complex-generic

Claude Heiland-Allen

2012–2019

Contents

1 complex-generic.cabal

```
name:                complex-generic
version:             0.1.1.1
synopsis:            complex numbers with non-mandatory RealFloat
description:
  The base package's 'Data.Complex' has a 'RealFloat' requirement for
  almost all operations, which rules out uses such as 'Complex Rational'
  or 'Complex Integer'.  This package provides an alternative, putting
  most operations into additional type classes.  Generating instances
  with template haskell helps reduce excessive boilerplate and avoids
  instance overlap.

homepage:            https://code.mathr.co.uk/complex-generic
license:             BSD3
license-file:        LICENSE
author:              Claude Heiland-Allen
maintainer:          claude@mathr.co.uk
category:            Math
build-type:          Simple
cabal-version:       >=1.8

library
  exposed-modules:
    Data.Complex.Generic,
    Data.Complex.Generic.Default,
    Data.Complex.Generic.TH,
    Data.Complex.Generic.Class
  build-depends:
    base < 4.14,
    template-haskell < 2.16

source-repository head
  type:     git
```

```
  location: https://code.mathr.co.uk/complex-generic.git

source-repository this
  type:     git
  location: https://code.mathr.co.uk/complex-generic.git
  tag:      v0.1.1.1
```

2 Data/Complex/Generic/Class.hs

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
{- |
Module      :  Data.Complex.Generic.Class
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  MultiParamTypeClasses, FunctionalDependencies

Classes for complex number operations.
-}
module Data.Complex.Generic.Class where

-- | Rectangular form.
class ComplexRect c r | c -> r where
  -- | Construction.
  mkRect :: r {- ^ real -} -> r {- ^ imaginary -} -> c
  -- | Construction with imagPart 0.
  real :: r {- ^ real -} -> c
  -- | Construction with realPart 0.
  imag :: r {- ^ imaginary -} -> c
  -- | Deconstruction.
  rect :: c -> (r, r)
  -- | Get the real part.
  realPart :: c -> r
  -- | Get the imaginary part.
  imagPart :: c -> r
  -- | Conjugation.
  conjugate :: c -> c
  -- | Squared magnitude.
  magnitudeSquared :: c -> r
  -- | Complex square.
  sqr :: c -> c
  -- | Real-complex multiplication.
  (.*) :: r -> c -> c
  infixl 7 .*
  -- | Complex-real multiplication.
  (*.) :: c -> r -> c
  infixl 7 *.

-- | Complex-real division.
(/.) :: (Fractional r, ComplexRect c r) => c -> r -> c
z /. a = z *. recip a
infixl 6 /.

-- | A synonym for 'mkRect'.
(.+) :: ComplexRect c r => r -> r -> c
```

```
(.+) = mkRect
infix 6 .+

-- | Polar form.
class ComplexPolar c r | c -> r where
  -- Construction.
  mkPolar :: r -> r -> c
  -- Construction with magnitude 1.
  cis :: r -> c
  -- Deconstruction.
  polar :: c -> (r, r)
  -- | Magnitude.
  magnitude :: c -> r
  -- | Phase in (-pi, pi].
  phase :: c -> r
```

3 Data/Complex/Generic/Default.hs

```
{- |
Module      :  Data.Complex.Generic.Default
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

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

Default implementations of complex number operations.
-}
{- heavily based on:
-- Module      :  Data.Complex
-- Copyright   :  (c) The University of Glasgow 2001
-- License     :  BSD-style (see the file libraries/base/LICENSE)
-- http://hackage.haskell.org/packages/archive/base/4.5.0.0/doc/html/src/Data-
   ↳ Complex.html
--}

module Data.Complex.Generic.Default where

import Data.Complex.Generic.Class

realDefault :: (Num r, ComplexRect c r) => r -> c
realDefault r = r .+ 0

imagDefault :: (Num r, ComplexRect c r) => r -> c
imagDefault i = 0 .+ i

rectDefault :: ComplexRect c r => c -> (r, r)
rectDefault c = (realPart c, imagPart c)

realPartDefault :: ComplexRect c r => c -> r
realPartDefault = fst . rect

imagPartDefault :: ComplexRect c r => c -> r
imagPartDefault = snd . rect

conjugateDefault :: (Num r, ComplexRect c r) => c -> c
```

```
conjugateDefault c =
  let (x, y) = rect c
  in  x .+ negate y

magnitudeSquaredDefault :: (Num r, ComplexRect c r) => c -> r
magnitudeSquaredDefault c =
  let (x, y) = rect c
  in  x * x + y * y

sqrDefault :: (Num r, ComplexRect c r) => c -> c
sqrDefault z =
  let (x, y) = rect z
      xy = x * y
  in  (x + y) * (x - y) .+ (xy + xy)

sqrDefaultRF :: (RealFloat r, ComplexRect c r) => c -> c
sqrDefaultRF z =
  let (x, y) = rect z
  in  (x + y) * (x - y) .+ scaleFloat 1 (x * y)  -- FIXME assumes binary

rmulDefault :: (Num r, ComplexRect c r) => r -> c -> c
rmulDefault a z =
  let (x, y) = rect z
  in  (a * x) .+ (a * y)

mulrDefault :: (Num r, ComplexRect c r) => c -> r -> c
mulrDefault z a =
  let (x, y) = rect z
  in  (x * a) .+ (y * a)

mkPolarDefault :: (Floating r, ComplexRect c r) => r -> r -> c
mkPolarDefault r theta = r * cos theta .+ r * sin theta

cisDefault :: (Floating r, ComplexRect c r) => r -> c
cisDefault theta = cos theta .+ sin theta

polarDefault :: (ComplexPolar c r) => c -> (r, r)
polarDefault c = (magnitude c, phase c)

magnitudeDefault :: (Floating r, ComplexRect c r) => c -> r
magnitudeDefault = sqrt . magnitudeSquared

magnitudeDefaultRF :: (RealFloat r, ComplexRect c r) => c -> r
magnitudeDefaultRF w =
  let (x, y) = rect w
      k = max (exponent x) (exponent y)
      mk = - k
      sqr z = z * z
  in  scaleFloat k (sqrt (sqr (scaleFloat mk x) + sqr (scaleFloat mk y)))

phaseDefault :: (Ord r, Floating r, ComplexRect c r) => c -> r
phaseDefault c
  | x > 0             =  atan (y/x)
  | x == 0 && y > 0   =  pi/2
  | x <  0 && y > 0   =  pi + atan (y/x)
  | x <= 0 && y < 0   = -phaseDefault (conjugate c)
  | y == 0 && x < 0   =  pi    -- must be after the previous test on zero y
```

```
    | x==0 && y==0     =  y     -- must be after the other double zero tests
    | otherwise        =  x + y -- x or y is a NaN, return a NaN (via +)
    where
      x = realPart c
      y = imagPart c

phaseDefaultRF :: (RealFloat r, ComplexRect c r) => c -> r
phaseDefaultRF c = case (realPart c, imagPart c) of
  (0, 0) -> 0
  (x, y) -> atan2 y x

addDefault :: (Num r, ComplexRect c r) => c -> c -> c
addDefault z w =
  let (x,y) = rect z
      (x',y') = rect w
  in  (x+x') .+ (y+y')

subDefault :: (Num r, ComplexRect c r) => c -> c -> c
subDefault z w =
  let (x,y) = rect z
      (x',y') = rect w
  in  (x-x') .+ (y-y')

mulDefault :: (Num r, ComplexRect c r) => c -> c -> c
mulDefault z w =
  let (x,y) = rect z
      (x',y') = rect w
  in  (x*x'-y*y') .+ (x*y'+y*x')

negateDefault :: (Num r, ComplexRect c r) => c -> c
negateDefault z =
  let (x,y) = rect z
  in  negate x .+ negate y

absDefault :: (Num r, ComplexRect c r, ComplexPolar c r) => c -> c
absDefault = real . magnitude

signumDefault :: (Eq r, Fractional r, ComplexRect c r, ComplexPolar c r) => c ->↙
    ↳ c
signumDefault z = case rect z of
  (0, 0) -> 0 .+ 0
  (x, y) -> x/r .+ y/r
  where r = magnitude z

fromIntegerDefault :: (Num r, ComplexRect c r) => Integer -> c
fromIntegerDefault = real . fromInteger

divDefault :: (Fractional r, ComplexRect c r) => c -> c -> c
divDefault z w =
  let (x,y) = rect z
      (x',y') = rect w
      d = x'*x' + y'*y'
  in  (x*x'+y*y') / d .+ (y*x'-x*y') / d

divDefaultRF :: (RealFloat r, ComplexRect c r) => c -> c -> c
divDefaultRF z w =
  let (x,y) = rect z
```

```
        (x',y') = rect w
        x'' = scaleFloat k x'
        y'' = scaleFloat k y'
        k = max (exponent x') (exponent y')
        d = x'*x'' + y'*y''
    in  (x*x''+y*y'') / d :+ (y*x''-x*y'') / d

fromRationalDefault :: (Fractional r, ComplexRect c r) => Rational -> c
fromRationalDefault = real . fromRational

piDefault :: (Floating r, ComplexRect c r) => c
piDefault = real pi

expDefault :: (Floating r, ComplexRect c r) => c -> c
expDefault z =
  let (x, y) = rect z
      expx = exp x
  in  expx * cos y :+ expx * sin y

logDefault :: (Floating r, ComplexRect c r, ComplexPolar c r) => c -> c
logDefault z = log (magnitude z) :+ phase z

sqrtDefault :: (Eq r, Ord r, Floating r, ComplexRect c r, ComplexPolar c r) => c
    -> c
sqrtDefault z = case rect z of
  (0, 0) -> 0 :+ 0
  (x, y) ->
    let (u,v) = if x < 0 then (v',u') else (u',v')
        v'    = abs y / (u'*2)
        u'    = sqrt ((magnitude z + abs x) / 2)
    in  u :+ (if y < 0 then -v else v)

sinDefault :: (Floating r, ComplexRect c r) => c -> c
sinDefault z =
  let (x, y) = rect z
  in  sin x * cosh y :+ cos x * sinh y

cosDefault :: (Floating r, ComplexRect c r) => c -> c
cosDefault z =
  let (x, y) = rect z
  in  cos x * cosh y :+ (- sin x * sinh y)

tanDefault :: (Floating r, Fractional c, ComplexRect c r) => c -> c
tanDefault z =
  let (x, y) = rect z
      sinx  = sin x
      cosx  = cos x
      sinhy = sinh y
      coshy = cosh y
  in  (sinx*coshy.+cosx*sinhy)/(cosx*coshy.+(-sinx*sinhy))

sinhDefault :: (Floating r, ComplexRect c r) => c -> c
sinhDefault z =
  let (x, y) = rect z
  in  cos y * sinh x :+ sin  y * cosh x

coshDefault :: (Floating r, ComplexRect c r) => c -> c
```

```
coshDefault z =
  let (x, y) = rect z
  in  cos y * cosh x .+ sin y * sinh x

tanhDefault :: (Floating r, Floating c, ComplexRect c r) => c -> c
tanhDefault z =
  let (x, y) = rect z
      siny  = sin y
      cosy  = cos y
      sinhx = sinh x
      coshx = cosh x
  in  (cosy*sinhx.+siny*coshx)/(cosy*coshx.+siny*sinhx)

asinDefault :: (Num r, Floating c, ComplexRect c r) => c -> c
asinDefault z =
  let (x, y) = rect z
      (x', y') = rect $ log (((-y).+x) + sqrt (1 - z*z))
  in  y'.+(-x')

acosDefault :: (Num r, Floating c, ComplexRect c r) => c -> c
acosDefault z =
  let (x'',y'') = rect $ log (z + ((-y').+x'))
      (x',y')   = rect $ sqrt (1 - z*z)
  in  y''.+(-x'')

atanDefault :: (Num r, Floating c, ComplexRect c r) => c -> c
atanDefault z =
  let (x, y) = rect z
      (x',y') = rect $ log (((1-y).+x) / sqrt (1+z*z))
  in  y'.+(-x')

asinhDefault :: (Floating c, ComplexRect c r) => c -> c
asinhDefault z = log (z + sqrt (1+z*z))

acoshDefault :: (Floating c, ComplexRect c r) => c -> c
acoshDefault z = log (z + (z+1) * sqrt ((z-1)/(z+1)))

atanhDefault :: (Floating c, ComplexRect c r) => c -> c
atanhDefault z =  0.5 * log ((1.0+z) / (1.0-z))
```

4 Data/Complex/Generic.hs

```
{-# LANGUAGE DeriveDataTypeable, TemplateHaskell, MultiParamTypeClasses, ↙
    ↳ FlexibleInstances #-}
{- |
Module      :  Data.Complex.Generic
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  unstable
Portability :  DeriveDataTypeable, TemplateHaskell, MultiParamTypeClasses, ↙
    ↳ FlexibleInstances

Complex numbers.
-}
module Data.Complex.Generic
```

```
    ( module Data.Complex.Generic.Class
    , Complex((:+))
    , toDataComplex
    , fromDataComplex
    ) where

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

import Foreign.C (CFloat, CDouble)
import Data.Int
import Data.Word
import Data.Fixed
import Data.Ratio

import qualified Data.Complex as X

import Data.Complex.Generic.Class
import Data.Complex.Generic.Default
import Data.Complex.Generic.TH

-- | Complex numbers in rectangular form.
data Complex a = !a :+ !a
  deriving (Eq, Show, Read, Data, Typeable)
infix 6 :+

-- | Convert to 'Data.Complex.Complex'.
toDataComplex :: Complex r -> X.Complex r
toDataComplex (x :+ y) = x X.:+ y

-- | Convert from 'Data.Complex.Complex'.
fromDataComplex :: X.Complex r -> Complex r
fromDataComplex (x X.:+ y) = x :+ y

instance Functor Complex where
  fmap f (x :+ y) = f x :+ f y

mk :: a -> a -> Complex a
{- needed because of this bug(?) in TemplateHaskell
    Illegal variable name: ':+'
    When splicing a TH declaration
-}
mk = (:+)

toPair :: Complex a -> (a, a)
toPair (x :+ y) = (x, y)

deriveComplexRF ''Complex ''Float 'mk 'toPair
deriveComplexRF ''Complex ''Double 'mk 'toPair
deriveComplexRF ''Complex ''CFloat 'mk 'toPair
deriveComplexRF ''Complex ''CDouble 'mk 'toPair

deriveComplexN ''Complex ''Integer 'mk 'toPair
deriveComplexN ''Complex ''Int 'mk 'toPair
deriveComplexN ''Complex ''Int8 'mk 'toPair
deriveComplexN ''Complex ''Int16 'mk 'toPair
deriveComplexN ''Complex ''Int32 'mk 'toPair
```

```
deriveComplexN ''Complex ''Int64 'mk 'toPair
deriveComplexN ''Complex ''Word 'mk 'toPair
deriveComplexN ''Complex ''Word8 'mk 'toPair
deriveComplexN ''Complex ''Word16 'mk 'toPair
deriveComplexN ''Complex ''Word32 'mk 'toPair
deriveComplexN ''Complex ''Word64 'mk 'toPair
{-
deriveComplex1F ''Complex ''HasResolution ''Fixed 'mk 'toPair
deriveComplex1F ''Complex ''Integral ''Ratio 'mk 'toPair
-}

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

instance HasResolution t => Num (Complex (Fixed t)) where
  (+) = addDefault
  (-) = subDefault
  (*) = mulDefault
  negate = negateDefault
  fromInteger = fromIntegerDefault
  abs = error $ "Num.abs: not implementable for Complex Fixed"
  signum = error $ "Num.signum: not implementable for Complex Fixed"

instance HasResolution t => Fractional (Complex (Fixed t)) where
  (/) = divDefault
  fromRational = fromRationalDefault

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

instance Integral t => Num (Complex (Ratio t)) where
  (+) = addDefault
  (-) = subDefault
  (*) = mulDefault
  negate = negateDefault
  fromInteger = fromIntegerDefault
  abs = error $ "Num.abs: not implementable for Complex Ratio"
```

```
  signum = error $ "Num.signum: not implementable for Complex Ratio"

instance Integral t => Fractional (Complex (Ratio t)) where
  (/) = divDefault
  fromRational = fromRationalDefault
```

5 Data/Complex/Generic/TH.hs

```
{-# LANGUAGE TemplateHaskell, MultiParamTypeClasses, FlexibleInstances, ↙
    ↳ UndecidableInstances #-}
{- |
Module      : Data.Complex.Generic.TH
Copyright   : (c) Claude Heiland-Allen 2012,2017
License     : BSD3

Maintainer  : claude@mathr.co.uk
Stability   : unstable
Portability : TemplateHaskell, MultiParamTypeClasses, FlexibleInstances, ↙
    ↳ UndecidableInstances

Derive instances for complex numbers using template haskell.
-}
module Data.Complex.Generic.TH where

import Data.Typeable (typeOf, typeOf1)
import Language.Haskell.TH

import Data.Complex.Generic.Class
import Data.Complex.Generic.Default

-- | Derive instances for 'RealFloat' types.
deriveComplexRF :: Name {- ^ complex type -} -> Name {- ^ real type -} -> Name ↙
    ↳ {- ^ constructor -} -> Name {- ^ destructor -} -> Q [Dec]
deriveComplexRF cTy' rTy' mkRectI' rectI' = [d|
    instance ComplexRect ($(cTy) $(rTy)) $(rTy) where
      mkRect = $(mkRectI)
      rect = $(rectI)
      real = realDefault
      imag = imagDefault
      realPart = realPartDefault
      imagPart = imagPartDefault
      conjugate = conjugateDefault
      magnitudeSquared = magnitudeSquaredDefault
      sqr = sqrDefault
      (.*) = rmulDefault
      (*.) = mulrDefault

    instance ComplexPolar ($(cTy) $(rTy)) $(rTy) where
      mkPolar = mkPolarDefault
      cis = cisDefault
      polar = polarDefault
      magnitude = magnitudeDefaultRF
      phase = phaseDefaultRF

    instance Num ($(cTy) $(rTy)) where
      (+) = addDefault
      (-) = subDefault
```

```
        (*) = mulDefault
        negate = negateDefault
        fromInteger = fromIntegerDefault
        abs = absDefault
        signum = signumDefault

      instance Fractional ($(cTy) $(rTy)) where
        (/) = divDefaultRF
        fromRational = fromRationalDefault

      instance Floating ($(cTy) $(rTy)) where
        pi = piDefault
        exp = expDefault
        log = logDefault
        sqrt = sqrtDefault
        sin = sinDefault
        cos = cosDefault
        tan = tanDefault
        sinh = sinhDefault
        cosh = coshDefault
        tanh = tanhDefault
        asin = asinDefault
        acos = acosDefault
        atan = atanDefault
        asinh = asinhDefault
        acosh = acoshDefault
        atanh = atanhDefault
    |]
    where
      cTy = conT cTy'
      rTy = conT rTy'
      mkRectI = varE mkRectI'
      rectI = varE rectI'

-- | Derive instances for 'Floating' types.
deriveComplexF :: Name {- ^ complex type -} -> Name {- ^ real type -} -> Name {-↙
    ↪ ^ constructor -} -> Name {- ^ destructor -} -> Q [Dec]
deriveComplexF cTy' rTy' mkRectI' rectI' = [d|
    instance ComplexRect ($(cTy) $(rTy)) $(rTy) where
      mkRect = $(mkRectI)
      rect = $(rectI)
      real = realDefault
      imag = imagDefault
      realPart = realPartDefault
      imagPart = imagPartDefault
      conjugate = conjugateDefault
      magnitudeSquared = magnitudeSquaredDefault
      sqr = sqrDefault
      (.*) = rmulDefault
      (*.) = mulrDefault

    instance ComplexPolar ($(cTy) $(rTy)) $(rTy) where
      mkPolar = mkPolarDefault
      cis = cisDefault
      polar = polarDefault
      magnitude = magnitudeDefault
      phase = phaseDefault
```

```
      instance Num ($(cTy) $(rTy)) where
        (+) = addDefault
        (-) = subDefault
        (*) = mulDefault
        negate = negateDefault
        fromInteger = fromIntegerDefault
        abs = absDefault
        signum = signumDefault

      instance Fractional ($(cTy) $(rTy)) where
        (/) = divDefault
        fromRational = fromRationalDefault

      instance Floating ($(cTy) $(rTy)) where
        pi = piDefault
        exp = expDefault
        log = logDefault
        sqrt = sqrtDefault
        sin = sinDefault
        cos = cosDefault
        tan = tanDefault
        sinh = sinhDefault
        cosh = coshDefault
        tanh = tanhDefault
        asin = asinDefault
        acos = acosDefault
        atan = atanDefault
        asinh = asinhDefault
        acosh = acoshDefault
        atanh = atanhDefault
    |]
    where
      cTy = conT cTy'
      rTy = conT rTy'
      mkRectI = varE mkRectI'
      rectI = varE rectI'

-- | Derive instances for 'Num' types.
deriveComplexN :: Name {- ^ complex type -} -> Name {- ^ real type -} -> Name {-↙
    ↳ ^ constructor -} -> Name {- ^ destructor -} -> Q [Dec]
deriveComplexN cTy' rTy' mkRectI' rectI' = [d|
    instance ComplexRect ($(cTy) $(rTy)) $(rTy) where
      mkRect = $(mkRectI)
      rect = $(rectI)
      real = realDefault
      imag = imagDefault
      realPart = realPartDefault
      imagPart = imagPartDefault
      conjugate = conjugateDefault
      magnitudeSquared = magnitudeSquaredDefault
      sqr = sqrDefault
      (.*) = rmulDefault
      (*.) = mulrDefault

    instance Num ($(cTy) $(rTy)) where
      (+) = addDefault
```

```
        (-) = subDefault
        (*) = mulDefault
        negate = negateDefault
        fromInteger = fromIntegerDefault
        abs = error $ "Num.abs: not implementable for " ++ show (typeOf (undefined ↙
            ↳ :: ($(cTy) $(rTy))))
        signum = error $ "Num.signum: not implementable for " ++ show (typeOf (↙
            ↳ undefined :: ($(cTy) $(rTy))))

    |]
    where
      cTy = conT cTy'
      rTy = conT rTy'
      mkRectI = varE mkRectI'
      rectI = varE rectI'

{-
-- | Derive instances for 'Fractional' types with one class constraint.
deriveComplex1F :: Name {- ^ complex type -} -> Name {- ^ constraint class -} ->↙
    ↳ Name {- ^ real type constructor -} -> Name {- ^ constructor -} -> Name {-↙
    ↳ ^ destructor -} -> Q [Dec]
deriveComplex1F cTy' sTy' rTy' mkRectI' rectI' = do
  t' <- newName "t"
  let t = varT t'
  c <- classP sTy' [t]
  is <- [d|
    instance ComplexRect ($(cTy) ($(rTy) $(t))) ($(rTy) $(t)) where
      mkRect = $(mkRectI)
      rect = $(rectI)
      real = realDefault
      imag = imagDefault
      realPart = realPartDefault
      imagPart = imagPartDefault
      conjugate = conjugateDefault
      magnitudeSquared = magnitudeSquaredDefault
      sqr = sqrDefault
      (.*) = rmulDefault
      (*.) = mulrDefault

    instance Num ($(cTy) ($(rTy) $(t))) where
      (+) = addDefault
      (-) = subDefault
      (*) = mulDefault
      negate = negateDefault
      fromInteger = fromIntegerDefault
      abs = error $ "Num.abs: not implementable for " ++ show (typeOf1 (↙
          ↳ undefined :: $(cTy) (($(rTy) $(t)))) ++ " " ++ show (typeOf1 (↙
          ↳ undefined :: $(rTy) $(t)))
      signum = error $ "Num.signum: not implementable for " ++ show (typeOf1 (↙
          ↳ undefined :: $(cTy) (($(rTy) $(t)))) ++ " " ++ show (typeOf1 (↙
          ↳ undefined :: $(rTy) $(t)))

    instance Fractional ($(cTy) ($(rTy) $(t))) where
      (/) = divDefault
      fromRational = fromRationalDefault
    |]
  return (map (\(InstanceD _ ty decs) -> InstanceD [c] ty decs) is)
```

```
  where
    cTy = conT cTy'
    sTy = conT sTy'
    rTy = conT rTy'
    mkRectI = varE mkRectI'
    rectI = varE rectI'
-}
```

6 .gitignore

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

7 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.
```

8 Setup.hs

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