Generic FFT

Conal Elliott

Target

August 31, 2016

Paths from circular motion

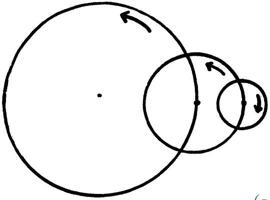


FIGURE 1

(source)

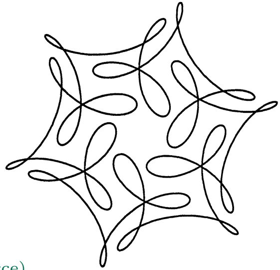


FIGURE 2

- Circular motion:
  - Frequency/speed, f
  - Radius, r
  - Starting angle, a
- Combine several motions: center of each follows path of previous.
- Observe final motion.
- Another example

Paths from circular motions

$$x(t) = \sum_{(f,r,a)\in S} (r\cos(2\pi f t + a), r\sin(2\pi f t + a))$$

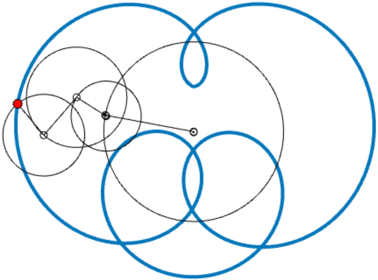
More succinct in complex polar form:

$$x(t) = \sum_{(f,r,a)\in S} r\, e^{i(2\pi f t + a)}$$

Yet more succinct with $X = re^{ia}$:

$$x(t) = \sum_{(f,X)\in S} X\, e^{i2\pi f t}$$

Questions



(source)

- Which motions can be generated in this way?
- How to generate the circular components for a given motion?

Answers: all periodic functions; the Fourier transform.

Some other uses of the Fourier transform

- Hearing (roughly)
- Geocentrism (Ptolemy's deferent & epicycle)
- Sound & image compression
- Audio equalization
- Solving differential equations
- Convolution, for signal processing, probability, neural networks
- Derivatives of signals

Discrete Fourier Transform (DFT)

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}} \qquad k = 0, \ldots, N-1$$

Direct implementation does $O(N^2)$ work.

DFT in Haskell

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i 2 \pi k n}{N}}$$

$$dft :: \forall f\ a \ldots . \Rightarrow Unop\ (f\ (Complex\ a))$$
$$dft\ xs = omegas\ (size\ @f)\ \hat{\$}\ xs$$

$$omegas :: \ldots \Rightarrow Int \rightarrow g\ (f\ (Complex\ a))$$
$$omegas\ n = powers <\$> powers\ (exp\ (-i * 2 * \pi\ /\ fromIntegral\ n))$$

$$powers :: \ldots \Rightarrow a \rightarrow f\ a$$
$$powers = fst \circ lscanAla\ Product \circ pure$$

$(\hat{\$}) :: \ldots \Rightarrow n\ (m\ a) \rightarrow m\ a \rightarrow n\ a$ -- matrix × vector
$$mat\ \hat{\$}\ vec = (\cdot vec) <\$> mat$$

$(\cdot) :: \ldots \Rightarrow f\ a \rightarrow f\ a \rightarrow a$ -- dot product
$$u \cdot v = sum\ (liftA_2\ (*)\ u\ v)$$

Fast Fourier transform (FFT)

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}$$

- DFT in $O(N \log N)$ work
- Better numeric properties than naive DFT
- Long history:
  - Gauss: 1805
  - Danielson & Lanczos: 1942
  - Cooley & Tukey: 1965

A summation trick

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}$$

For composite bounds:

$$\sum_{n=0}^{N_1 N_2 - 1} F(n) \;=\; \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} F(N_1 n_2 + n_1)$$

Factoring DFT — math

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}$$

From Wikipedia:

When this re-indexing is substituted into the DFT formula for nk, the $N_1 n_2 N_2 k_1$ cross term vanishes (its exponential is unity), and the remaining terms give

$$X_{N_2 k_1 + k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_1 N_2} \cdot (N_1 n_2 + n_1) \cdot (N_2 k_1 + k_2)}$$

$$= \sum_{n_1=0}^{N_1-1} \left[e^{-\frac{2\pi i}{N} n_1 k_2} \right] \left(\sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right) e^{-\frac{2\pi i}{N_1} n_1 k_1}$$

Factoring DFT — math

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}$$

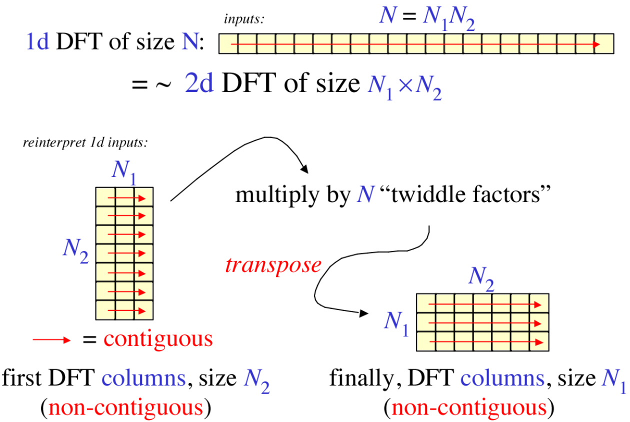
From Wikipedia:

When this re-indexing is substituted into the DFT formula for nk, the $N_1 n_2 N_2 k_1$ cross term vanishes (its exponential is unity), and the remaining terms give

$$X_{N_2 k_1 + k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_1 N_2} \cdot (N_1 n_2 + n_1) \cdot (N_2 k_1 + k_2)}$$

$$= \underbrace{\sum_{n_1=0}^{N_1-1} \left[e^{-\frac{2\pi i}{N} n_1 k_2} \right] \overbrace{\left(\sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right)}^{\text{inner FFTs}} e^{-\frac{2\pi i}{N_1} n_1 k_1}}_{\text{outer FFTs}}$$

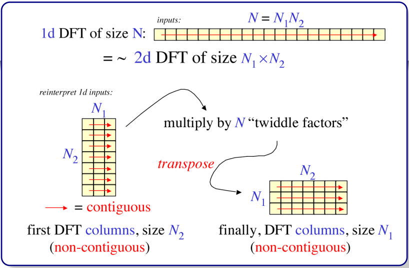
Factoring DFT — pictures



(source)

How might we implement in Haskell?

Factoring DFT — Haskell



Factor types, not numbers!

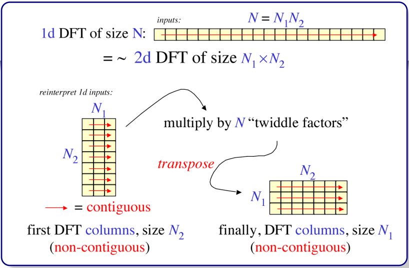
$$\mathbf{newtype}\ (g \circ f)\ a = O\ (g\ (f\ a))$$

$$\mathbf{instance}\ (\mathit{Sized}\ g, \mathit{Sized}\ f) \Rightarrow \mathit{Sized}\ (g \circ f)\ \mathbf{where}$$
$$\quad \mathit{size} = \mathit{size}\ @g * \mathit{size}\ @f$$

Also closed under composition:

- *Functor*
- *Applicative*
- *Foldable*
- *Traversable*

Factoring DFT — Haskell



class $FFT\ f$ **where**

  type $Reverse\ f :: * \rightarrow *$

  $fft :: f\ \mathbb{C} \rightarrow Reverse\ f\ \mathbb{C}$

instance ... $\Rightarrow FFT\ (g \circ f)$ **where**

  type $Reverse\ (g \circ f) = Reverse\ f \circ Reverse\ g$

  $fft = O \circ ffts' \circ transpose \circ twiddle \circ ffts' \circ unO$

$ffts' :: \ldots \Rightarrow g\ (f\ \mathbb{C}) \rightarrow Reverse\ g\ (f\ \mathbb{C})$

$ffts' = transpose \circ fmap\ fft \circ transpose$

$twiddle :: \ldots \Rightarrow g\ (f\ \mathbb{C}) \rightarrow g\ (f\ \mathbb{C})$

$twiddle = (liftA_2 \circ liftA_2)\ (*)\ (omegas\ (size\ @(g \circ f)))$

Optimizing fft for $g \circ f$

$$
\begin{aligned}
& \mathit{ffts}' \circ \mathit{transpose} \circ \mathit{twiddle} \circ \mathit{ffts}' \\
\equiv \\
& \quad \mathit{transpose} \circ \mathit{fmap}\ \mathit{fft} \circ \mathit{transpose} \\
& \circ \mathit{transpose} \\
& \circ \mathit{twiddle} \\
& \circ \mathit{transpose} \circ \mathit{fmap}\ \mathit{fft} \circ \mathit{transpose} \\
\equiv \\
& \mathit{transpose} \circ \mathit{fmap}\ \mathit{fft} \circ \mathit{twiddle} \circ \mathit{transpose} \circ \mathit{fmap}\ \mathit{fft} \circ \mathit{transpose} \\
\equiv \\
& \mathit{traverse}\ \mathit{fft} \circ \mathit{twiddle} \circ \mathit{traverse}\ \mathit{fft} \circ \mathit{transpose}
\end{aligned}
$$

Binary FFT

Uniform pairs:

$$\textbf{data}\ Pair\ a = a :\#\ a\ \textbf{deriving}\ (Functor, Foldable, Traversable)$$

$$\textbf{instance}\ Sized\ Pair\ \textbf{where}\ size = 2$$

$$\textbf{instance}\ FFT\ Pair\ \textbf{where}$$
$$\quad \textbf{type}\ Reverse\ Pair = Pair$$
$$\quad fft = dft$$

Equivalently,

$$fft\ (a :\#\ b) = (a + b) :\#\ (a - b)$$

Exponentiating functors

$$f^n = \overbrace{f \circ \cdots \circ f}^{n \text{ times}}$$

Example: $Pair^n$ is a depth-n, perfect, binary, leaf tree.

Associating functor composition

$$(h \circ g) \circ f \simeq h \circ (g \circ f)$$

Does the same FFT algorithm arise?

Associating functor exponentiation

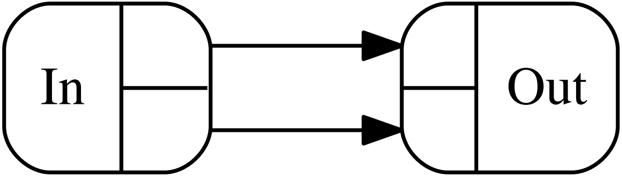
Right-associated/top-down:

$$\begin{aligned}&\textbf{type family } \mathit{RPow}\ h\ n \textbf{ where}\\&\quad \mathit{RPow}\ h\ Z \qquad = \mathit{Id}\\&\quad \mathit{RPow}\ h\ (S\ n) = h \circ \mathit{RPow}\ h\ n\end{aligned}$$

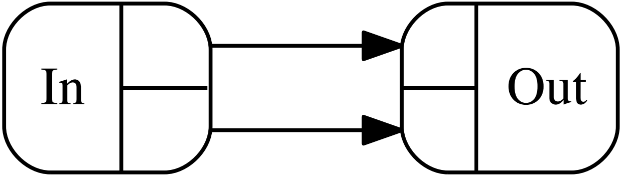
Left-associated/bottom-up:

$$\begin{aligned}&\textbf{type family } \mathit{LPow}\ h\ n \textbf{ where}\\&\quad \mathit{LPow}\ h\ Z \qquad = \mathit{Id}\\&\quad \mathit{LPow}\ h\ (S\ n) = \mathit{LPow}\ h\ n \circ h\end{aligned}$$

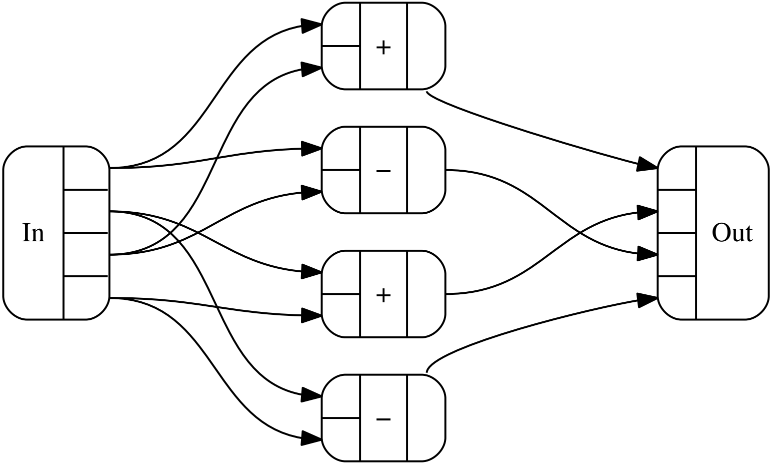
$fft\ @(RPow\ Pair\ 0)$



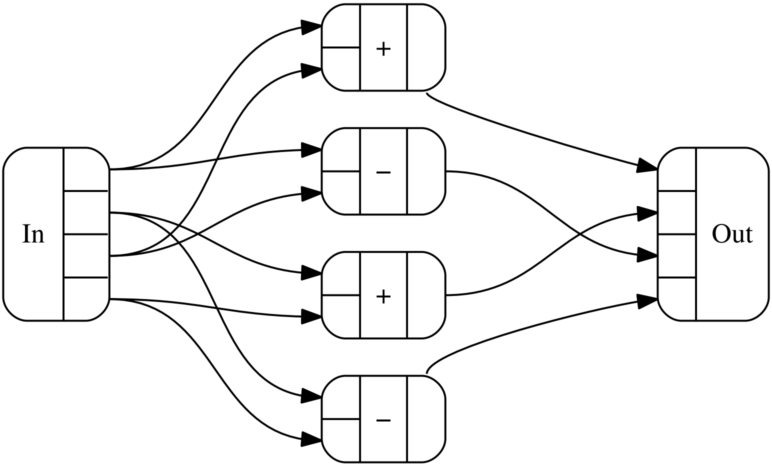
fft @($LPow$ $Pair$ 0)



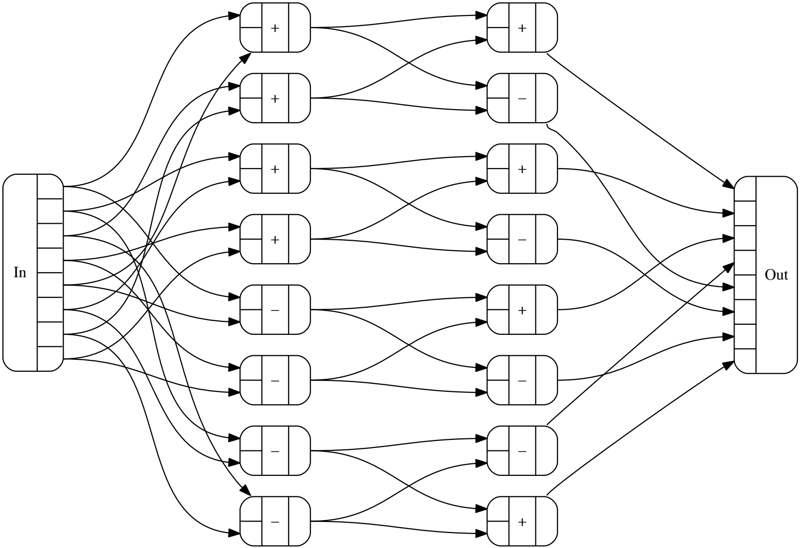
$fft\ @(RPow\ Pair\ 1)$



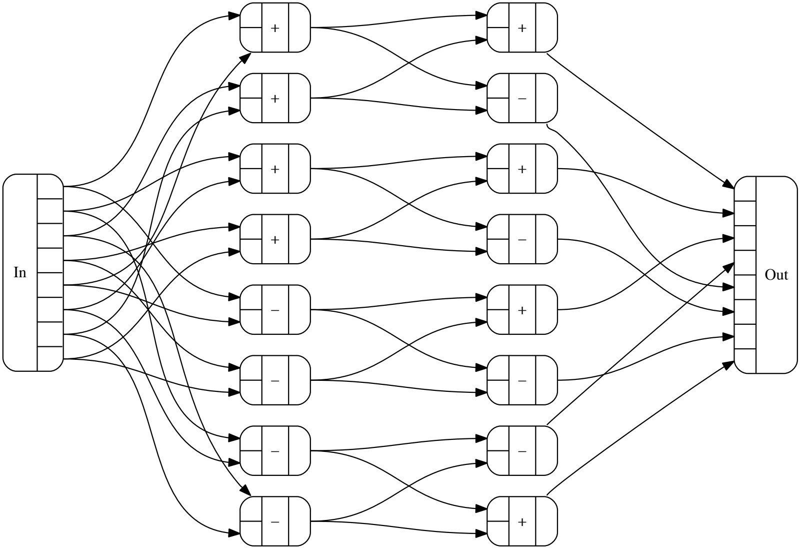
fft @(*LPow Pair* 1)



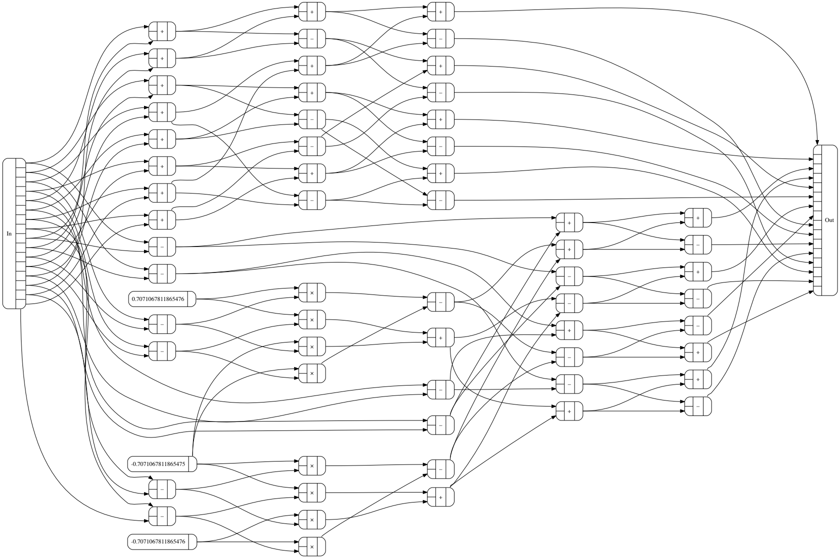
fft @(*RPow Pair* 2)



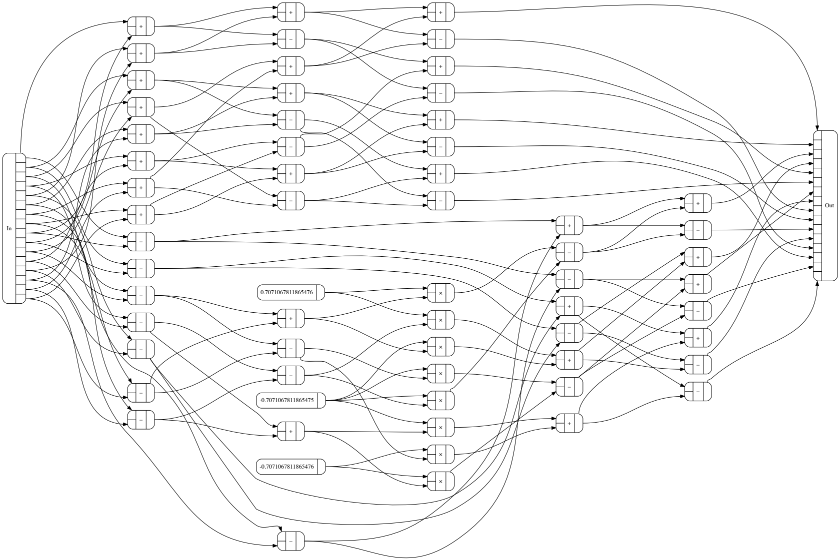
$fft\ @(LPow\ Pair\ 2)$



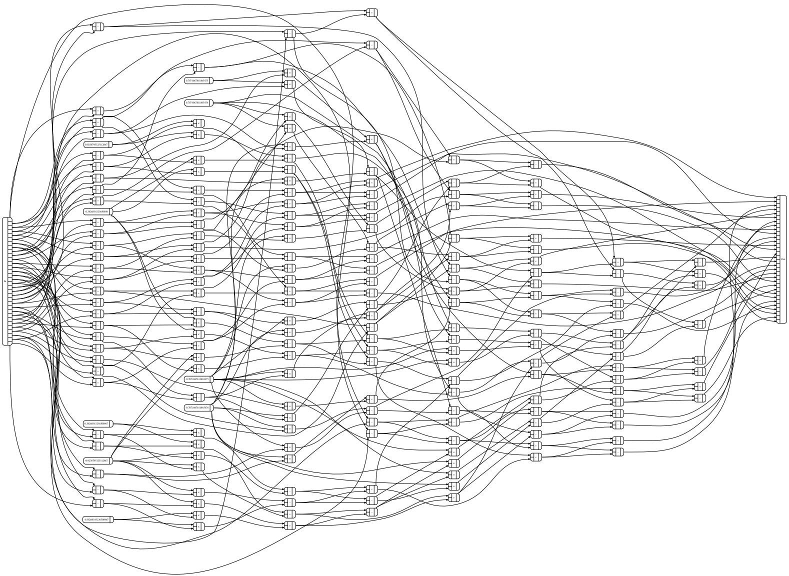
fft @(*RPow Pair* 3)



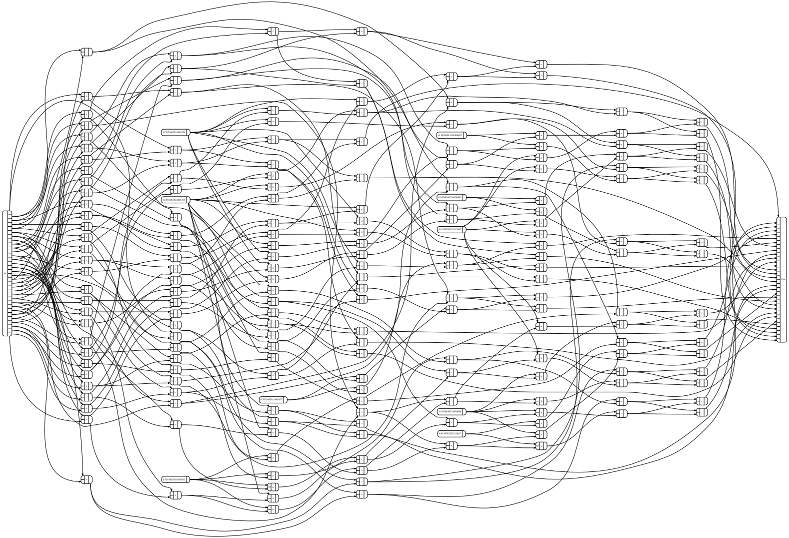
$fft\ @(LPow\ Pair\ 3)$



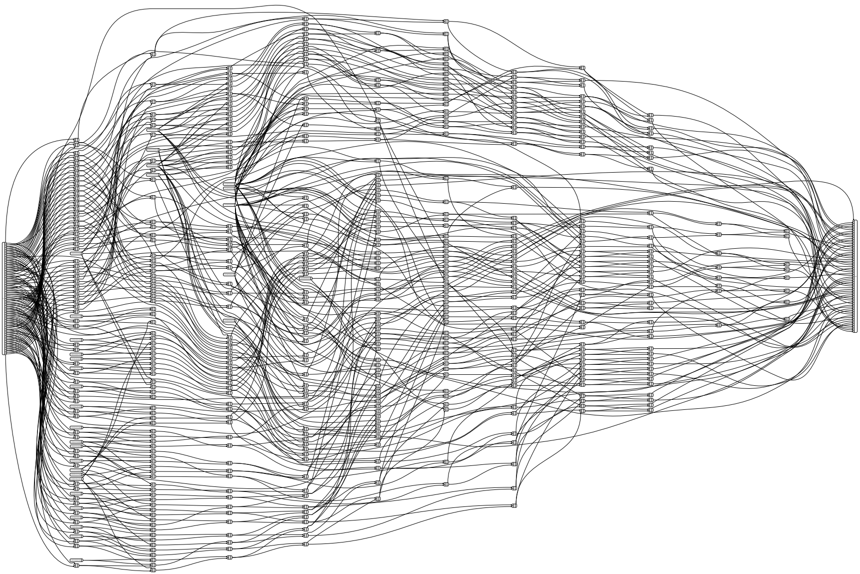
fft @(*RPow Pair* 4)



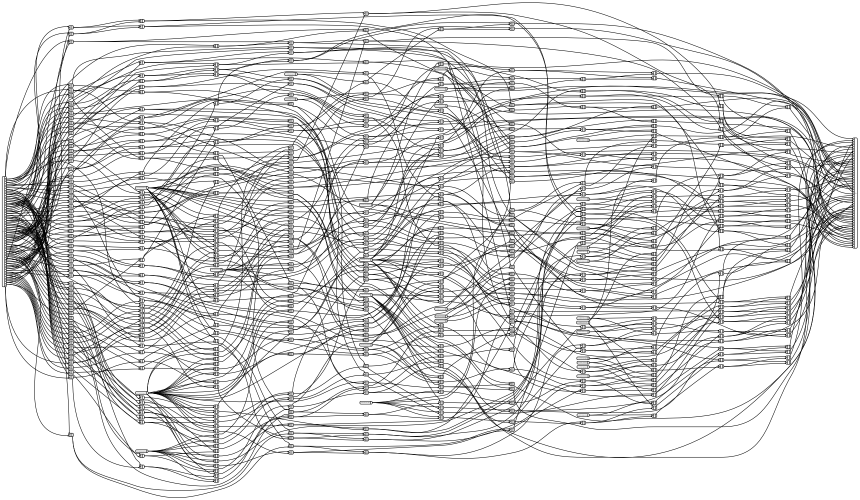
fft @(*LPow Pair* 4)



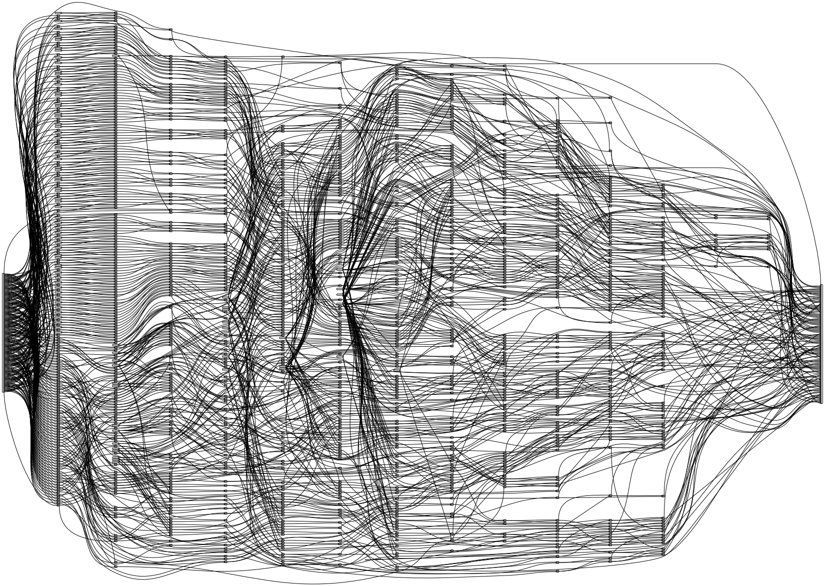
fft @(*RPow Pair* 5)



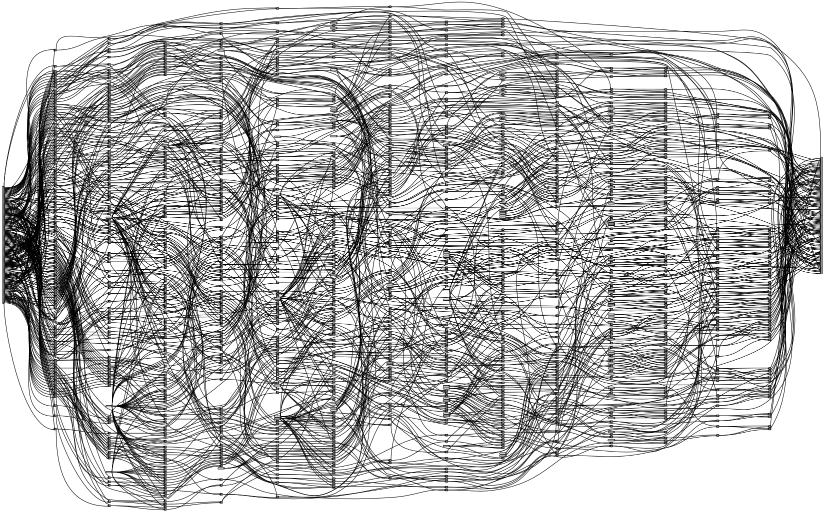
fft @($LPow$ $Pair$ 5)



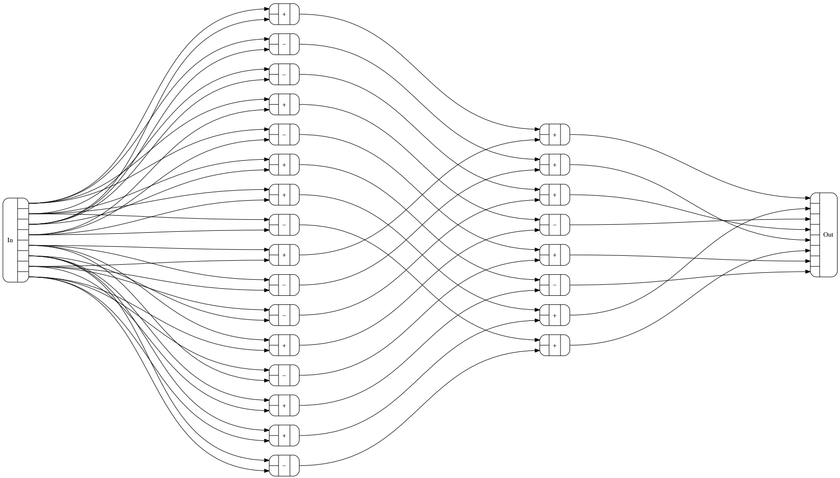
fft @(*RPow Pair* 6)



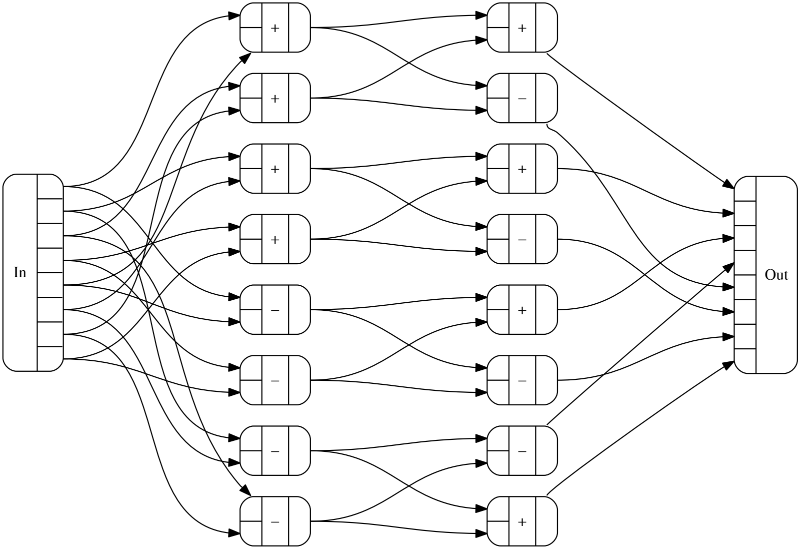
fft @(*LPow Pair* 6)



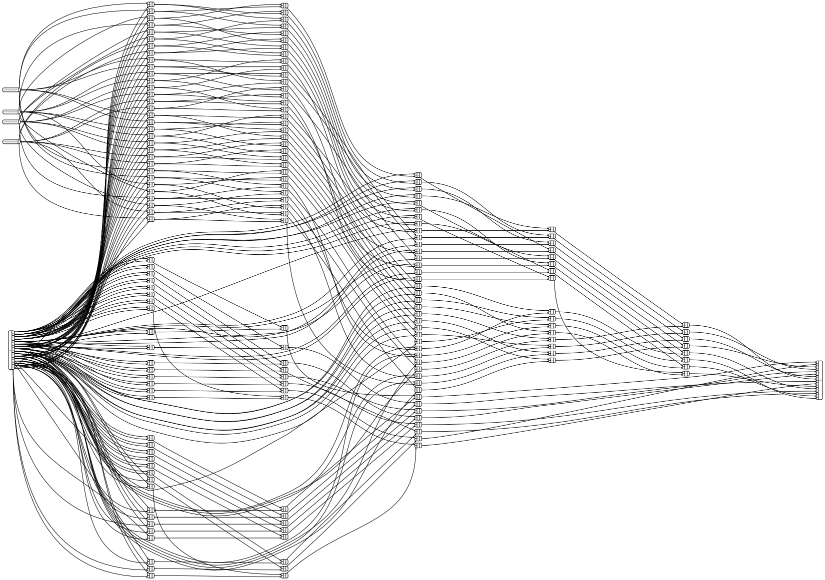
dft @(*RPow Pair* 2)



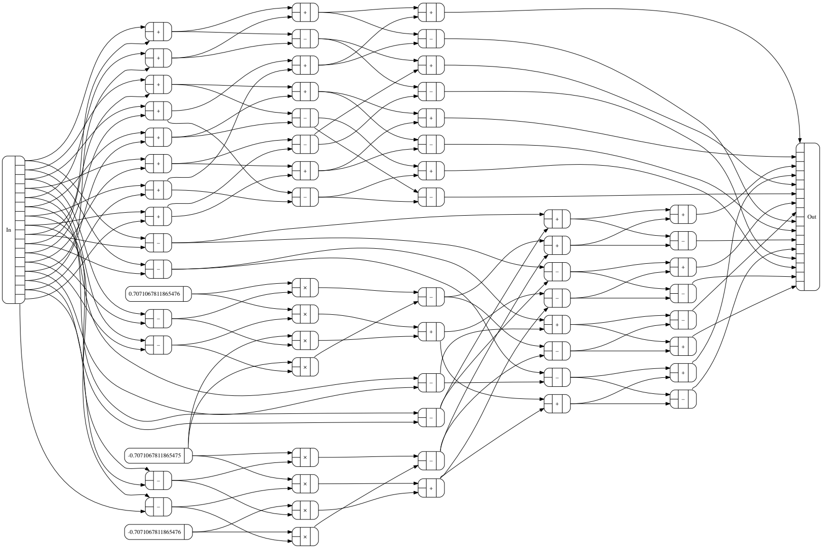
fft @($RPow\ Pair\ 2$)



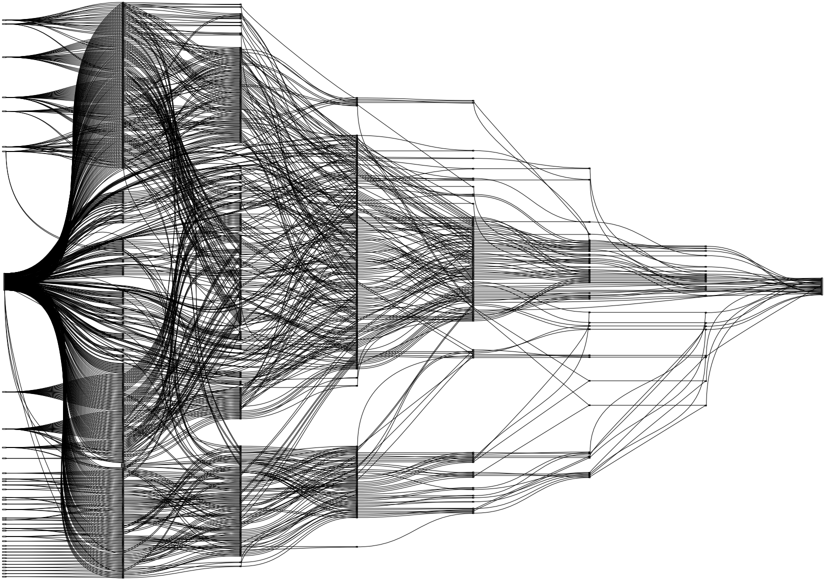
dft @(*RPow Pair* 3)



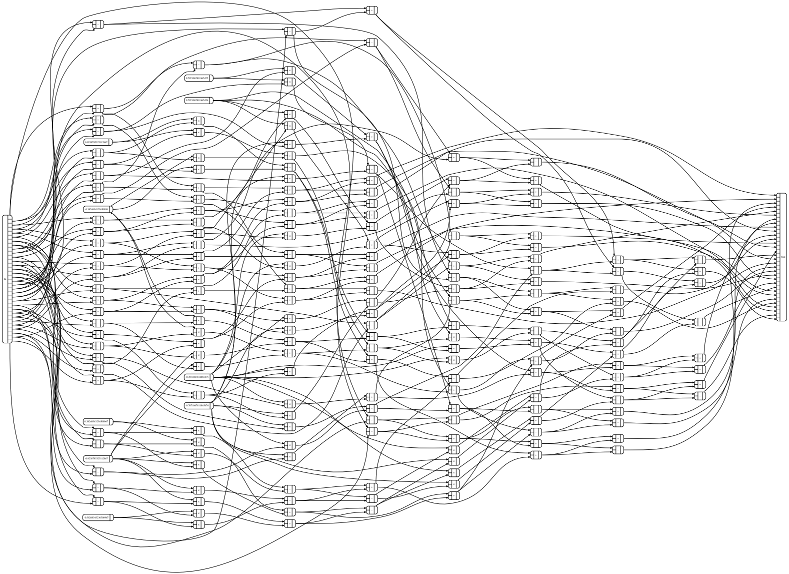
fft @(*RPow Pair* 3)



dft @(*RPow Pair* 4)



fft @(*RPow Pair* 4)



Generic FFT

$$
\begin{array}{l}
\textbf{class}\ \mathit{FFT}\ f\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Reverse}\ f :: * \rightarrow * \\
\quad \mathit{fft} :: f\ \mathbb{C} \rightarrow \mathit{Reverse}\ f\ \mathbb{C} \\
\quad \textbf{default}\ \mathit{fft} :: (\mathit{Generic}_1\ f, \mathit{Generic}_1\ (\mathit{Reverse}\ f), \mathit{FFT}\ (\mathit{Rep}_1\ f) \\
\qquad\qquad\qquad , \mathit{Reverse}\ (\mathit{Rep}_1\ f) \sim \mathit{Rep}_1\ (\mathit{Reverse}\ f)) \\
\qquad\qquad\quad \Rightarrow f\ \mathbb{C} \rightarrow \mathit{Reverse}\ f\ \mathbb{C} \\
\quad \mathit{fft}\ \mathit{xs} = \mathit{to}_1 \circ \mathit{fft}\ \mathit{xs} \circ \mathit{from}_1
\end{array}
$$

using `GHC.Generics`.

Concluding remarks

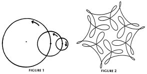
Type-driven, parallel-friendly algorithm:

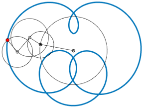
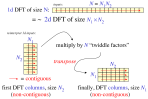
- Factor types, not numbers.
- Well-known algorithms as special cases (DIT & DIF).
- Works well with `GHC.Generics`.

In contrast to array algorithms:

- Elegantly compositional.
- Free of index computations.
- Safe from out-of-bounds errors.

Picture credits

- 

Frank A. Farris
- 
Ivan Kuckir
- 

Steven G. Johnson

Extras

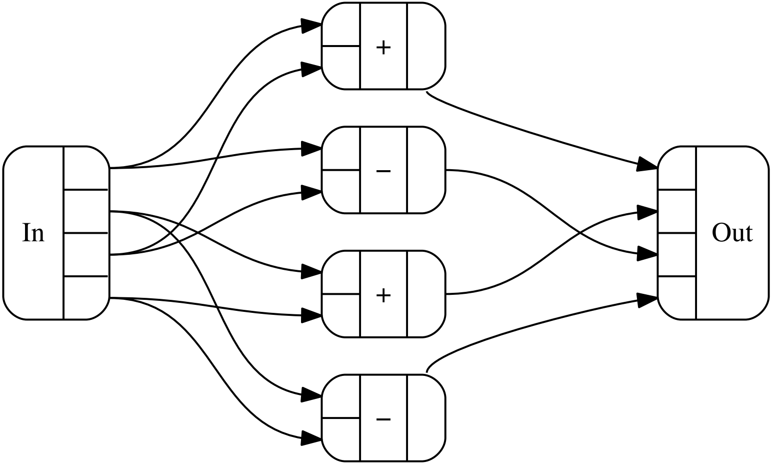
Bushes

$$
\begin{aligned}
&\textbf{type family } \mathit{Bush}\ n \textbf{ where}\\
&\quad \mathit{Bush}\ Z \quad\ \ = \mathit{Pair}\\
&\quad \mathit{Bush}\ (S\ n) = \mathit{Bush}\ n \circ \mathit{Bush}\ n
\end{aligned}
$$

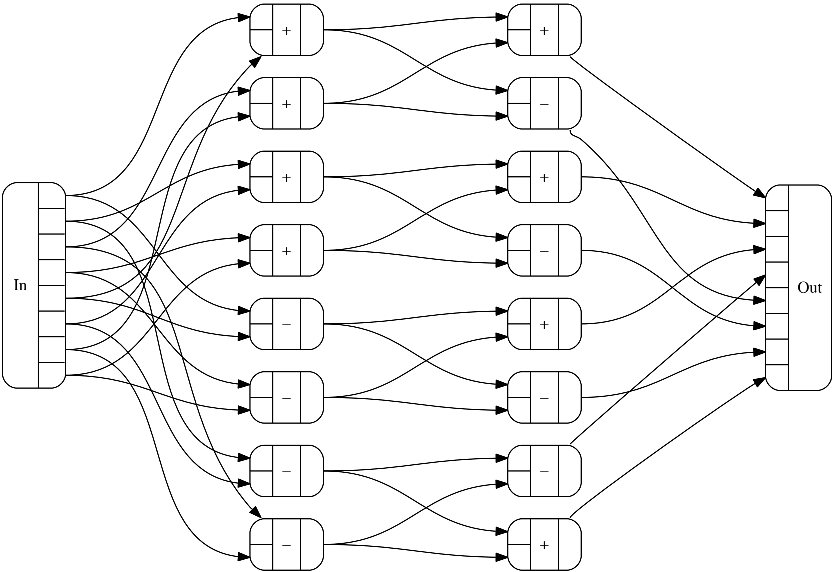
Notes:

- Composition-balanced counterpart to *LPow* and *RPow*.
- Variation of *Bush* type in *Nested Datatypes* by Bird & Meertens.
- Size 2^{2^n}, i.e., $2, 4, 16, 256, 65536, \ldots$.
- Easily generalizes beyond pairing and squaring.

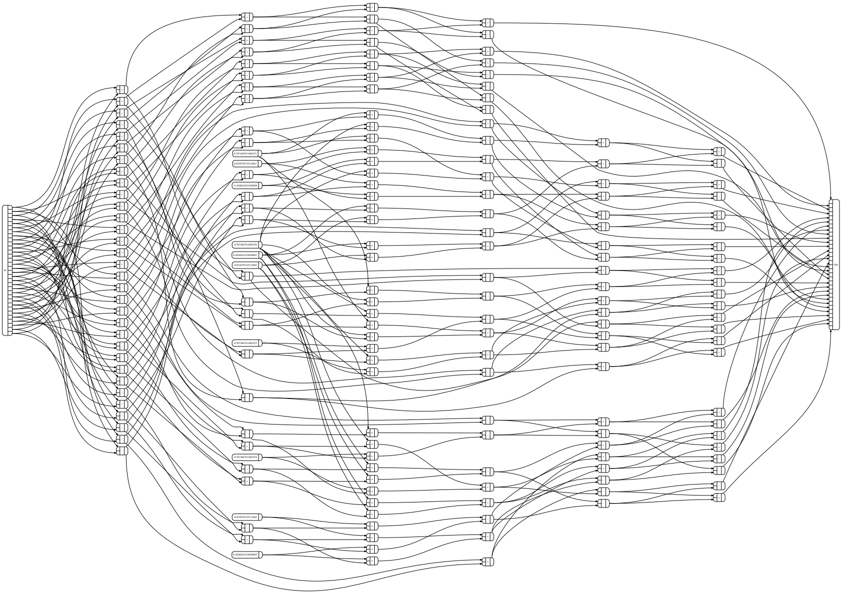
$\mathit{fft}\ @(\mathit{Bush}\ 0)$



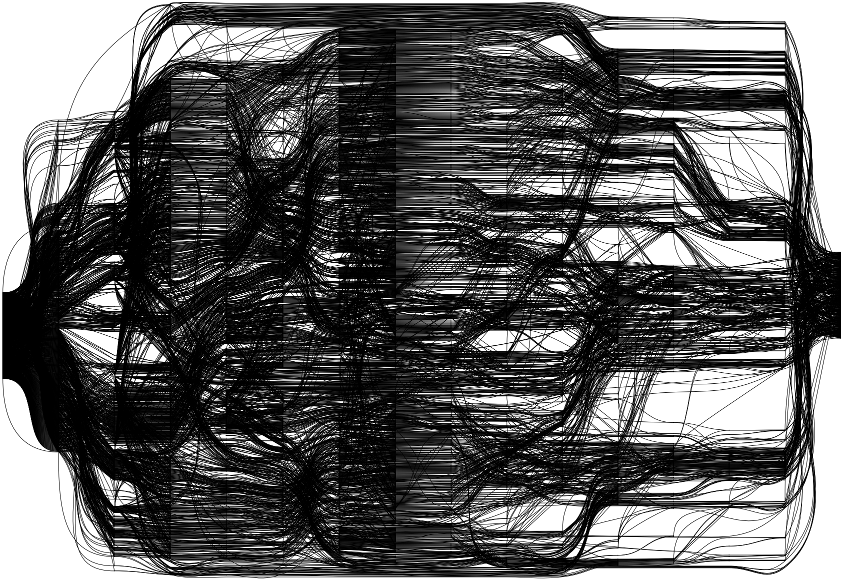
fft @($Bush$ 1)



fft @(*Bush* 2)



fft @(*Bush* 3)



Comparison

For 16 complex inputs and results:

Type	+	×	−	total	max depth
RPow Pair 4	74	40	74	197	8
LPow Pair 4	74	40	74	197	8
Bush 2	72	32	72	186	6

For 256 complex inputs and results:

Type	+	×	−	total	max depth
RPow Pair 8	2690	2582	2690	8241	20
LPow Pair 8	2690	2582	2690	8241	20
Bush 3	2528	1922	2528	7310	14