FFT via Circat 2

Dave <capn.freako@gmail.com>
&
Conal <conal@conal.net>

Haskell Meetup
Front Row Education
San Francisco, CA
November 12, 2015

Not so long ago, at a Haskell meetup not really that far away…

- Conal Elliott presents his recent work on ***parallel scan***.

- Conal reveals that he's found a way to reduce *work/time* of this operation, below certain limits previously believed to be fundamental.

- Dave gets REALLY excited, because…

A new curiosity awakens.

- In his day job, Dave lives and breaths the *Fast Fourier Transform* (FFT), another one of these logarithmically broken down computations, which is believed to have been optimized.

- Dave starts drawing lots of pictures of FFT data flow, for differently parameterized implementations…

- …and realizes what a pain in the ass it is!

A new visualization tool is born.

- Dave decides to try writing a *computation breakdown visualization* tool. The result is *TreeViz*, available on both Hackage and GitHub.

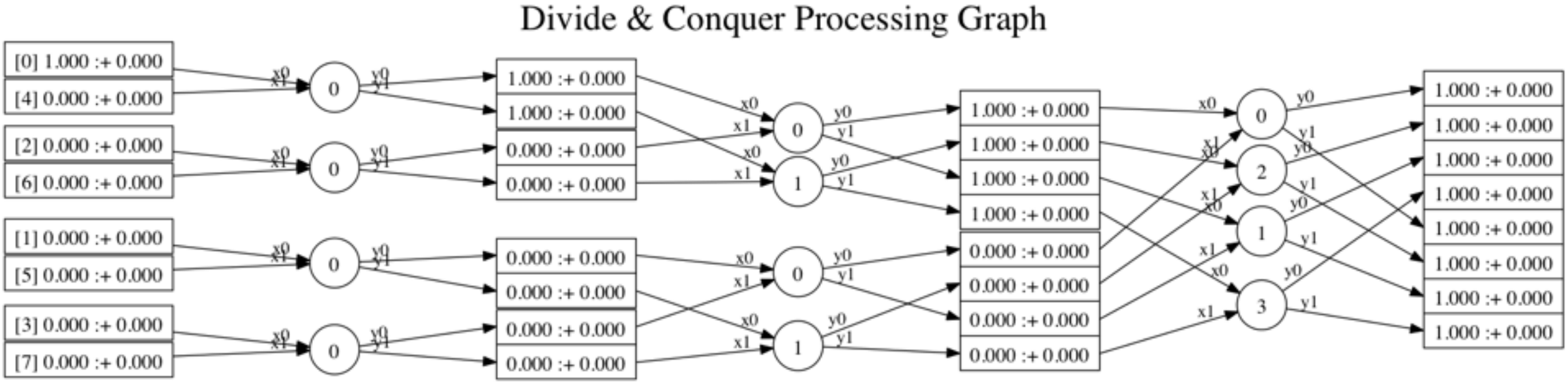
Treeviz

A set of types, classes, and functions for visualizing computation decomposition trees.

1 Introduction

This library can assist you in visualizing how computation is broken down, or decomposed, by certain *Divide-And-Conquer* type algorithms. Such algorithms are often capable of significantly reducing the order of complexity of an operation, by taking advantage of recursions, which occur when the original input is broken into halves, thirds, etc. Such repeated recursive breakdown often produces tree structures from an initially linear data input. And being able to visualize how the elements of these trees are recombined, when the operation is evaluated, can help us make smarter choices about how to apply a particular type of massively parallel computational resource to the computational problem.

A year and a half ago…



Computational Node Legend

A useful pursuit, but…

- Assumes container will always be a list.
- What about trees? Or, other traversable structures?
- How about defining FFT for a few *primitives*, and then *deriving* it for more complex structures?

Enter *Circat*…

- Conal Elliott's machinery for representing circuits (and other things), using *Cartesian Closed Categories*.

- Contains some very useful data structures, and higher order functions on them, for doing FFT (and other things).

- We'll use: *RTree*, which is a perfect binary leaf tree parameterized by depth, because it naturally enforces a balanced, log-2 breakdown of computation and provides some very elegant decimation mechanisms.

FFT in Haskell, for *List* (ca. early 2014)

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

FFT in Haskell, for *RTree* (now)

```
-- Phasor, as a function of tree depth.
phasor :: (IsNat n, RealFloat a, Enum a) =>
    Nat n -> RTree n (Complex a)
phasor n = scanlTEx (*) 1 (pure phaseDelta)
    where phaseDelta = cis ((-pi) / 2 ** natToZ n)

-- Radix-2, DIT FFT
fft_r2_dit :: (IsNat n, RealFloat a, Enum a) =>
    RTree n (Complex a) -> RTree n (Complex a)
fft_r2_dit = fft_r2_dit' nat

fft_r2_dit' :: (RealFloat a, Enum a) => Nat n ->
    RTree n (Complex a) -> RTree n (Complex a)
fft_r2_dit'  Zero     = id
fft_r2_dit' (Succ n) = toB
                     . inP (uncurry (+) &&& uncurry (-))
                     . secondP (liftA2 (*) (phasor n))
                     . fmap (fft_r2_dit' n)
                     . bottomSplit
```

Some “old vs. new” comparisons…

- Old:

```
[ wn ** (fromIntegral k) | k <- [0..]]
 where wn  = exp ( 0.0 :+ ( -2.0 * pi / n))
       n   = fromIntegral (length xs)
```

- New:

```
-- Phasor, as a function of tree depth.
phasor :: (IsNat n, RealFloat a, Enum a) =>
     Nat n -> RTree n (Complex a)
phasor n = scanlTEx (*) 1 (pure phaseDelta)
 where phaseDelta = cis ((-pi) / 2 ** natToZ n)
```

How to define* phasor *more generally?

- Old:

```
radix2_DIT []     = []
radix2_DIT [x]    = [x]
radix2_DIT xs     = (++) (zipWith (+) xes xos)
                         (zipWith (-) xes xos)
    where xes = radix2_DIT (evens xs)
          xos = zipWith (*)
                  (radix2_DIT (odds xs))
                  {phasor}
```

- New:

```
-- Radix-2, DIT FFT
fft_r2_dit'  Zero     = id
fft_r2_dit' (Succ n) =
    toB
      . inP (uncurry (+) &&& uncurry (-))
      . secondP (liftA2 (*) (phasor n))
      . fmap (fft_r2_dit' n)
      . bottomSplit
```

Description has been elevated in its level of abstraction.

Future Directions

- FFT as a class:

```
class (LScan f) => FFT f where
    fft  :: (RealFloat a) => f (Complex a) -> f (Complex a)

instance IsNat n => FFT (RTree n) where
  fft = fft' nat
    where   fft' :: (RealFloat a) => Nat n -> RTree n (Complex a)
                                           -> RTree n (Complex a)
            fft' Zero     = id
            fft' (Succ n) = inDIT $ fftP . P.secondP addPhase
                                         . fmap (fft' n)
                where   inDIT g  = RT.toB . g . bottomSplit
                        fftP     = P.inP (uncurry (+) &&& uncurry (-))
                        addPhase = liftA2 (*) id phasor
```

phasor ?

Future Directions

- FFT as a class (cont'd.):

```
phasor :: (Applicative f, Foldable f, LScan f, RealFloat b) =>
    f a -> f (Complex b)
phasor f = fst $ lproducts (pure phaseDelta)
    where   phaseDelta = cis ((-pi) / fromIntegral n)
            n          = flen f

flen :: (Foldable f) => f a -> Int
flen = foldl' (flip ((+) . const 1)) 0
```

phasor ***has now been generalized to any Foldable, Applicative, LScan.***

Future Directions

- Defined instances for:
  - Id
  - Pair
  - (:.) (functor composition)
- FFT of higher order structures *derived* from above.

Questions?

Thank you!

References

TreeViz: https://wiki.haskell.org/Treeviz

Circat: https://github.com/conal/circat

Lambda-CCC: https://github.com/conal/lambda-ccc

FFT: https://en.wikipedia.org/wiki/Fast_Fourier_transform