle $ num % den
            , l < r, r < h
            , anglePeriod r == Just n
            ]
      accumulate [] ws = ws
      accumulate (l : h : lhs) ws =
        let (ls, ms@((ml, _):_)) = break (l `strictlyWithin`) ws
            (_s, (_, rh):rs) = break (h `strictlyWithin`) ms
        in  ls ++ [(ml, l)] ++ accumulate lhs ((h, rh) : rs)
      -- deliberately unhandled case
```

```
  in  chunkWith2 ExternalAnglePair . candidates q . gaps (lo, hi) $ (q - 1)
```

12 Fractal/Mandelbrot/Symbolic/Antenna.hs

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving #-}
{- |
Module      :  Fractal.Mandelbrot.Symbolic.Antenna
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

An implementation of the algorithms described in
R L Devaney and M Moreno-Rocha's paper of April 11, 2000
/Geometry of the Antennas in the Mandelbrot Set/
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.1348&rep=rep1&type=pdf>.

-}
module Fractal.Mandelbrot.Symbolic.Antenna
  ( bulb
  , antenna
  , spokes
  ) where

import Data.List (genericTake, genericDrop)
import Data.Ratio (numerator, denominator)

import Fractal.Mandelbrot.Symbolic.ExternalAngle
import Fractal.Mandelbrot.Symbolic.InternalAngle

bulb' :: InternalAngle -> BinaryAnglePair
bulb' (InternalAngle pq) = BinaryAnglePair (BinaryAngle [] ls) (BinaryAngle [] us)
  where
    q = denominator (w pq)
    ls = is ++ [False, True]
    us = is ++ [True, False]
    is = genericTake (q - 2) . map i . iterate r $ pq
    i a = not $ 0 < a && a < 1 - pq
    r a = w (a + pq)
    w a
      | f < 0 = 1 + f
      | otherwise = f
      where
        (_, f) = properFraction a :: (Integer, Rational)

-- | Theorem 3.1: angles of the bulb.
bulb :: InternalAngle -> ExternalAnglePair
bulb = fromBinaryAnglePair . bulb'

antenna' :: BinaryAnglePair -> BinaryAnglePair
antenna' (BinaryAnglePair (BinaryAngle [] ls) (BinaryAngle [] us)) = BinaryAnglePair (BinaryAngle ls us) (BinaryAngle us ls))
```

```
-- | Proposition 3.2: angles of the antenna.
antenna :: InternalAngle -> ExternalAnglePair
antenna = fromBinaryAnglePair . antenna' . bulb'

spokes' :: FareyAngle -> FareyAngle -> [BinaryAngle]
spokes' (FareyAngle (InternalAngle ab) _) (FareyAngle (InternalAngle pq) (↙
    ↳ BinaryAnglePair (BinaryAngle [] ls) (BinaryAngle [] us))) =
  [ BinaryAngle ls (rotate (k * b) us) | k <- [ 0 .. q - p - 1 ] ] ++
  [ BinaryAngle us (rotate (k * b) us) | k <- [ q - p .. q - 1 ] ]
  where
    p = numerator pq
    q = denominator pq
    b = denominator ab
    rotate i = genericTake q . genericDrop i . cycle

-- | Theorem 5.3: angles of the spokes
spokes :: InternalAngle -> [ExternalAngle]
spokes pq = map fromBinaryAngle (spokes' fab fpq)
  where
    (fab, fcd) = fareyParents pq
    fpq = fab <+> fcd
```

13 Fractal/Mandelbrot/Symbolic/ConciseAddress.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{- |
Module      :  Fractal.Mandelbrot.Symbolic.ConciseAddress
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Concise form of angled internal addresses.

-}
module Fractal.Mandelbrot.Symbolic.ConciseAddress
  ( ConciseAddress(ConciseAddress)
  , ConciseItem(ConcisePeriod, ConciseAngle)
  , toConciseAddress
  , fromConciseAddress
  , conciseItem
  ) where

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

import Fractal.Mandelbrot.Symbolic.ExternalAngle (Period())
import Fractal.Mandelbrot.Symbolic.InternalAngle (InternalAngle(), (*/))
import Fractal.Mandelbrot.Symbolic.AngledInternalAddress (AngledInternalAddress(↙
    ↳ Angled, Unangled))

-- | Concise angled internal address type.
newtype ConciseAddress = ConciseAddress [ConciseItem]
  deriving (Read, Show, Eq, Ord, Data, Typeable)
```

```
-- | Concise addresses are mixed sequences of periods and internal angles.
data ConciseItem = ConcisePeriod !Period | ConciseAngle !InternalAngle
  deriving (Read, Show, Eq, Ord, Data, Typeable)

-- | Deconstructor for items in the vein of 'either'.
conciseItem :: (Period -> a) -> (InternalAngle -> a) -> ConciseItem -> a
conciseItem f _ (ConcisePeriod p) = f p
conciseItem _ g (ConciseAngle r) = g r

-- | Reduce an angled internal address to concise form.
toConciseAddress :: AngledInternalAddress -> ConciseAddress
toConciseAddress = ConciseAddress . rule2 1 . rule1 . toItems
  where
    toItems (Unangled q) = ConcisePeriod q : []
    toItems (Angled q r a) = ConcisePeriod q : ConciseAngle r : toItems a
    rule1 = filter (/= ConciseAngle 0.5)
    rule2 _ [] = []
    rule2 p (ConcisePeriod q : rest)
      | p == q = rule2 q rest
      | otherwise = ConcisePeriod q : rule2 q rest
    rule2 p (ConciseAngle r : rest) = ConciseAngle r : rule2 (p */ r) rest

-- | Recreate an angled internal address from concise form.
fromConciseAddress :: ConciseAddress -> AngledInternalAddress
fromConciseAddress (ConciseAddress is0) = fromItems . rule1 . rule2 1 $ is0
  where
    fromItems [ConcisePeriod p] = Unangled p
    fromItems (ConcisePeriod p : ConciseAngle r : is) = Angled p r (fromItems is↙
      ↳ )
    rule2 p [] = [ConcisePeriod p]
    rule2 p (ConcisePeriod q : is)
      | p < q = ConcisePeriod p : rule2 q is
      | otherwise = rule2 q is
    rule2 p (ConciseAngle r : is) = ConcisePeriod p : ConciseAngle r : rule2 (p ↙
      ↳ */ r) is
    rule1 (ConcisePeriod p : ConcisePeriod q : is) = ConcisePeriod p : ↙
      ↳ ConciseAngle 0.5 : rule1 (ConcisePeriod q : is)
    rule1 (i : is) = i : rule1 is
    rule1 [] = []

{-
quickCheck (\k -> let r = flip approxRational 1e-3 . abs . snd . properFraction ↙
    ↳ $ k
                  in 0 < r && r < 1 && odd (denominator r) ==>
                  (let ma = angledInternalAddress (ExternalAngle r)
                  in isJust ma ==>
                  ma == (fromConciseAddress . toConciseAddress) `fmap` ma))
-}
```

14 Fractal/Mandelbrot/Symbolic/ExternalAngle.hs

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving #-}
{- |
Module      :  Fractal.Mandelbrot.ExternalAngle
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3
```

```
    Maintainer  :  claude@mathr.co.uk
    Stability   :  unstable
    Portability :  DeriveDataTypeable, GeneralizedNewtypeDeriving

    External angles and tuning transformations.

    Described with examples in (and in many others) G Pastor et al's paper
    /Operating with external arguments in the Mandelbrot set antenna/
    <http://www.iec.csic.es/~fausto/publica/Pastor02_pre.pdf>.

    -}
    module Fractal.Mandelbrot.Symbolic.ExternalAngle
      ( ExternalAngle(ExternalAngle)
      , wrapAngle
      , doubleAngle
      , doublingPreimages
      , ExternalAnglePair(ExternalAnglePair)
      , externalAnglePair
      , BinaryAngle(BinaryAngle)
      , fromBinaryAngle
      , toBinaryAngle
      , BinaryAnglePair(BinaryAnglePair)
      , fromBinaryAnglePair
      , toBinaryAnglePair
      , tuneBinary
      , tuneBinaryPair
      , tune
      , tunePair
      , Period(Period)
      , anglePeriod
      , Preperiod(Preperiod)
      , anglePreperiod
      ) where

    import Data.Data (Data())
    import Data.Typeable (Typeable())
    import Data.Bits (testBit)
    import Data.List (genericLength)
    import Data.Ratio ((%), numerator, denominator)

    import Fractal.Mandelbrot.Utils (fromBits, genericElemIndex)

    -- | External angles define external rays, which land on the boundary.
    --   In particular, each hyperbolic component has two rays that land at
    --   its root (by convention the unique hyperbolic of period 1 has
    --   external angles 0 and 1).
    newtype ExternalAngle = ExternalAngle Rational
      deriving (Eq, Ord, Read, Show, Data, Typeable, Enum, Num, Fractional, Real, ↙
          ↳ RealFrac)

    -- | Wrap an external angle into [0, 1).
    wrapAngle :: ExternalAngle -> ExternalAngle
    wrapAngle a
      | f < 0 = 1 + f
      | otherwise = f
      where
        (_, f) = properFraction a :: (Integer, ExternalAngle)
```

```
-- | External angle doubling map.
doubleAngle :: ExternalAngle -> ExternalAngle
doubleAngle a = wrapAngle (2 * a)

-- | Pre-images under angle doubling.
doublingPreimages :: ExternalAngle -> (ExternalAngle, ExternalAngle)
doublingPreimages e0 = let e = wrapAngle e0 in (e / 2, (e + 1) / 2)

-- | A pair of external angles whose rays land at the root of the same
--   hyperbolic component, moreover @'ExternalAnglePair' lo hi@ also
--   satisfies @lo < hi@.
data ExternalAnglePair = ExternalAnglePair ExternalAngle ExternalAngle
  deriving (Eq, Ord, Read, Show, Data, Typeable)

-- | Put a pair of external angles that land at the root of the same
--   hyperbolic component into proper order.  The precondition is not
--   checked.
externalAnglePair :: ExternalAngle -> ExternalAngle -> ExternalAnglePair
externalAnglePair x y | x < y  = ExternalAnglePair x y
                      | x > y  = ExternalAnglePair y x
                      | x == 0 = ExternalAnglePair 0 1
                      -- deliberately unhandled case

-- | Binary representation of a (pre-)periodic external angle.
--   Big endian lists of bits for pre-period and period.
data BinaryAngle = BinaryAngle [Bool] [Bool]
  deriving (Eq, Ord, Read, Show, Data, Typeable)

-- | A pair of binary angles whose rays land at the root of the same
--   hyperbolic component, with ordering constraints similar to
--   'ExternalAnglePair'.
data BinaryAnglePair = BinaryAnglePair BinaryAngle BinaryAngle
  deriving (Eq, Ord, Read, Show, Data, Typeable)

-- | Convert an angle from binary representation.
fromBinaryAngle :: BinaryAngle -> ExternalAngle
fromBinaryAngle (BinaryAngle pre per)
  | n == 0    = ExternalAngle $  fromBits pre % (2 ^ m)
  | otherwise = ExternalAngle $ (fromBits pre % (2 ^ m)) + (fromBits per % (2 ^ ↙
      ↳ m * (2 ^ n - 1)))
  where
    m, n :: Integer
    m = genericLength pre
    n = genericLength per

-- | Convert an angle to binary representation.
toBinaryAngle :: ExternalAngle -> BinaryAngle
toBinaryAngle e@(ExternalAngle a)
  | a == 0 = BinaryAngle [] []
  | even (denominator a) =
      let BinaryAngle pre per = toBinaryAngle (doubleAngle e)
          b = a >= 1/2
      in  BinaryAngle (b:pre) per
  | otherwise =
      let (t, p):_ = dropWhile ((1 /=) . denominator . fst) . map (\q -> (a * ↙
          ↳ (2^q - 1), q)) $ [ 1 ..]
```

```
            s = numerator t
            per = [ s `testBit` i | i <- [p - 1, p - 2 .. 0] ]
        in  BinaryAngle [] per

-- | Convert an angle pair from binary representation.
fromBinaryAnglePair :: BinaryAnglePair -> ExternalAnglePair
fromBinaryAnglePair (BinaryAnglePair x y) = ExternalAnglePair (fromBinaryAngle x↙
    ↳ ) (fromBinaryAngle y)

-- | Convert an angle pair to binary representation.
toBinaryAnglePair :: ExternalAnglePair -> BinaryAnglePair
toBinaryAnglePair (ExternalAnglePair x y) = BinaryAnglePair (toBinaryAngle x) (↙
    ↳ toBinaryAngle y)

-- | Tuning transformation for binary represented periodic angles.
--   Probably only valid for angle pairs presenting ray pairs.
tuneBinary :: BinaryAngle -> BinaryAnglePair -> BinaryAngle
tuneBinary (BinaryAngle tpre tper) (BinaryAnglePair (BinaryAngle [] per0) (↙
    ↳ BinaryAngle [] per1))
  = BinaryAngle (concatMap f tpre) (concatMap f tper)
  where
    f False = per0
    f True  = per1
-- tuneBinary unhandled case, FIXME

-- | Tuning transformation lifted to binary pairs.
tuneBinaryPair :: BinaryAnglePair -> BinaryAnglePair -> BinaryAnglePair
tuneBinaryPair (BinaryAnglePair x y) t0t1 = BinaryAnglePair (tuneBinary x t0t1) ↙
    ↳ (tuneBinary y t0t1)

-- | Tuning transformation for external angles.  The implementation
--   transits via binary angles.
tune :: ExternalAngle -> ExternalAnglePair -> ExternalAngle
tune t t0t1 = fromBinaryAngle $ tuneBinary (toBinaryAngle t) (toBinaryAnglePair ↙
    ↳ t0t1)

-- | Tuning transformation lifted to pairs.  The implementation
--   transits via binary angle pairs.
tunePair :: ExternalAnglePair -> ExternalAnglePair -> ExternalAnglePair
tunePair xy t0t1 = fromBinaryAnglePair (tuneBinaryPair (toBinaryAnglePair xy) (↙
    ↳ toBinaryAnglePair t0t1))

-- | Periods.
newtype Period = Period Integer
  deriving (Eq, Ord, Read, Show, Data, Typeable, Enum, Num, Integral, Real)

-- | The period of an external angle, or 'Nothing' when it is pre-periodic.
anglePeriod :: ExternalAngle -> Maybe Period
anglePeriod e@(ExternalAngle r)
  | even (denominator r) = Nothing
  | otherwise = (1 +) `fmap` (genericElemIndex e . drop 1 . iterate doubleAngle)↙
      ↳ e

-- | Pre-periods.
newtype Preperiod = Preperiod Integer
  deriving (Eq, Ord, Read, Show, Data, Typeable, Enum, Num, Integral, Real)
```

```
-- | The pre-period of an external angle, 0 when it is strictly periodic.
anglePreperiod :: ExternalAngle -> Preperiod
anglePreperiod e = let BinaryAngle pre _ = toBinaryAngle e in genericLength pre
```

15 Fractal/Mandelbrot/Symbolic.hs

```
{- |
Module      :  Fractal.Mandelbrot.Symbolic
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Symbolic algorithms for the Mandelbrot set.

-}
module Fractal.Mandelbrot.Symbolic
  ( module Fractal.Mandelbrot.Symbolic.ExternalAngle
  , module Fractal.Mandelbrot.Symbolic.KneadingSequence
  , module Fractal.Mandelbrot.Symbolic.InternalAngle
  , module Fractal.Mandelbrot.Symbolic.InternalAddress
  , module Fractal.Mandelbrot.Symbolic.AngledInternalAddress
  , module Fractal.Mandelbrot.Symbolic.ConciseAddress
  , module Fractal.Mandelbrot.Symbolic.Antenna
  , module Fractal.Mandelbrot.Symbolic.Text
  ) where

import Fractal.Mandelbrot.Symbolic.ExternalAngle
import Fractal.Mandelbrot.Symbolic.KneadingSequence
import Fractal.Mandelbrot.Symbolic.InternalAngle
import Fractal.Mandelbrot.Symbolic.InternalAddress
import Fractal.Mandelbrot.Symbolic.AngledInternalAddress
import Fractal.Mandelbrot.Symbolic.ConciseAddress
import Fractal.Mandelbrot.Symbolic.Antenna
import Fractal.Mandelbrot.Symbolic.Text
```

16 Fractal/Mandelbrot/Symbolic/InternalAddress.hs

```
{-# LANGUAGE DeriveDataTypeable #-}
{- |
Module      :  Fractal.Mandelbrot.Symbolic.InternalAddress
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Implementation of ideas from Dierk Schleicher's paper
/Internal Addresses Of The Mandelbrot Set And Galois Groups Of Polynomials (↙
    ↳ version of February 5, 2008)/
<http://arxiv.org/abs/math/9411238v2>.

-}
```

```
module Fractal.Mandelbrot.Symbolic.InternalAddress
  ( InternalAddress(InternalAddress)
  , internalAddress
  , internalFromList
  , internalToList
  , associatedKneadings
  , upperKneading
  , lowerKneading
  ) where

import Data.Data (Data())
import Data.Typeable (Typeable())
import Data.List (genericLength, genericTake)

import Fractal.Mandelbrot.Symbolic.ExternalAngle (Period)
import Fractal.Mandelbrot.Symbolic.KneadingSequence (Knead(..), Kneading(↙
    ↳ Aperiodic, Periodic, StarPeriodic), unwrapKneading)
import Fractal.Mandelbrot.Utils (divisors)

-- | Internal addresses are a non-empty sequence of strictly increasing
--   periods beginning with '1'.
newtype InternalAddress = InternalAddress [Period]
  deriving (Read, Show, Eq, Ord, Data, Typeable)

-- | Construct a valid 'InternalAddress', checking the precondition.
internalFromList :: [Period] -> Maybe InternalAddress
internalFromList x0s@(1:_) = InternalAddress `fmap` fromList' 0 x0s
  where
    fromList' n [x]    | x > n = Just [x]
    fromList' n (x:xs) | x > n = (x:) `fmap` fromList' x xs
    fromList' _ _ = Nothing
internalFromList _ = Nothing

-- | Extract the sequence of periods.
internalToList :: InternalAddress -> [Period]
internalToList (InternalAddress xs) = xs

-- | Construct an 'InternalAddress' from a kneading sequence.
internalAddress :: Kneading -> Maybe InternalAddress
internalAddress (StarPeriodic [Star])      = Just (InternalAddress [1])
internalAddress (StarPeriodic v@(One:_))   = Just . InternalAddress . address'↙
    ↳ per (genericLength v) $ v
internalAddress (Periodic     v@(One:_))   = Just . InternalAddress . address'↙
    ↳ per (genericLength v) $ v
internalAddress k@(Aperiodic    (One:_))   = Just . InternalAddress . address'↙
    ↳ inf . unwrapKneading $ k
internalAddress _ = Nothing

address'inf :: [Knead] -> [Period]
address'inf v = address' v

address'per :: Period -> [Knead] -> [Period]
address'per p v = takeWhile (<= p) $ address' v

address' :: [Knead] -> [Period]
address' v = address'' 1 [One]
```

```
  where
    address'' sk vk = sk : address'' sk' vk'
      where
        sk' = (1 +) . genericLength . takeWhile id . zipWith (==) v . cycle $ vk
        vk' = genericTake sk' (cycle v)

-- | A star-periodic kneading sequence's upper and lower associated
--   kneading sequences.
associatedKneadings :: Kneading -> Maybe (Kneading, Kneading)
associatedKneadings (StarPeriodic k) = Just (Periodic a, Periodic abar)
  where
    n = genericLength k
    abar:_ = filter (and . zipWith (==) a' . cycle) . map (`genericTake` a') . ↙
        ↳ divisors $ n
    (a, a') = if ((n `elem`) . internalToList) `fmap` internalAddress (Periodic ↙
        ↳ a1) == Just True then (a1, a2) else (a2, a1)
    a1 = map (\s -> case s of Star -> Zero ; t -> t) k
    a2 = map (\s -> case s of Star -> One  ; t -> t) k
associatedKneadings _ = Nothing

-- | The upper associated kneading sequence.
upperKneading :: Kneading -> Maybe Kneading
upperKneading = fmap fst . associatedKneadings

-- | The lower associated kneading sequence.
lowerKneading :: Kneading -> Maybe Kneading
lowerKneading = fmap snd . associatedKneadings
```

17 Fractal/Mandelbrot/Symbolic/InternalAngle.hs

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving #-}
{- |
Module      :  Fractal.Mandelbrot.Symbolic.InternalAngle
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

A background to the ideas is presented in Robert L Devaney's paper
/The Mandelbrot Set and The Farey Tree/
<http://math.bu.edu/people/bob/papers/farey.ps>,
although the particular binary search algorithm used here was a
suggestion of Robert P Munafo.

-}
module Fractal.Mandelbrot.Symbolic.InternalAngle
  ( InternalAngle(InternalAngle)
  , (*/)
  , FareyAngle(FareyAngle)
  , fareyAngle
  , fareyParents
  , (<+>)
  , (</>)
  , fareyZero
  , fareyOne
```

```
  , fareyHalf
  ) where

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

import Data.Ratio ((%), numerator, denominator)

import Fractal.Mandelbrot.Symbolic.ExternalAngle (BinaryAngle(BinaryAngle), ↙
    ↳ BinaryAnglePair(BinaryAnglePair), Period(Period))

-- | Internal angles label the neighbouring descendents of hyperbolic
--   components.  The descendent of a hyperbolic component with period
--   @p@ at internal angle @a@ has period @p * denominator a@
newtype InternalAngle = InternalAngle Rational
  deriving (Eq, Ord, Read, Show, Data, Typeable, Enum, Num, Fractional, Real, ↙
      ↳ RealFrac)

-- | Multiply a period by the denominator of an internal angle.
(*/) :: Period -> InternalAngle -> Period
Period p */ InternalAngle a = Period (p * denominator a)
infixl 7 */

-- | Farey angles pair an internal angle together with the external
--   angles of the corresponding child of the top level cardioid.
data FareyAngle = FareyAngle InternalAngle BinaryAnglePair
  deriving (Eq, Ord, Read, Show, Data, Typeable)

-- | Special Farey angles.
fareyZero, fareyOne, fareyHalf :: FareyAngle
fareyZero = FareyAngle   0   (BinaryAnglePair (BinaryAngle [] [False]       ) (↙
    ↳ BinaryAngle [] [False]       ))
fareyOne  = FareyAngle   1   (BinaryAnglePair (BinaryAngle [] [True ]       ) (↙
    ↳ BinaryAngle [] [True ]       ))
fareyHalf = FareyAngle (1/2) (BinaryAnglePair (BinaryAngle [] [False, True ]) (↙
    ↳ BinaryAngle [] [True , False]))

-- | Farey addition of neighbours finding inner neighbour.
--   Precondition: a < b && a `neighbours` b && (a, b) /= (0, 1)
fareyAddition :: FareyAngle -> FareyAngle -> FareyAngle
fareyAddition
    (FareyAngle (InternalAngle a) (BinaryAnglePair (BinaryAngle [] al) (↙
        ↳ BinaryAngle [] ah)))
    (FareyAngle (InternalAngle b) (BinaryAnglePair (BinaryAngle [] bl) (↙
        ↳ BinaryAngle [] bh))) =
     FareyAngle (InternalAngle c) (BinaryAnglePair (BinaryAngle [] cl) (↙
         ↳ BinaryAngle [] ch))
  where
    c = (numerator a + numerator b) % (denominator a + denominator b)
    cl = p ++ l
    ch = p ++ h
    (p, l, h)
      | denominator a < denominator b = (ah, bl, bh)
      | denominator a > denominator b = (bl, al, ah)
     -- deliberately unhandled
-- deliberately unhandled
```

```
-- | Farey addition of neighbours finding inner neighbour.
--   Precondition: a `neighbours` b
(<+>) :: FareyAngle -> FareyAngle -> FareyAngle
-- special treatment for special cases
FareyAngle 0 _ <+> FareyAngle 0 _ = fareyZero
FareyAngle 1 _ <+> FareyAngle 1 _ = fareyOne
FareyAngle 0 _ <+> FareyAngle 1 _ = fareyHalf
FareyAngle 1 _ <+> FareyAngle 0 _ = fareyHalf
x@(FareyAngle a _) <+> y@(FareyAngle b _)
  | a < b = fareyAddition x y
  | b < a = fareyAddition y x
  -- deliberately unhandeled
infixl 6 <+>

-- | Farey search for parents.
--   Precondition: l < t && t < r && l `neighbours` r
fareySearch' :: FareyAngle -> FareyAngle -> InternalAngle -> (FareyAngle, ↙
    ↳ FareyAngle)
fareySearch' l r t
  | t <  s = fareySearch' l m t
  | t == s = (l, r)
  | t >  s = fareySearch' m r t
  where m@(FareyAngle s _) = l <+> r

-- | Farey search for angle.
fareySearch :: FareyAngle -> FareyAngle -> InternalAngle -> FareyAngle
fareySearch l r t = let (a, b) = fareySearch' l r t in a <+> b

-- | Find a Farey angle for an internal angle.
fareyAngle :: InternalAngle -> FareyAngle
fareyAngle t | 0 < t && t < 1 = fareySearch fareyZero fareyOne t
-- deliberately unhandled

-- | Find the Farey parent angles for an internal angle.
fareyParents :: InternalAngle -> (FareyAngle, FareyAngle)
fareyParents t | 0 < t && t < 1 = fareySearch' fareyZero fareyOne t
-- deliberately unhandled

-- | Find a Farey angle for an internal angle p/q.
(</>) :: Integer -> Integer -> FareyAngle
p </> q = fareyAngle (InternalAngle (p % q))
infixl 7 </>
```

18 Fractal/Mandelbrot/Symbolic/KneadingSequence.hs

```
{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving #-}
{- |
Module      :  Fractal.Mandelbrot.KneadingSequence
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Implementation of ideas from Dierk Schleicher's paper
/Internal Addresses Of The Mandelbrot Set And Galois Groups Of Polynomials (↙
```

```
    ↳ version of February 5, 2008)/
<http://arxiv.org/abs/math/9411238v2>.

-}
module Fractal.Mandelbrot.Symbolic.KneadingSequence
  ( Knead(..)
  , Kneading(..)
  , kneading
  , kneadingPeriod
  , unwrapKneading
  ) where

import Data.Data (Data())
import Data.Typeable (Typeable())
import Data.List (genericLength, genericSplitAt, genericTake)
import Data.Maybe (isJust, listToMaybe)

import Fractal.Mandelbrot.Symbolic.ExternalAngle (doubleAngle, doublingPreimages↙
    ↳ , ExternalAngle, Period(Period), wrapAngle)
import Fractal.Mandelbrot.Utils (strictlyWithin, strictlyWithout)

-- | Elements of kneading sequences.
data Knead
  = Zero
  | One
  | Star
  deriving (Read, Show, Eq, Ord, Enum, Bounded, Data, Typeable)

-- | Kneading sequences.  Note that the 'Aperiodic' case has an infinite list.
data Kneading
  = Aperiodic [Knead]
  | PrePeriodic [Knead] [Knead]
  | StarPeriodic [Knead]
  | Periodic  [Knead]
  deriving (Read, Show, Eq, Ord, Data, Typeable)

-- | The kneading sequence for an external angle.
kneading :: ExternalAngle -> Kneading
kneading a0'
  | a0 == 0 = StarPeriodic [Star]
  | otherwise = fst kneads
  where
    a0 = wrapAngle a0'
    lh = doublingPreimages a0
    kneads = kneading' 1 (doubleAngle a0)
    ks = (a0, One) : snd kneads
    kneading' :: Integer -> ExternalAngle -> (Kneading, [(ExternalAngle, Knead)↙
        ↳ ])
    kneading' n a
      | isJust i = case i of
          Just 0 -> case last qs of
            Star -> (StarPeriodic qs, [])
            _    -> (Periodic qs, [])
          Just j -> let (p, q) = genericSplitAt j qs
                    in (PrePeriodic p q, [])
          -- unreachable
      | a `strictlyWithin` lh = ((a, One ):) `mapP` k
```

```
      | a `strictlyWithout` lh = ((a, Zero):) `mapP` k
      | otherwise              = ((a, Star):) `mapP` k
      where
        k = kneading' (n+1) (doubleAngle a)
        ps = genericTake n ks
        qs = map snd ps
        i = fmap fst . listToMaybe . filter ((a ==) . fst . snd) . zip [(0 :: ↙
            ↳ Integer) ..] $ ps
        mapP f ~(x, y) = (x, f y)

-- | The period of a kneading sequence, or 'Nothing' when it isn't periodic.
kneadingPeriod :: Kneading -> Maybe Period
kneadingPeriod (StarPeriodic k) = Just (Period $ genericLength k)
kneadingPeriod (Periodic k) = Just (Period $ genericLength k)
kneadingPeriod _ = Nothing

-- | Unwrap a kneading sequence to an infinite list.
unwrapKneading :: Kneading -> [Knead]
unwrapKneading (Aperiodic vs) = vs
unwrapKneading (PrePeriodic us vs) = us ++ cycle vs
unwrapKneading (StarPeriodic vs) = cycle vs
unwrapKneading (Periodic vs) = cycle vs
```

19 Fractal/Mandelbrot/Symbolic/Text.hs

```
{- |
Module      :  Fractal.Mandelbrot.Symbolic.Text
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Human readable formatting.

-}
module Fractal.Mandelbrot.Symbolic.Text
  ( prettyAddress
  , parseAddress
  , prettyConciseAddress
  , parseConciseAddress
  , prettyConciseItem
  , parseConciseItem
  , prettyInternalAngle
  , parseInternalAngle
  , prettyExternalAngle
  , parseExternalAngle
  , prettyPeriod
  , parsePeriod
  , prettyInteger
  , parseInteger
  , prettyRational
  , parseRational
  ) where

import Control.Monad (liftM2)
```

```
import Data.Ratio ((%), numerator, denominator)

import Fractal.Mandelbrot.Symbolic.ExternalAngle (ExternalAngle(ExternalAngle), ↙
    ↳ Period(Period))
import Fractal.Mandelbrot.Symbolic.InternalAngle (InternalAngle(InternalAngle))
import Fractal.Mandelbrot.Symbolic.AngledInternalAddress (AngledInternalAddress↙
    ↳ ())
import Fractal.Mandelbrot.Symbolic.ConciseAddress(ConciseAddress(ConciseAddress)↙
    ↳ , ConciseItem(ConciseAngle, ConcisePeriod), toConciseAddress, ↙
    ↳ fromConciseAddress)

prettyInteger :: Integer -> String
prettyInteger i = show i

prettyRational :: Rational -> String
prettyRational r = show (numerator r) ++ "/" ++ show (denominator r)

prettyPeriod :: Period -> String
prettyPeriod (Period p) = prettyInteger p

prettyInternalAngle :: InternalAngle -> String
prettyInternalAngle (InternalAngle r) = prettyRational r

prettyExternalAngle :: ExternalAngle -> String
prettyExternalAngle (ExternalAngle r) = prettyRational r

prettyConciseItem :: ConciseItem -> String
prettyConciseItem (ConcisePeriod p) = prettyPeriod p
prettyConciseItem (ConciseAngle  r) = prettyInternalAngle r

prettyConciseAddress :: ConciseAddress -> String
prettyConciseAddress (ConciseAddress a) = unwords (map prettyConciseItem a)

prettyAddress :: AngledInternalAddress -> String
prettyAddress = prettyConciseAddress . toConciseAddress

parseAddress :: String -> Maybe AngledInternalAddress
parseAddress = fmap fromConciseAddress . parseConciseAddress

parseConciseAddress :: String -> Maybe ConciseAddress
parseConciseAddress s = ConciseAddress `fmap` mapM parseConciseItem (words s)

parseConciseItem :: String -> Maybe ConciseItem
parseConciseItem s
 | '/' `elem` s = ConciseAngle  `fmap` parseInternalAngle s
 | otherwise    = ConcisePeriod `fmap` parsePeriod s

parseInternalAngle :: String -> Maybe InternalAngle
parseInternalAngle s = InternalAngle `fmap` parseRational s

parseExternalAngle :: String -> Maybe ExternalAngle
parseExternalAngle s = ExternalAngle `fmap` parseRational s

parsePeriod :: String -> Maybe Period
parsePeriod s = Period `fmap` parseInteger s

parseInteger :: String -> Maybe Integer
```

```
parseInteger s = case reads s of
  [(i,"")] -> Just i
  _ -> Nothing

parseRational :: String -> Maybe Rational
parseRational s = case break ('/' ==) s of
  (n,'/':d) -> liftM2 (\n' d' -> (n' % d')) (parseInteger n) (parseInteger d)
  _ -> Nothing
```

20 Fractal/Mandelbrot/Utils.hs

```
{- |
Module      :  Fractal.Mandelbrot.Utils
Copyright   :  (c) Claude Heiland-Allen 2010-2012
License     :  BSD3

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

Miscellaneous utility functions.

-}
module Fractal.Mandelbrot.Utils
  ( fromBits
  , mod_
  , divisors
  , strictlyWithin
  , strictlyWithout
  , chunkWith2
  , genericElemIndex
  , magnitudeSquared
  , sqr
  , toComplexDouble
  , fromComplexDouble
  , recodeComplex
  , recodeFloat
  , scaleComplex
  , (-@?)
{-
  , safeGenericIndex
-}
  ) where

import Data.List (foldl')
import Data.Complex (Complex((:+)), magnitude)

magnitudeSquared :: Num r => Complex r -> r
magnitudeSquared (r:+i) = r*r + i*i

sqr :: Num r => Complex r -> Complex r
sqr (r:+i) = (r*r - i*i) :+ (2*r*i)

toComplexDouble :: RealFloat r => Complex r -> Complex Double
toComplexDouble = recodeComplex

fromComplexDouble :: RealFloat r => Complex Double -> Complex r
```

```
fromComplexDouble = recodeComplex

recodeComplex :: (RealFloat r, RealFloat t) => Complex r -> Complex t
recodeComplex (r:+i) = recodeFloat r :+ recodeFloat i

recodeFloat :: (RealFloat r, RealFloat t) => r -> t
recodeFloat = uncurry encodeFloat . decodeFloat

scaleComplex :: RealFloat r => Int -> Complex r -> Complex r
scaleComplex n (r:+i) = scaleFloat n r :+ scaleFloat n i

-- | Convert a big endian list of bits to an integer.
fromBits :: [Bool] -> Integer
fromBits = foldl' (\ a b -> 2 * a + if b then 1 else 0) 0

-- | Like @n `mod` d@ but with output in @[1..d]@.
mod_ :: (Eq a, Integral a) => a -> a -> a
n `mod_` d
  | m == 0 = d
  | otherwise = m
  where m = n `mod` d
infixl 7 `mod_`

-- | All divisors of a number.
divisors :: (Eq a, Integral a) => a -> [a]
divisors n = [ m | m <- [1 .. n], n `mod` m == 0 ]

-- | Inside the range?
strictlyWithin  :: Ord a => a -> (a, a) -> Bool
strictlyWithin  x (l, h) = l < x && x < h

-- | Outside the range?
strictlyWithout :: Ord a => a -> (a, a) -> Bool
strictlyWithout x (l, h) = x < l || h < x

-- | Split a list 2 elements at a time and combine.
chunkWith2 :: (a -> a -> b) -> [a] -> [b]
chunkWith2 _ [] = []
chunkWith2 f (x:y:zs) = f x y : chunkWith2 f zs
-- deliberately unhandled

-- | Generic version of 'elemIndex'.
genericElemIndex :: (Eq a, Integral b) => a -> [a] -> Maybe b
genericElemIndex _ [] = Nothing
genericElemIndex e (f:fs)
  | e == f = Just 0
  | otherwise = (1 +) `fmap` genericElemIndex e fs

-- | A measure of meaningful precision in the difference of two
--   finite non-zero values.
--
--   Values of very different magnitude have little meaningful
--   difference, because @a + b `approxEq` a@ when @|a| >> |b|@.
--
--   Very close values have little meaningful difference,
--   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
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)
    d = (x `max` y) - z
    e = abs (x - y) `min` p
    p = floatDigits a
    x = exponent a
    y = exponent b
    z = exponent c
    a = n i
    b = n j
    c = n (i - j)

-- | Much like 'effectivePrecisionWith' combined with 'normInfinity'.
effectivePrecision :: (Num t, RealFloat t) => Complex t -> Complex t -> Int
effectivePrecision = effectivePrecisionWith magnitude
infix 6 `effectivePrecision`


-- | An alias for 'effectivePrecision'.
(-@?) :: (Num t, RealFloat t) => Complex t -> Complex t -> Int
(-@?) = effectivePrecision
infix 6 -@?

{-
-- | Safe version of 'genericIndex'.
safeGenericIndex :: Integral b => [a] -> b -> Maybe a
safeGenericIndex [] _ = Nothing
safeGenericIndex (x:xs) i
  | i < 0 = Nothing
  | i > 0 = safeGenericIndex xs (i - 1)
  | otherwise = Just x
-}
```

21 .gitignore

```
dist
-
*.hi
*.o
```

22 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 Claude Heiland-Allen 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.
```

23 ruff.cabal

```
Name:                ruff
Version:             1.0
Synopsis:            relatively useful fractal functions
Description:
    A library for analysis and exploration of fractals, providing
    angled internal addresses, external ray tracing, nucleus and
    bond point finding algorithms for the Mandelbrot Set.

Homepage:            http://code.mathr.co.uk/ruff
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:  TODO

Library
  Exposed-modules:
    Fractal.Mandelbrot
    Fractal.Mandelbrot.Symbolic
    Fractal.Mandelbrot.Symbolic.AngledInternalAddress
    Fractal.Mandelbrot.Symbolic.Antenna
    Fractal.Mandelbrot.Symbolic.ConciseAddress
    Fractal.Mandelbrot.Symbolic.ExternalAngle
    Fractal.Mandelbrot.Symbolic.InternalAddress
    Fractal.Mandelbrot.Symbolic.InternalAngle
    Fractal.Mandelbrot.Symbolic.KneadingSequence
```

```
      Fractal.Mandelbrot.Symbolic.Text
      Fractal.Mandelbrot.Numeric
      Fractal.Mandelbrot.Numeric.Atom
      Fractal.Mandelbrot.Numeric.RayTraceForward
      Fractal.Mandelbrot.Numeric.RayTraceForward4
      Fractal.Mandelbrot.Numeric.RayTraceForward4Converge
      Fractal.Mandelbrot.Numeric.RayTraceReverse
      Fractal.Mandelbrot.Utils
    Build-depends:
      base >= 3 && < 6,
      rounded >= 0.1 && < 0.2
    GHC-Options:         -Wall -O2
    GHC-Prof-Options:    -prof -auto-all -caf-all

source-repository head
  type:     git
  location: http://code.mathr.co.uk/ruff.git

source-repository this
  type:     git
  location: http://code.mathr.co.uk/ruff.git
  tag:      v1.0
```

24 Setup.hs

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

25 TODO

```
rearrange module heirarchy to F.M.Symbolic and F.M.Numeric
split the Atom
refactor incremental algorithms into steppers and convergers
pretty printing to human-readable forms
parsing from human-readable forms
re-add missing references accidentally from the documentation
change  (b -> a) -> Maybe a  to  (Maybe b -> a) -> a
```