Visualizing Decomposition

a Haskell project by David Banas
Last updated on February 1, 2014

Discrete Fourier Transform (DFT)

$$\{x_n\} \Leftrightarrow \{X_k\}$$

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn}, 0 \leq k < N$$

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{j\frac{2\pi}{N}nk}, 0 \leq n < N$$

DFT in Haskell

```haskell
-- Discrete Fourier Transform (DFT)
dft :: [Complex Double] -> [Complex Double]
dft xs = [ sum [ x * exp((0.0 :+ (-1.0)) * 2 * pi / lenXs * fromIntegral(k * n))
                   | (x, n) <- zip xs [0..]
                 ]
             | k <- [0..(length xs - 1)]
         ]
    where lenXs = fromIntegral $ length xs
```

O(DFT) ?

- $\{X_k\}$, $0 <= k < N$: N elements to compute, each requiring N multiply-accumulate (MAC) operations => $O(N^2)$.
- but, …

Divide & Conquer?

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j\frac{2\pi}{N}k\cdot 2n} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-j\frac{2\pi}{N}k(2n+1)}$$

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j\frac{2\pi}{N/2}kn} + e^{-j\frac{2\pi}{N}k} \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-j\frac{2\pi}{N/2}kn}$$

Yes!

$$X_k = DFT_{N/2}(y_n)[k'] + e^{-j\frac{2\pi}{N}k} DFT_{N/2}(z_n)[k']$$

$$k' = k\% \frac{N}{2}$$

$$\mathrm{y_n} = \mathrm{even}(\mathrm{x_n})$$

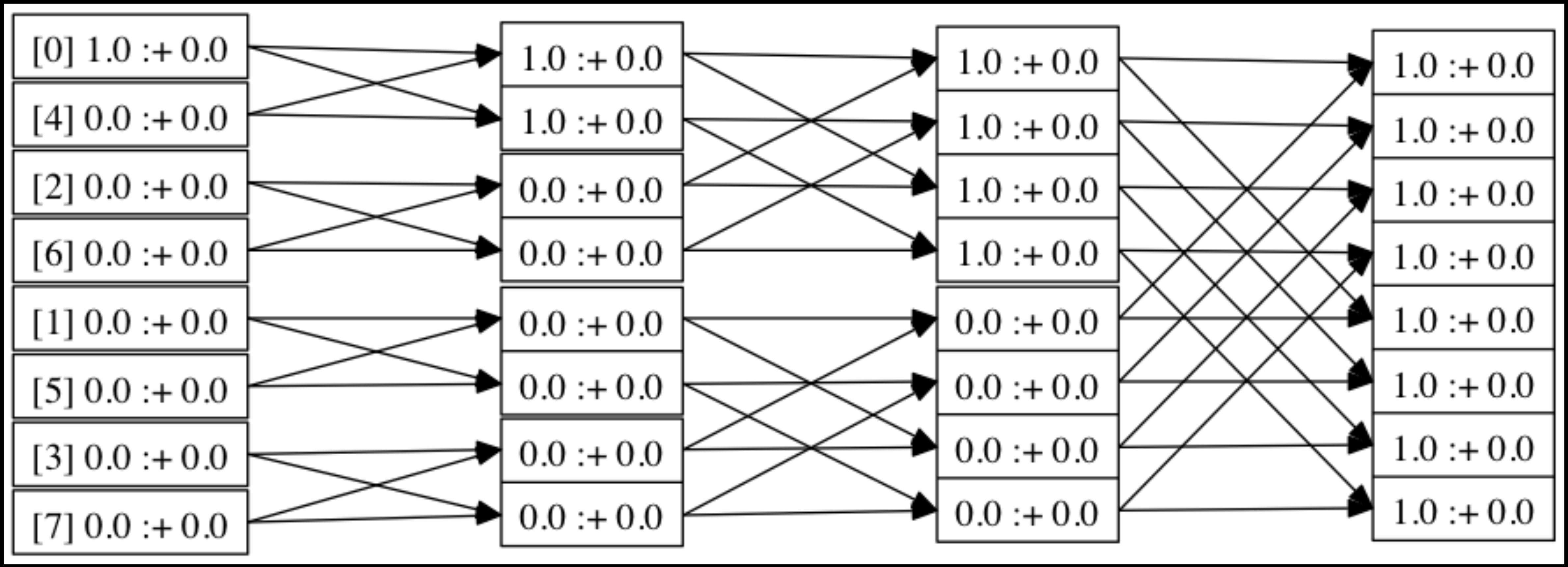
$$\mathrm{z_n} = \mathrm{odd}(\mathrm{x_n})$$

- Radix-2, decimation in time (DIT) decomposition.
- a.k.a. the “Fast Fourier Transform” (FFT).
- O(n * log n)

FFT in Haskell

```
radix2_DIT :: RealFloat a =>
              Bool -> [Complex a] -> [Complex a]
radix2_DIT _ []    = []
radix2_DIT _ [x]   = [x]
radix2_DIT rev xs  = (++) (zipWith (+) xes xos)
                          (zipWith (-) xes xos)
    where xes = radix2_DIT rev (evens xs)
          xos = zipWith (*)
                  (radix2_DIT rev (odds xs))
                  [ wn ** (fromIntegral k)
                    | k <- [0..]
                  ]
          wn | rev       = exp ( 0.0 :+
                                 ( 2.0 * pi /
                                   (fromIntegral (length xs))
                                  )
                                )
             | otherwise = exp ( 0.0 :+
                                 ( -2.0 * pi /
                                   (fromIntegral (length xs))
                                  )
                                )
```

Radix-2 DIT Tree



Other radices?

$$X_k = \sum_{n=0}^{\frac{N}{3}-1} x_{3n} e^{-j\frac{2\pi}{N}k\cdot 3n} + \sum_{n=0}^{\frac{N}{3}-1} x_{3n+1} e^{-j\frac{2\pi}{N}k(3n+1)} + \sum_{n=0}^{\frac{N}{3}-1} x_{3n+2} e^{-j\frac{2\pi}{N}k(3n+2)}$$

$$= \sum_{n=0}^{\frac{N}{3}-1} x_{3n} e^{-j\frac{2\pi}{N/3}kn} + e^{-j\frac{2\pi}{N}k} \sum_{n=0}^{\frac{N}{3}-1} x_{3n+1} e^{-j\frac{2\pi}{N/3}kn} + e^{-j\frac{2\pi}{N}2k} \sum_{n=0}^{\frac{N}{3}-1} x_{3n+2} e^{-j\frac{2\pi}{N/3}kn}$$

$$= e^{-j\frac{2\pi}{N}0k} DFT_{\frac{N}{3}}(\{x_0, x_3, \ldots\})[k'] + e^{-j\frac{2\pi}{N}1k} DFT_{\frac{N}{3}}(\{x_1, x_4, \ldots\})[k']$$

$$+ e^{-j\frac{2\pi}{N}2k} DFT_{\frac{N}{3}}(\{x_2, x_5, \ldots\})[k']$$

$$k' = k \% \tfrac{N}{3}$$

Generalized DIT FFT

$$X_k = \sum_{r=0}^{R-1} e^{-j\frac{2\pi}{N}kr} \cdot DFT_{\frac{N}{R}}(\{x_{mR+r}\})[k']$$

$$k' = k\% \frac{N}{R}, 0 \le m < \frac{N}{R}$$

$$X = \sum_{r=0}^{R-1} E_r^N \cdot replicate\left(R, DFT_{\frac{N}{R}}\left(\{x_{mR+r}\}\right)\right)$$

$$E_r^N = \left\{ e^{-j\frac{2\pi}{N}rn} \right\}, 0 \le n < N$$

Decimation in Frequency (DIF)

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn} = \sum_{n=0}^{\frac{N}{2}-1} x_n e^{-j\frac{2\pi}{N}kn} + \sum_{n=\frac{N}{2}}^{N-1} x_n e^{-j\frac{2\pi}{N}kn}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x_n e^{-j\frac{2\pi}{N}kn} + x_{n+\frac{N}{2}} e^{-j\frac{2\pi}{N}k\left(n+\frac{N}{2}\right)} = \sum_{n=0}^{\frac{N}{2}-1} \left(x_n + x_{n+\frac{N}{2}} e^{-j\pi k}\right) e^{-j\frac{2\pi}{N}kn}$$

$$= \left\{ \begin{array}{l} \sum_{n=0}^{\frac{N}{2}-1} \left(x_n + x_{n+\frac{N}{2}}\right) e^{-j\frac{2\pi}{N/2}k'n}, k = 2k' \\ \sum_{n=0}^{\frac{N}{2}-1} \left(x_n - x_{n+\frac{N}{2}}\right) e^{-j\frac{2\pi}{N}n} e^{-j\frac{2\pi}{N/2}k'n}, k = 2k'+1 \end{array} \right\}, 0 \le k' < \frac{N}{2}$$

Decimation in Frequency (DIF)

$$X_k = \left\{ \begin{matrix} DFT_{k'}(\{x_n + x_{n+\frac{N}{2}}\}), k = 2k' \\ DFT_{k'}(\{(x_n - x_{n+\frac{N}{2}})e^{-j\frac{2\pi}{N}n}\}), k = 2k'+1 \end{matrix} \right\}, 0 \le n, k' < \frac{N}{2}$$

In the DIF case, we interleave, rather than concatenate, the sub-transforms, in order to form the final output.

Other radices?

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn} = \sum_{n=\frac{0N}{3}}^{\frac{1N}{3}-1} x_n e^{-j\frac{2\pi}{N}kn} + \sum_{n=\frac{1N}{3}}^{\frac{2N}{3}-1} x_n e^{-j\frac{2\pi}{N}kn} + \sum_{n=\frac{2N}{3}}^{\frac{3N}{3}-1} x_n e^{-j\frac{2\pi}{N}kn}$$

$$= \sum_{r=0}^{R-1} \sum_{n=0}^{\frac{N}{R}-1} x_{n+r\frac{N}{R}} e^{-j\frac{2\pi}{N}k\left(n+r\frac{N}{R}\right)} = \sum_{r=0}^{R-1} e^{-j\frac{2\pi}{R}kr} \sum_{n=0}^{\frac{N}{R}-1} x_{n+r\frac{N}{R}} e^{-j\frac{2\pi}{N}kn}$$

Let k = Rk' + m; $0 \le k' < \frac{N}{R}, 0 \le m < R$...

Generalized DIF FFT

$$X_k = \sum_{r=0}^{R-1} e^{-j\frac{2\pi}{R}kr} \sum_{n=0}^{\frac{N}{R}-1} x_{n+r\frac{N}{R}} e^{-j\frac{2\pi}{N}(Rk'+m)n}$$

$$= \sum_{r=0}^{R-1} e^{-j\frac{2\pi}{R}kr} \sum_{n=0}^{\frac{N}{R}-1} \left(x_{n+r\frac{N}{R}} e^{-j\frac{2\pi}{N}mn} \right) e^{-j\frac{2\pi}{N/R}k'n}; m = k\%R$$

$$X = ?$$

$$k' = ?$$

$$m = ?$$

Generalized DIF FFT (cont'd.)

k	k' (R = 3)	m (R = 3)
0	0	0
1	0	1
2	0	2
3	1	0

Generalized DIF FFT (cont'd.)

$$X_k = \sum_{r=0}^{R-1} e^{-j\frac{2\pi}{R}kr} \sum_{n=0}^{\frac{N}{R}-1} \left(x_{n+r\frac{N}{R}} e^{-j\frac{2\pi}{N}mn} \right) e^{-j\frac{2\pi}{N/R}k'n}; m = k\%R$$

$$X = \sum_{r=0}^{R-1} E_r^R \cdot \text{interleave}\left(\{ DFT_{\frac{N}{R}}(x^r \cdot W_m^R) \mid 0 \le m < R \} \right)$$

$$x^r = \left\{ x_{r\frac{N}{R}+n} \right\}; 0 \le n < \frac{N}{R}$$

$$W_m^R = \left\{ e^{-j\frac{2\pi}{N}mn} \right\}; 0 \le n < \frac{N}{R}$$

Generalized FFT

DIT

$$X = \sum_{r=0}^{R-1} W_r^{1,1} \cdot \text{replicate}\left(R, DFT_{\frac{N}{R}}\left(\{x_{nR+r} \mid 0 \le n < \tfrac{N}{R}\}\right)\right)$$

$$= \sum_{r=0}^{R-1} W_r^{1,1} \cdot \text{concatenate}\left(\{DFT_{\frac{N}{R}}(\left\{x_{nR+r} \mid 0 \le n < \tfrac{N}{R}\right\} \cdot W_0^{1,R}) \mid 0 \le m < R\}\right)$$

DIF

$$X = \sum_{r=0}^{R-1} W_r^{N/R,1} \cdot \text{interleave}\left(\{DFT_{\frac{N}{R}}(\left\{x_{r\frac{N}{R}+n} \mid 0 \le n < \tfrac{N}{R}\right\} \cdot W_m^{1,R}) \mid 0 \le m < R\}\right)$$

$$W_k^{l,L} = \left\{e^{-j\frac{2\pi}{N}kln}\right\}; 0 \le n < \frac{N}{L}$$

Generalized FFT in Haskell

```
evalNode (Node (      _,  _, _, dif) children) =
    foldl (zipWith (+)) [0.0 | n <- [1..nodeLen]]
      $ zipWith (zipWith (*)) subTransforms phasors
  where subTransforms =
              [ subCombFunc
                  $ map evalNode
                        [ snd (coProd twiddle child)
                          | twiddle <- twiddles
                        ]
                | child <- children
              ]
        subCombFunc =
          if dif then concat . transpose -- i.e. - interleave
                 else concat             -- simple replication.
```

evalNode ?

Exploring the new code

```haskell
type GenericLogTree a = Tree (Maybe a, [Int], Int, Bool)

class (t ~ GenericLogTree a) => LogTree t a | t -> a where

    -- evalNode - Evaluates a node in a tree, returning a list of values
    --            of the original type.
    evalNode :: t -> [a]

-- FFTTree - an instance of LogTree, this type represents the Fast Fourier
--           Transform (FFT) of arbitrary radix and decimation scheme.
type FFTTree = GenericLogTree (Complex Double)
instance LogTree FFTTree (Complex Double) where
    evalNode … (See previous slide.)
```

Building a *FFTTree*

```haskell
data TreeData a = TreeData {
    modes  :: [(Int, Bool)]
  , values :: [a]
} deriving(Show)

newTreeData :: [(Int, Bool)] -- ^ Decomposition modes : (radix, DIF_flag).
            -> [a]           -- ^ Values for populating the tree.
            -> TreeData a    -- ^ Resultant data structure for passing to tree
constructor.
newTreeData modes values = TreeData {
                               modes  = modes
                             , values = values
                           }

newtype TreeBuilder t = TreeBuilder {
   buildTree :: LogTree t a => TreeData a -> Either String t
}

-- | Returns a tree builder suitable for constructing Fast Fourier Transform
--   (FFT) decomposition trees of arbitrary radices and either decimation
--   style (i.e. - DIT or DIF).
newFFTTree :: TreeBuilder FFTTree
newFFTTree = TreeBuilder buildMixedRadixTree
```

Building a *GenericLogTree*

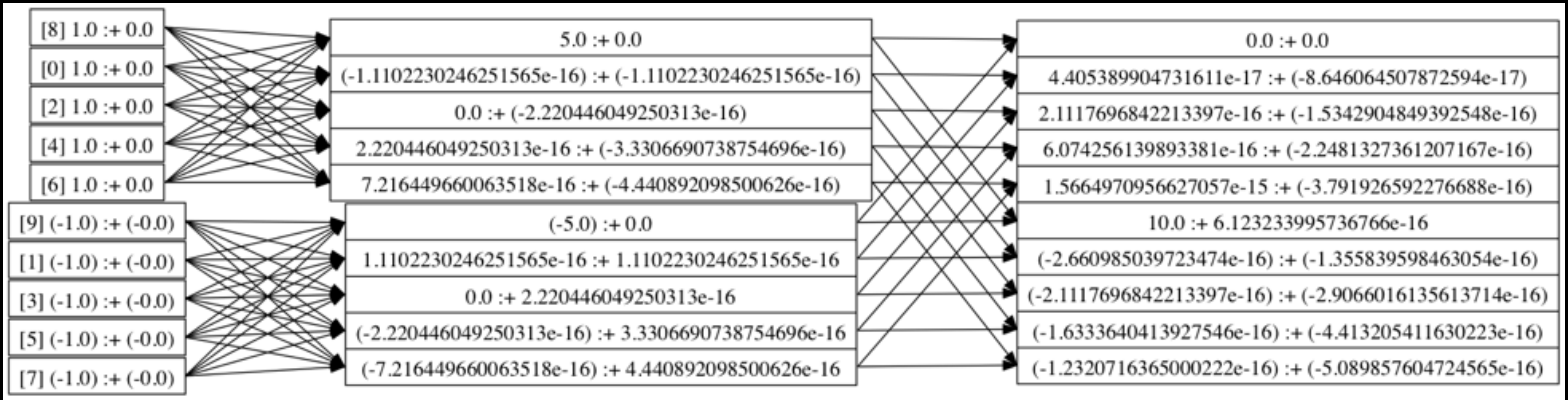
```
buildMixedRadixTree :: TreeData a -> Either String (GenericLogTree a)
buildMixedRadixTree td = mixedRadixTree td_modes td_values
    where td_modes  = modes td
          td_values = values td

mixedRadixTree :: [(Int, Bool)] -> [a] -> Either String (GenericLogTree a)
mixedRadixTree _     []  = Left "mixedRadixTree(): called with empty list."
mixedRadixTree _     [x] = return $ Node (Just x, [], 0, False) []
mixedRadixTree modes xs  = mixedRadixRecurse 0 1 modes xs


mixedRadixRecurse :: Int -> Int -> [(Int, Bool)] -> [a] -> Either String (GenericLogTree
a)
mixedRadixRecurse _ _ _ []  = Left "mixedRadixRecurse(): called with empty list."
mixedRadixRecurse myOffset _ _ [x] = return $ Node (Just x, [myOffset], 0, False) []
mixedRadixRecurse myOffset mySkipFactor modes xs
```

Example Generic FFT

```
tData6  = newTreeData [(2, False), (5, False)]
                      [1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0]
```

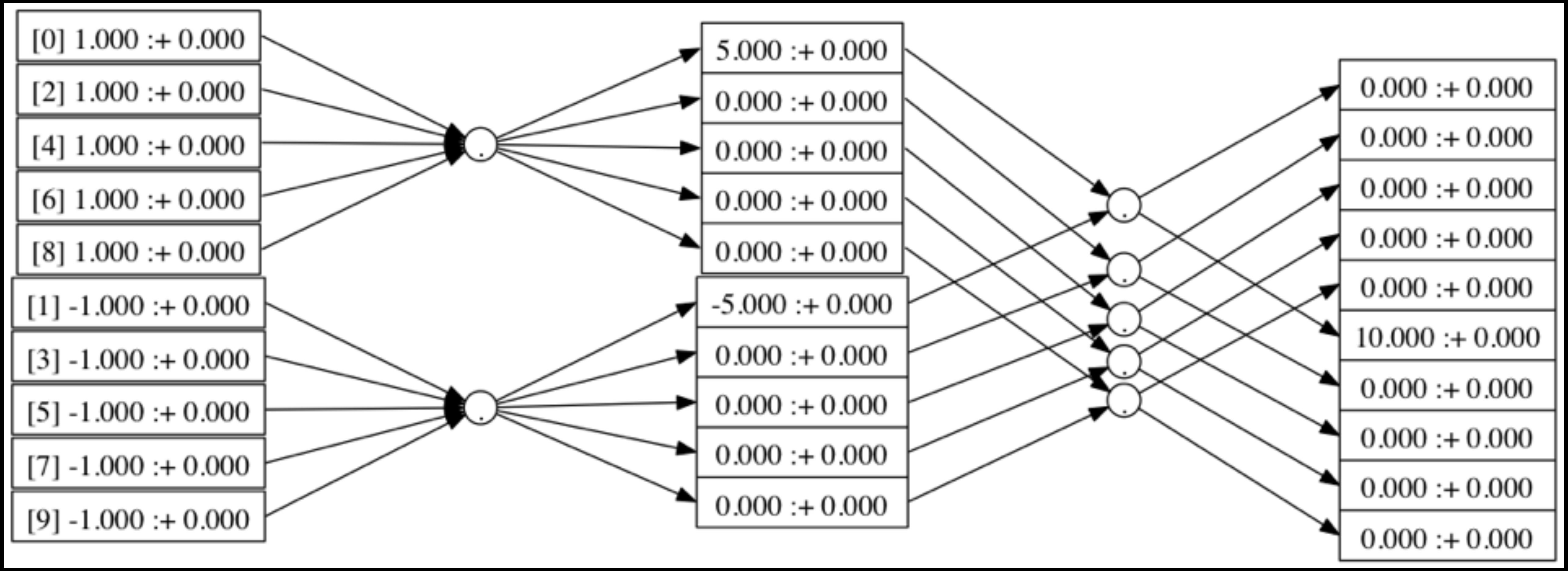


"v0.0.2" - Data flow is correct, but:

- **There're twice as many arrows as we need.**
- **What computations are being performed?**

Example Generic FFT

```
tData6  = newTreeData [(2, False), (5, False)]
                      [1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0]
```



"v0.0.3" - Data flow is now easier to visualize, but:

- **We're still not showing the computations being performed.**

A quick aside...

The numbers in the new graph look much nicer; what changed?

```
newtype PrettyDouble = PrettyDouble {
    value :: Double
  } deriving (Num, Eq, Ord, Fractional, Floating, Real, RealFrac, RealFloat)
instance Show PrettyDouble where
    show = printf "%10.3g" . zeroThresh . value
        where zeroThresh y =
                if abs y < 1.0e-10
                then 0.0
                else y
```

…and a newbie mistake

- I thought I'd be helpful:

```
--newFFTTree = TreeBuilder ( fmap (fmap PrettyDouble) . buildMixedRadixTree )
newFFTTree = TreeBuilder ( buildMixedRadixTree . makePrettyData )
    where makePrettyData td = TreeData {
                                        modes   = oldModes
                                       , values = newValues
                                      }
              where oldModes  = modes td
                    newValues = map (fmap PrettyDouble) (values td)

toUgly :: Complex PrettyDouble -> Complex Double
toUgly z = (value $ realPart z) :+ (value $ imagPart z)
```

Survey said: "XXXXXXXXXXXXXXX!"

Now…

- The only lines in my code mentioning *PrettyDouble*:

```
type FFTTree = GenericLogTree (Complex PrettyDouble)
instance LogTree FFTTree (Complex PrettyDouble) where
```

- And everything just works!

***Moral*: If you love Haskell, set it free.**

***Or, if you're into Zen*: If you want Haskell to do something, stop trying to make it do that.**

My theory…

I think this type declaration:

```
type FFTTree = GenericLogTree (Complex PrettyDouble)
```

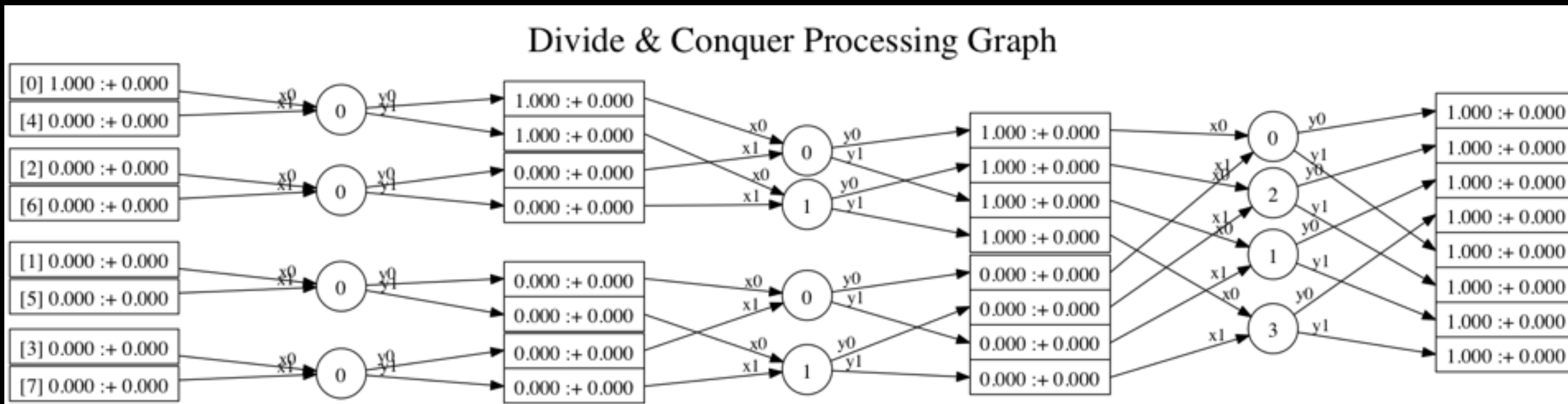
is causing these constants:

```
tData6 = newTreeData [(2, False), (5, False)]
  [1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0]
```

to be typecast as *Complex PrettyDouble*, via the Haskell type inferencing system, when the tree is constructed and that that type casting is surviving all subsequent operations when the tree is evaluated to produce the final output.

(See Appendix A for relevant code.)

Example Generic FFT



0: y0 = (1.000 :+ 0.000) * x0 + (1.000 :+ 0.000) * x1   y1 = (1.000 :+ 0.000) * x0 + (-1.000 :+ 0.000) * x1

1: y0 = (1.000 :+ 0.000) * x0 + (0.000 :+ -1.000) * x1   y1 = (1.000 :+ 0.000) * x0 + (0.000 :+ 1.000) * x1

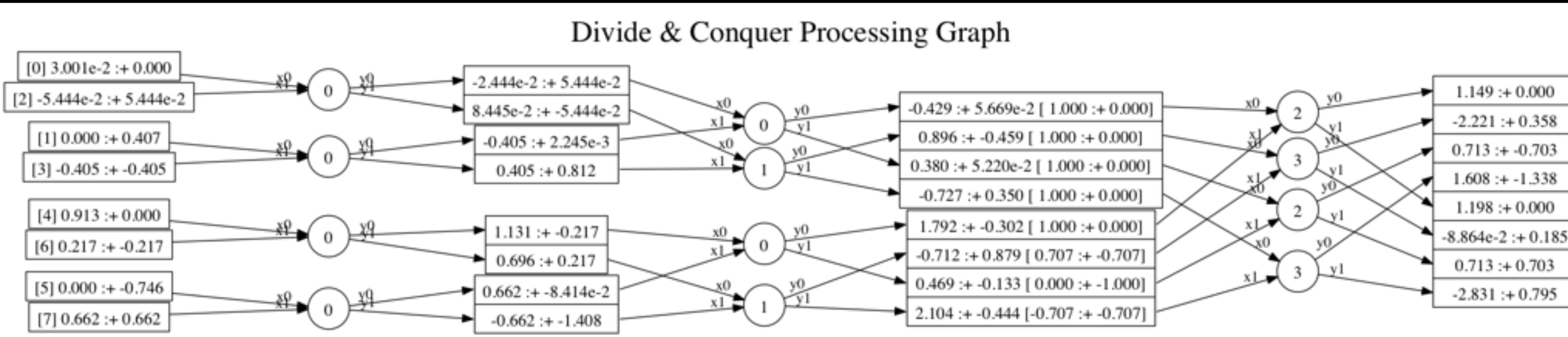
2: y0 = (1.000 :+ 0.000) * x0 + (0.707 :+ -0.707) * x1   y1 = (1.000 :+ 0.000) * x0 + (-0.707 :+ 0.707) * x1

3: y0 = (1.000 :+ 0.000) * x0 + (-0.707 :+ -0.707) * x1   y1 = (1.000 :+ 0.000) * x0 + (0.707 :+ 0.707) * x1

Computational Node Legend

“v0.0.4” - Now, we see what computations are being performed. However, we’d like to see the twiddle factors getting inserted into the sub-transforms, in the DIF case.

Example Generic FFT



0:	y0 = (1.000 :+ 0.000) * x0 + (1.000 :+ 0.000) * x1	y1 = (1.000 :+ 0.000) * x0 + (-1.000 :+ 0.000) * x1
1:	y0 = (1.000 :+ 0.000) * x0 + (0.000 :+ -1.000) * x1	y1 = (1.000 :+ 0.000) * x0 + (0.000 :+ 1.000) * x1
2:	y0 = (1.000 :+ 0.000) * x0 + (1.000 :+ 0.000) * x1	y1 = (1.000 :+ 0.000) * x0 + (1.000 :+ 0.000) * x1
3:	y0 = (1.000 :+ 0.000) * x0 + (-1.000 :+ 0.000) * x1	y1 = (1.000 :+ 0.000) * x0 + (-1.000 :+ 0.000) * x1

Computational Node Legend

“v0.0.5” - Now, we see everything. The only problem is… the answer is WRONG! And I can’t figure out why!

Mixed Radix Testing

Trial	Decimation Style			Result
	Stage 1	Stage 2	Stage 3	
0	DIT	DIT	X	Pass
1	DIF	DIF	X	Pass
2	DIT	DIF	X	Pass
3	DIF	DIT	X	Fail

Testing the Generic Code

```
prop_fft_test testVal = collect (length values) $ collect modes $
    (getEval $ buildTree newFFTTree tData) == answer
    where types  = testVal :: FFTTestVal
          tData  = newTreeData modes values
          modes  = snd $ getVal testVal
          values = fst $ getVal testVal
          answer = dft values
```

Generating Test Values

```haskell
newtype FFTTestVal = FFTTestVal {
    getVal :: ([Complex PrettyDouble], [(Int, Bool)])
} deriving (Show)
instance Arbitrary FFTTestVal where
    arbitrary = do
        n       <- choose (2, 100)
        let radices = primeFactors n
        rValues <- vectorOf n $ choose (-1.0, 1.0)
        let values  = map (:+ 0.0) rValues
        -- This doesnt work, although I think it should; why not?:
        --difs   <- vectorOf (length radices) $ elements [True, False]
        dif     <- elements [True, False]
        let difs    = replicate (length radices) dif
        return $ FFTTestVal (values, zip radices difs)
```

Generating Prime Factors

```
-- Determines the prime factors of an integer.
primeFactors :: Int -> [Int]
primeFactors n
    | isPrime n = [n]
    | otherwise = primeDivisor : primeFactors result
      where result       = n `div` primeDivisor
            primeDivisor = head $ filter ((== 0) . (n `mod`)) primes

-- Tests an integer for primeness.
isPrime :: Int -> Bool
isPrime n = elem n $ takeWhile (<= n) primes

-- Prime generator. (Sieve of Eratosthenes)
primes :: [Int]
primes = primesRecurse [2..]

primesRecurse :: [Int] -> [Int]
primesRecurse ns = n : primesRecurse ms
    where n  = head ns
          ms = filter ((/= 0) . (`mod` n)) ns
```

Test Results

```
Test suite test-treeviz: RUNNING...
+++ OK, passed 100 tests:
 4% 38, [(2,False),(19,False)]
 4% 17, [(17,True)]
 3% 11, [(11,False)]
 2% 89, [(89,False)]
 2% 8, [(2,True),(2,True),(2,True)]
 2% 77, [(7,True),(11,True)]
 2% 77, [(7,False),(11,False)]
 2% 74, [(2,False),(37,False)]
 2% 72, [(2,True),(2,True),(2,True),(3,True),(3,True)]
…
```

Final Tidbits

- cabal install treeviz
- https://github.com/capn-freako/treeviz
- ToDo:
  - Solve mixed decimation style breakdown.
  - Reconsider separation of element extraction and tree evaluation.
  - Move to Foldable/Traversable.

Questions?

Thank you!

Appendix A - Code Example

```haskell
type GenericLogTree a = Tree (Maybe a, [Int], Int, Bool)
class (t ~ GenericLogTree a) => LogTree t a | t -> a where
    evalNode :: t -> [a] -- Evaluates a node in a tree, returning a list of values of the original type.

type FFTTree = GenericLogTree (Complex PrettyDouble)
newtype TreeBuilder t = TreeBuilder {
   buildTree :: LogTree t a => TreeData a -> Either String t
}
newFFTTree :: TreeBuilder FFTTree
newFFTTree = TreeBuilder buildMixedRadixTree

newTreeData :: [(Int, Bool)] -- ^ Decomposition modes : (radix, DIF_flag).
            -> [a]           -- ^ Values for populating the tree.
            -> TreeData a    -- ^ Resultant data structure for passing to tree constructor.
newTreeData modes values = TreeData {
                               modes  = modes
                             , values = values
                           }

buildMixedRadixTree :: TreeData a -> Either String (GenericLogTree a)
buildMixedRadixTree td = mixedRadixTree td_modes td_values
    where td_modes  = modes td
          td_values = values td

mixedRadixTree :: [(Int, Bool)] -> [a] -> Either String (GenericLogTree a)

tData6  = newTreeData [(2, False), (5, False)]
                      [1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0]
exeMain = do
    let tree   = buildTree newFFTTree tData6
    let res    = getEval tree

-- Helper function to evaluate a node.
getEval (Left msg)   = []
getEval (Right tree) = evalNode tree
```