Generic Functional Parallel Algorithms

Scan and FFT

Conal Elliott

Target, USA

September 2017

Arrays

- Dominant data type for parallel programming (even functional).
- Unsafe (indexing is partial).
- Obfuscate parallel algorithms (array encodings).

Generic building blocks

$$
\begin{array}{llll}
\textbf{data} & V & a & \text{-- void} \\
\textbf{newtype} & U & a = U & \text{-- unit} \\
\textbf{newtype} & I & a = I\ a & \text{-- singleton} \\
\textbf{data} & (f + g) & a = L\ (f\ a) \mid R\ (g\ a) & \text{-- sum} \\
\textbf{data} & (f \times g) & a = f\ a \times g\ a & \text{-- product} \\
\textbf{newtype} & (g \circ f) & a = O_1\ (g\ (f\ a)) & \text{-- composition}
\end{array}
$$

Plan:

- Define algorithm for each.
- Use directly, *or*
- automatically via (derived) encodings.
- Data types give rise to (correct) algorithms.

Some data types

Vectors

$$n = \overbrace{I \times \cdots \times I}^{n \text{ times}}$$

Left-associated:

$$\begin{aligned}
&\textbf{type family } \overleftarrow{n} \textbf{ where} \\
&\quad \overleftarrow{0} = U \\
&\quad \overleftarrow{n+1} = \overleftarrow{n} \times I
\end{aligned}$$

Right-associated:

$$\begin{aligned}
&\textbf{type family } \overrightarrow{n} \textbf{ where} \\
&\quad \overrightarrow{0} = U \\
&\quad \overrightarrow{n+1} = I \times \overrightarrow{n}
\end{aligned}$$

Perfect trees

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

Left-associated/bottom-up:

$$\begin{array}{l} \textbf{type family } h^{\uparrow n} \textbf{ where} \\ \quad h^{\uparrow 0} \quad\;\; = I \\ \quad h^{\uparrow n+1} = h^{\uparrow n} \circ h \end{array}$$

Right-associated/top-down:

$$\begin{array}{l} \textbf{type family } h^{\downarrow n} \textbf{ where} \\ \quad h^{\downarrow 0} \quad\;\; = I \\ \quad h^{\downarrow n+1} = h \circ h^{\downarrow n} \end{array}$$

Scan

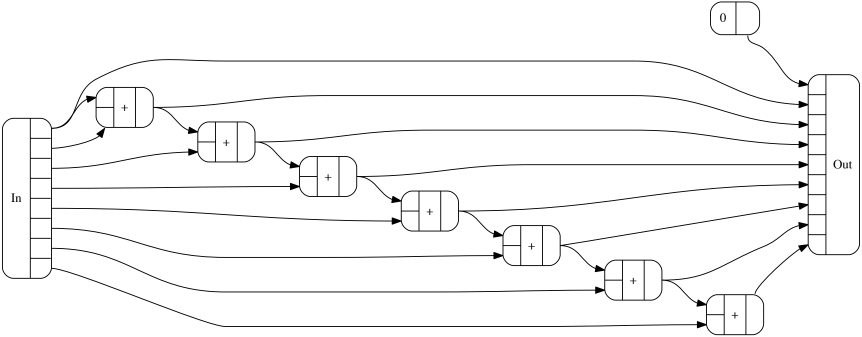
Prefix sum (left scan)

Given $a_1, \ldots, a_n$, compute

$$b_k = \sum_{1 \leqslant i < k} a_i \qquad \text{for } k = 1, \ldots, n+1$$

Note that a_k does *not* influence b_k.

Linear left scan



Work: $O(n)$

Depth: $O(n)$ (ideal parallel "time")

Linear *dependency chain* thwarts parallelism.

Scan interface

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{LScan}\ f\ \textbf{where} \\
\quad \mathit{lscan} :: \mathit{Monoid}\ a \Rightarrow f\ a \rightarrow f\ a \times a
\end{array}
$$

Easy instances

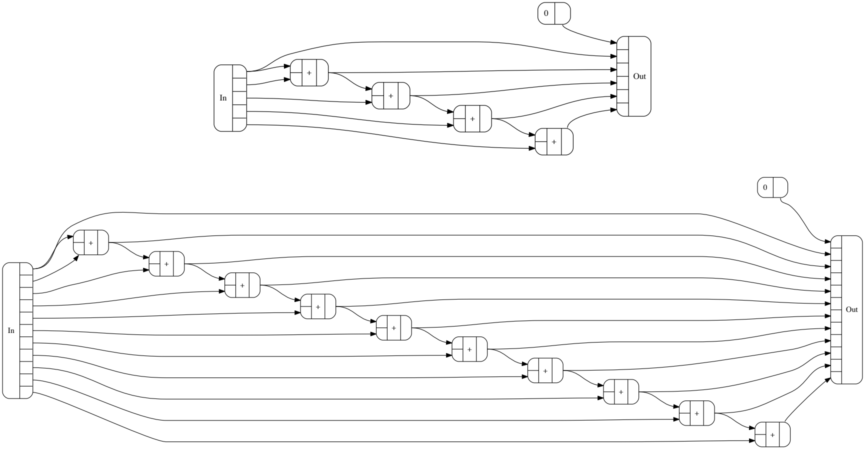
$$\textbf{instance}\ LScan\ V\ \textbf{where}\ lscan = \lambda\ \textbf{case}$$

$$\textbf{instance}\ LScan\ U\ \textbf{where}\ lscan\ U = (U, \emptyset)$$

$$\textbf{instance}\ LScan\ I\ \ \textbf{where}\ lscan\ (I\ a) = (I\ \emptyset, a)$$

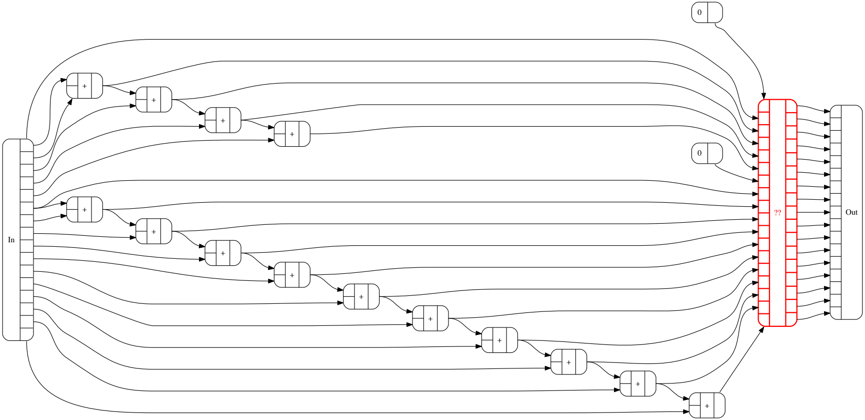
$$\begin{array}{l}\textbf{instance}\ (LScan\ f, LScan\ g) \Rightarrow LScan\ (f + g)\ \textbf{where} \\ \quad lscan\ (L\ fa) = \mathit{first}\ L\ (lscan\ fa) \\ \quad lscan\ (R\ ga) = \mathit{first}\ R\ (lscan\ ga)\end{array}$$

Product example: $\overleftarrow{5} \times \overleftarrow{11}$

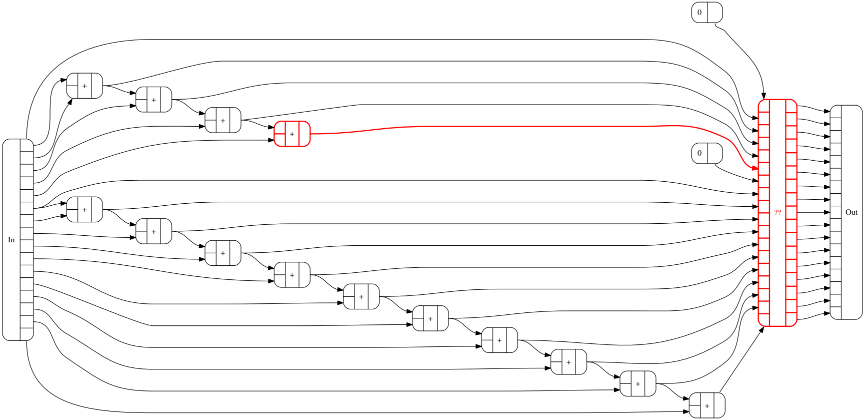


Then what?

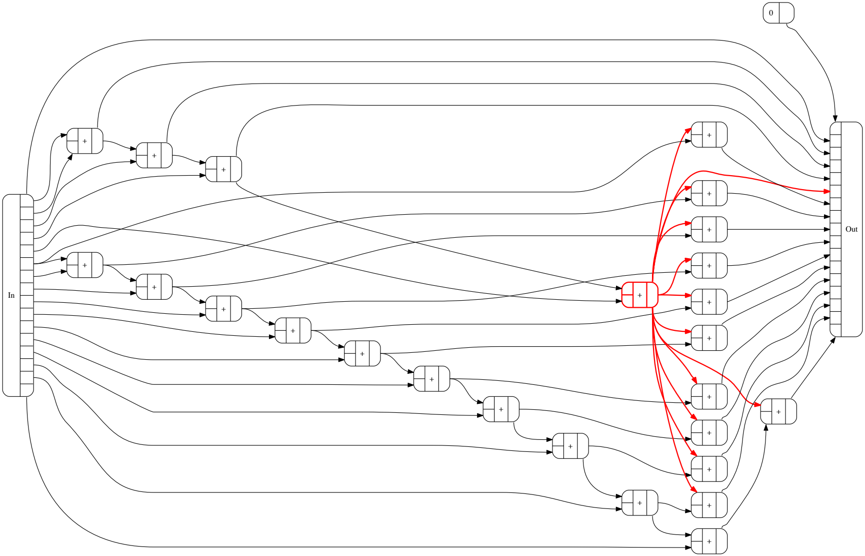
Combine?



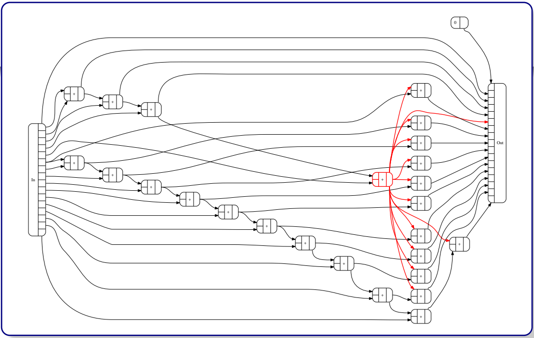
Combine?



Right adjustment

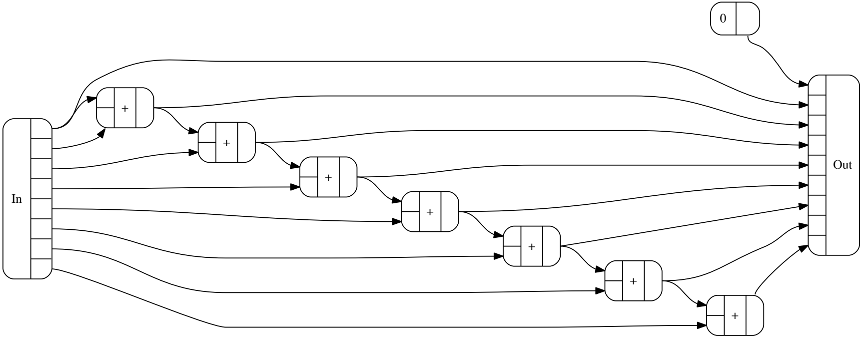


Products

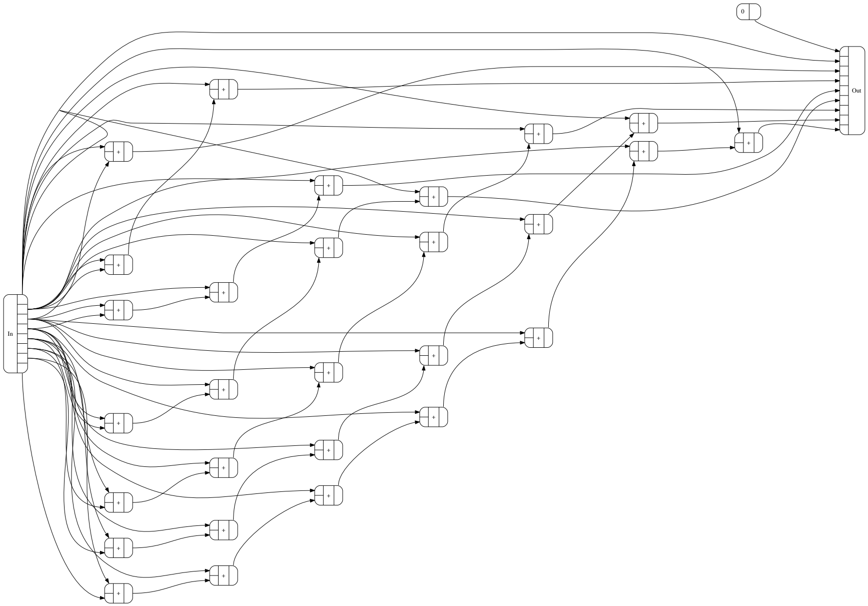


$$
\begin{array}{l}
\textbf{instance}\ (\mathit{LScan}\ f, \mathit{LScan}\ g) \Rightarrow \mathit{LScan}\ (f \times g)\ \textbf{where} \\
\quad \mathit{lscan}\ (\mathit{fa} \times \mathit{ga}) = (\mathit{fa}' \times ((\mathit{fx} \oplus)\ \mathbin{<\$>}\ \mathit{ga}'), \mathit{fx} \oplus \mathit{gx}) \\
\qquad \textbf{where} \\
\qquad\quad (\mathit{fa}', \mathit{fx}) = \mathit{lscan}\ \mathit{fa} \\
\qquad\quad (\mathit{ga}', \mathit{gx}) = \mathit{lscan}\ \mathit{ga}
\end{array}
$$

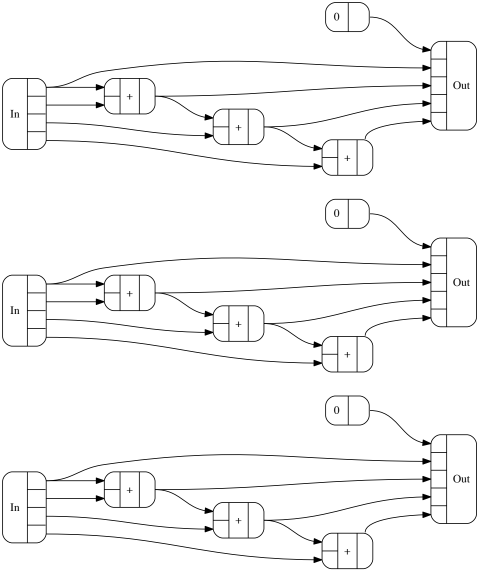
work: 7, depth: 7



work: 28, depth: 7

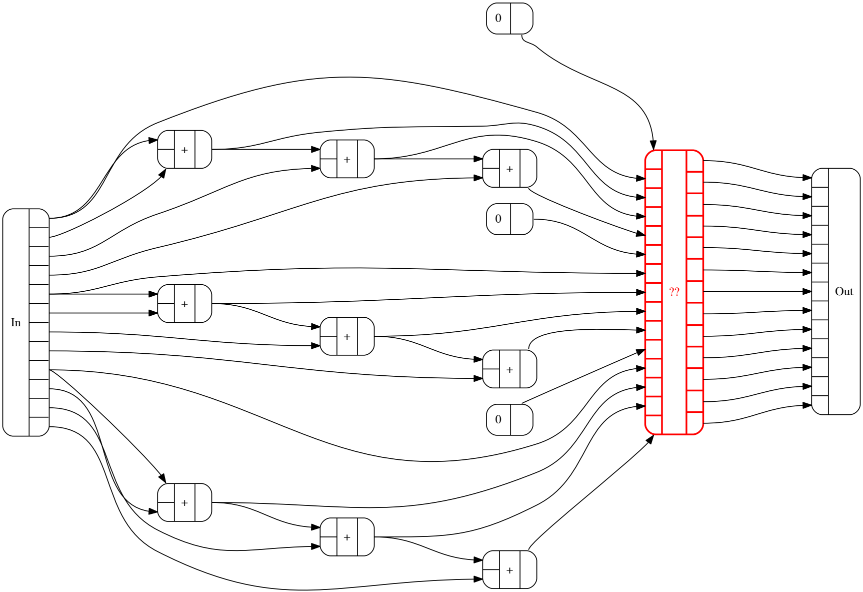


Composition example: $\overleftarrow{3} \circ \overleftarrow{4}$

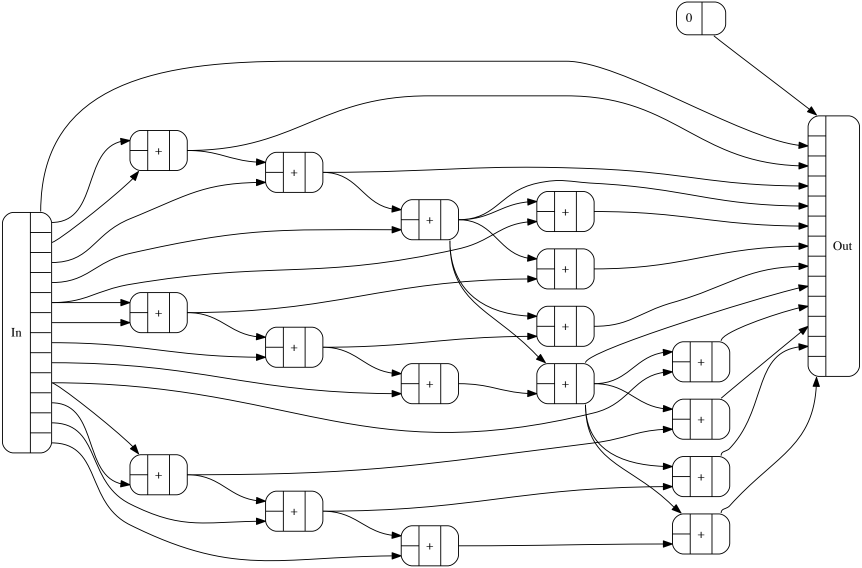


Then what?

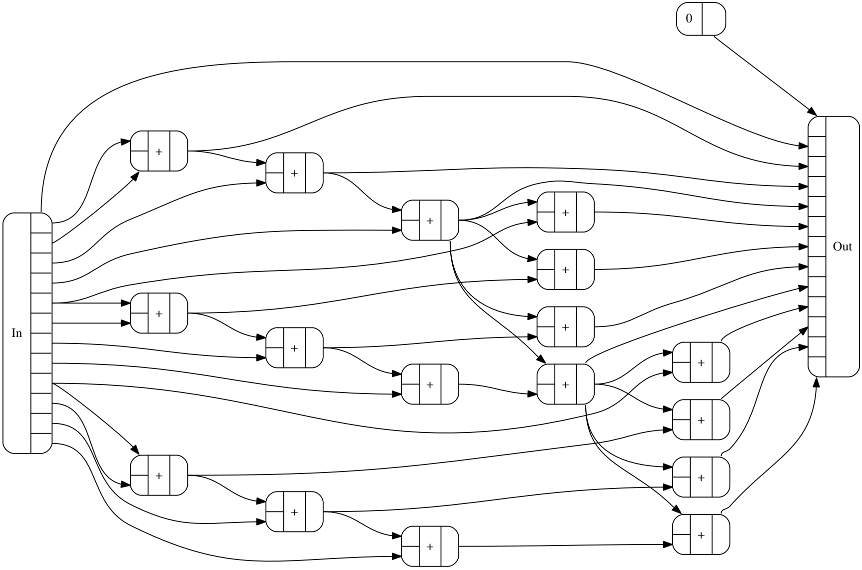
Combine?



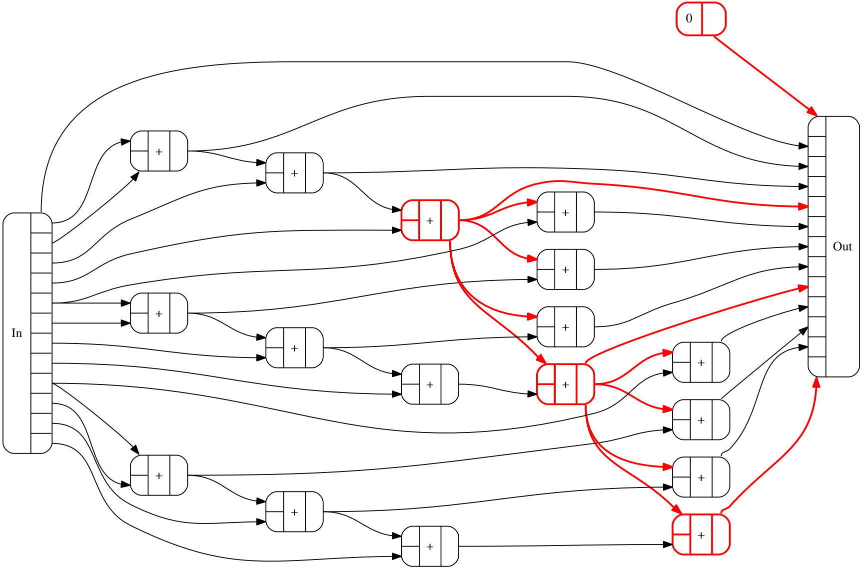
$(\overleftarrow{4} \times \overleftarrow{4}) \times \overleftarrow{4}$



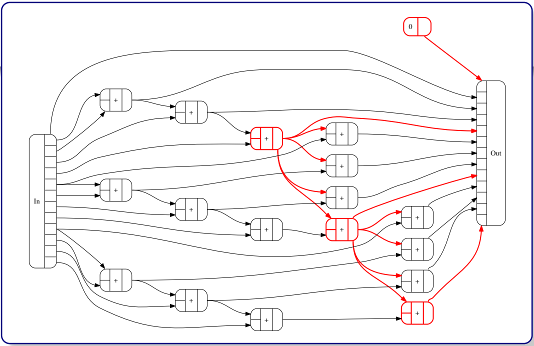
$\overleftarrow{3} \circ \overleftarrow{4}$



$\overleftarrow{3} \circ \overleftarrow{4}$

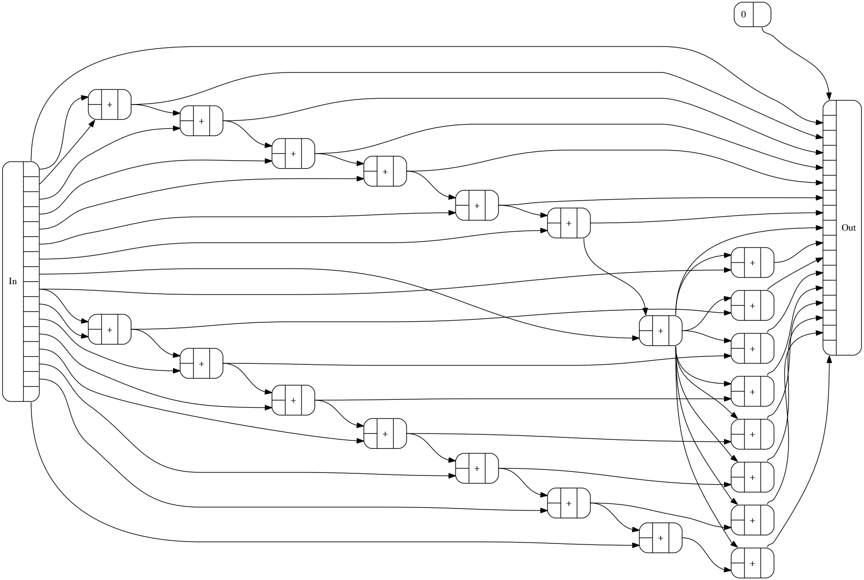


Composition



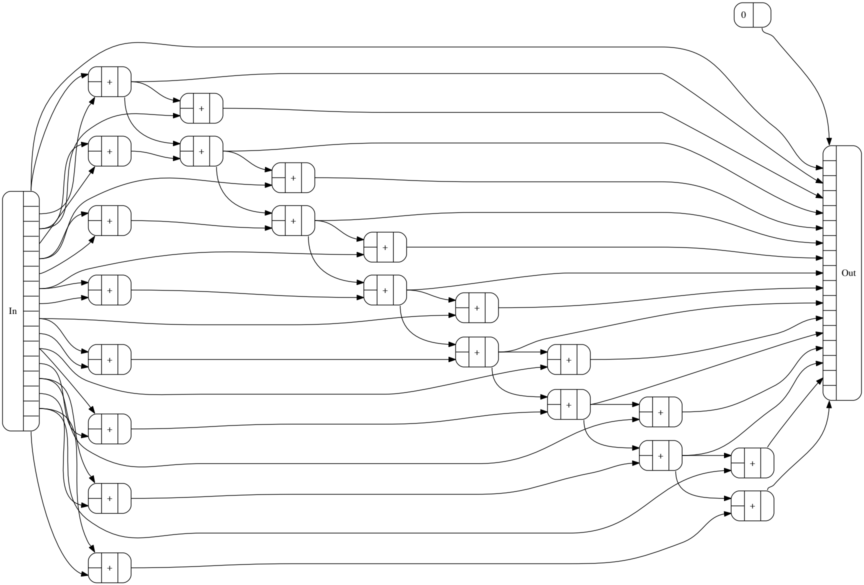
$$\textbf{instance}\ (LScan\ g, LScan\ f, Zip\ g) \Rightarrow LScan\ (g \circ f)\ \textbf{where}$$
$$\quad lscan\ (O_1\ gfa) = (O_1\ (zipWith\ adjustl\ tots'\ gfa'), tot)$$
$$\quad\quad \textbf{where}$$
$$\quad\quad\quad (gfa', tots) = unzip\ (lscan \mathbin{<\$>} gfa)$$
$$\quad\quad\quad (tots', tot) = lscan\ tots$$
$$\quad\quad\quad adjustl\ t \quad = fmap\ (t \oplus)$$

$2 \circ \overleftarrow{8}$ work: 22, depth: 8



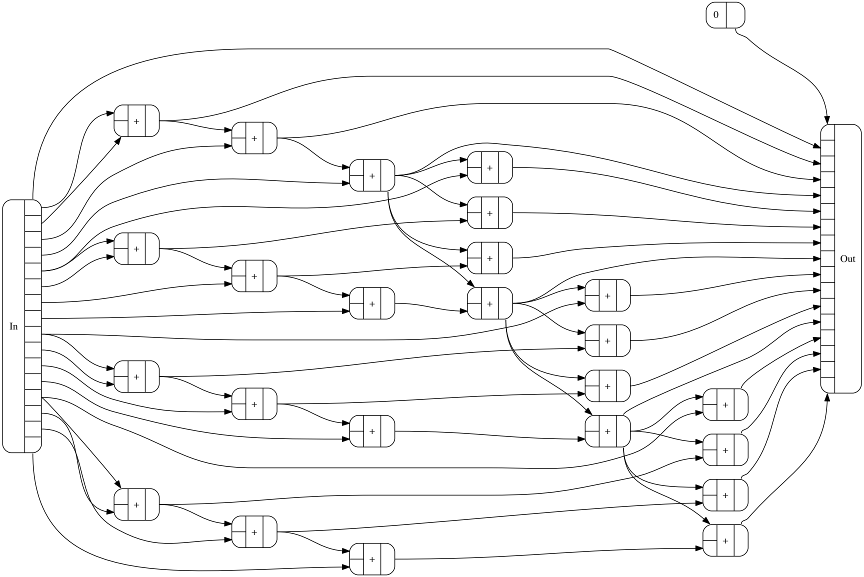
$\overleftarrow{8} \circ 2$

work: 22, depth: 8

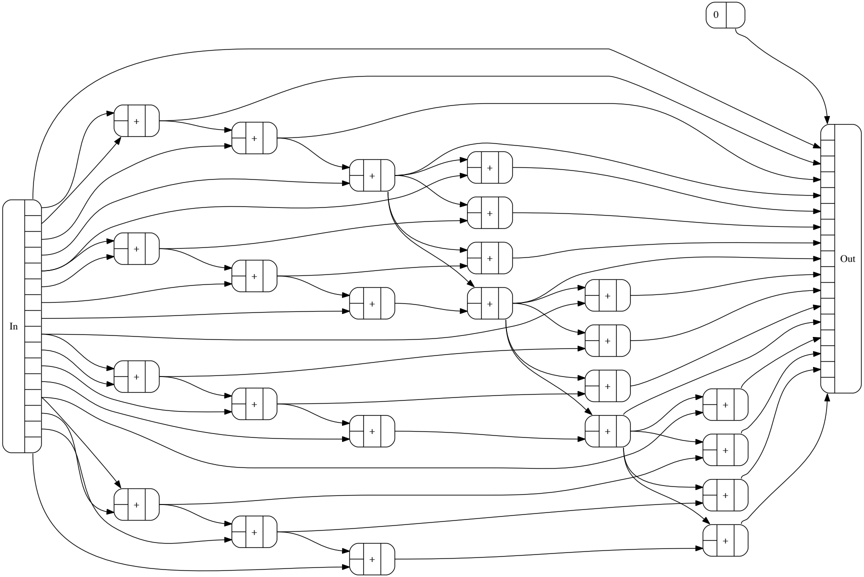


$\overleftarrow{4} \circ \overleftarrow{4}$

work: 24, depth: 6

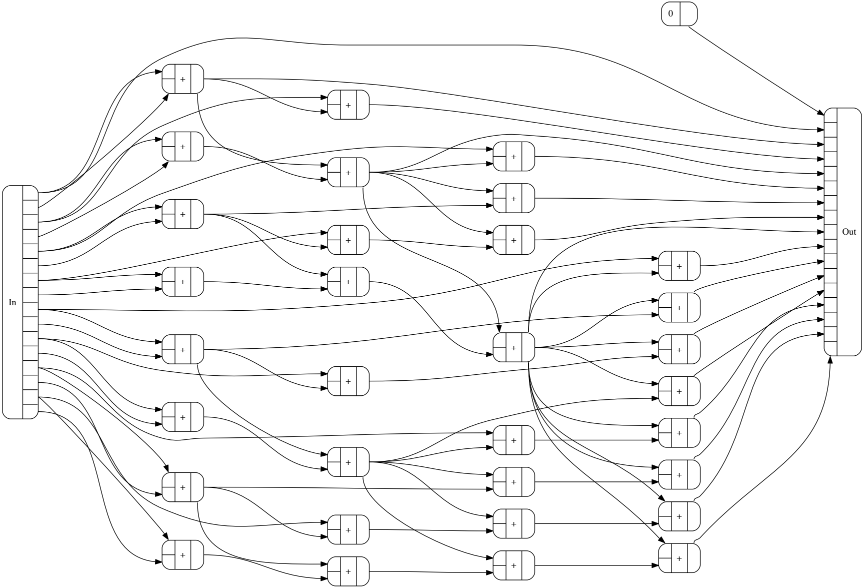


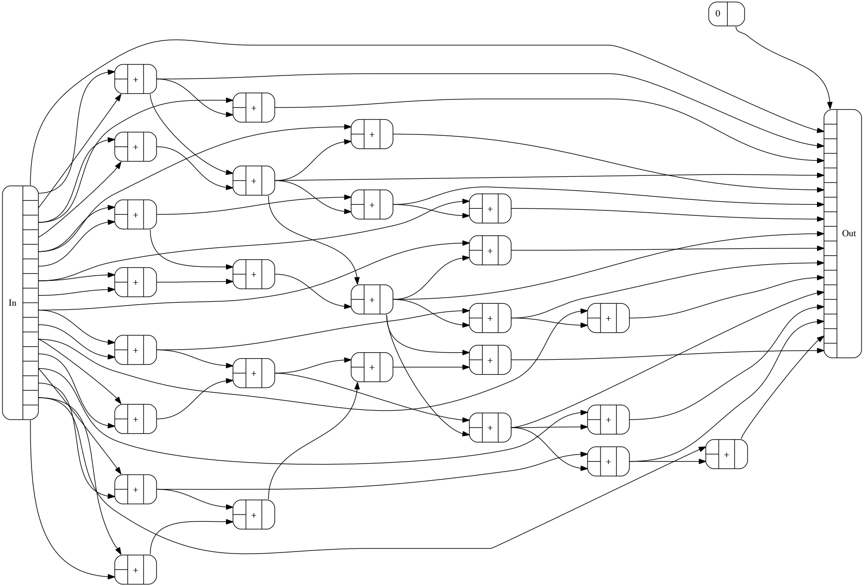
work: 24, depth: 6



$2^{\downarrow 4}$

work: 32, depth: 4





0
In
Out
+

FFT

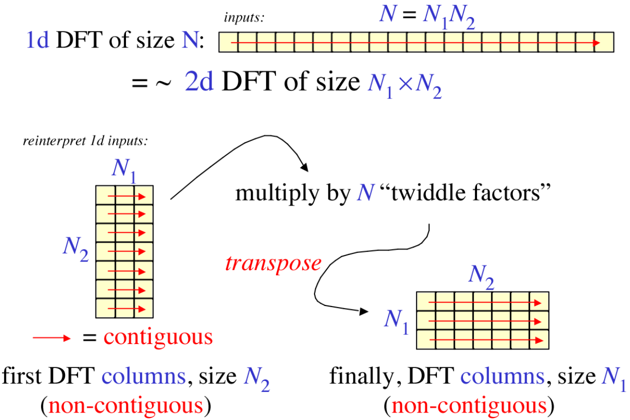
Discrete Fourier Transform (DFT)

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

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

FFT computes DFT in $O(N \log N)$ work.

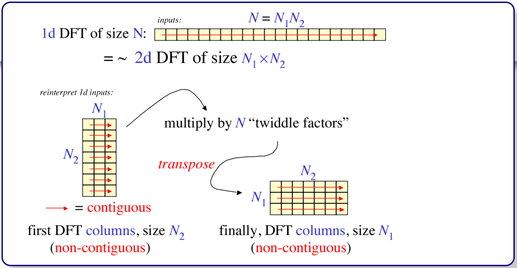
Factoring DFT — pictures



Johnson [2010]

How might we express generically?

Factoring DFT

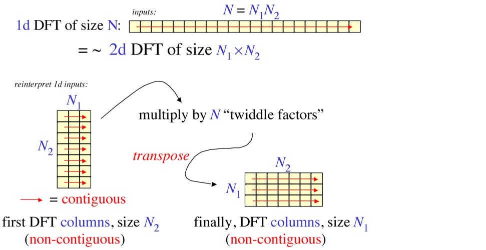


Factor types, not numbers!

$$\textbf{newtype}\ (g \circ f)\ a = O_1\ (g\ (f\ a))$$

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

Factoring DFT



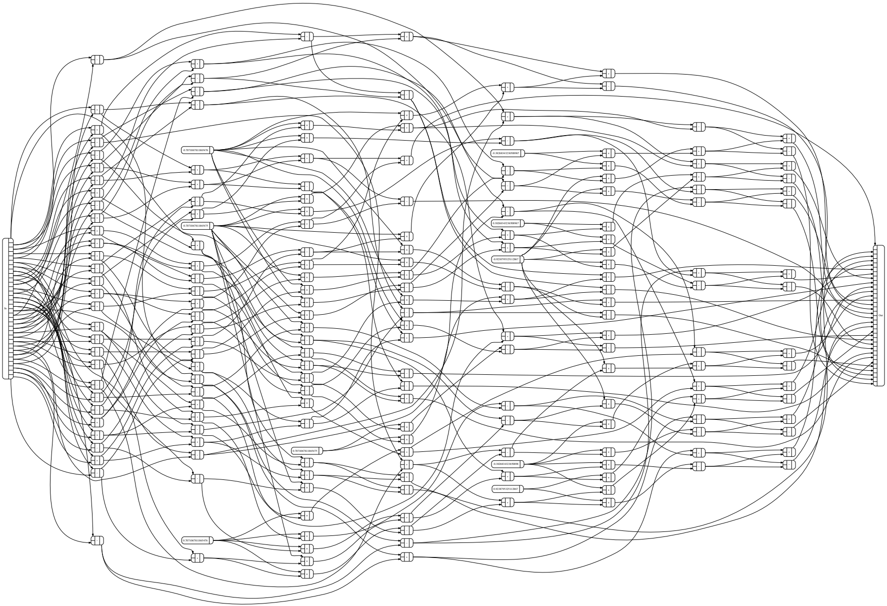
class $\mathit{FFT}\ f$ **where**
  type $\mathit{Reverse}\ f :: * \to *$
  $\mathit{fft} :: f\ \mathbb{C} \to \mathit{Reverse}\ f\ \mathbb{C}$

instance ... $\Rightarrow \mathit{FFT}\ (g \circ f)$ **where**
  type $\mathit{Reverse}\ (g \circ f) = \mathit{Reverse}\ f \circ \mathit{Reverse}\ g$
  $\mathit{fft} = O_1 \circ \mathit{ffts}' \circ \mathit{transpose} \circ \mathit{twiddle} \circ \mathit{ffts}' \circ \mathit{unO}_1$

$\mathit{ffts}' :: ... \Rightarrow g\ (f\ \mathbb{C}) \to \mathit{Reverse}\ g\ (f\ \mathbb{C})$
$\mathit{ffts}' = \mathit{transpose} \circ \mathit{fmap}\ \mathit{fft} \circ \mathit{transpose}$

fft @$2^{\downarrow 4}$

fft @$2^{\uparrow 4}$



More goodies in the paper

- Scan and FFT on 2^{2^n}.
- Log time polynomial evaluation via scan.
- Complexity, generically.
- Additional examples.
- Details.

Conclusions

- Alternative to array programming:
  - Elegantly compositional.
  - Uncluttered by index computations.
  - Safe from out-of-bounds errors.
  - Reveals algorithm essence and connections.
- Four well-known parallel algorithms: $h^{\downarrow n}$, $h^{\uparrow n}$.
- Two possibly new and useful algorithms: 2^{2^n}.