# From Haskell to Hardware via CCCs


Conal Elliott

Tabula


August, 2014

# Tabula

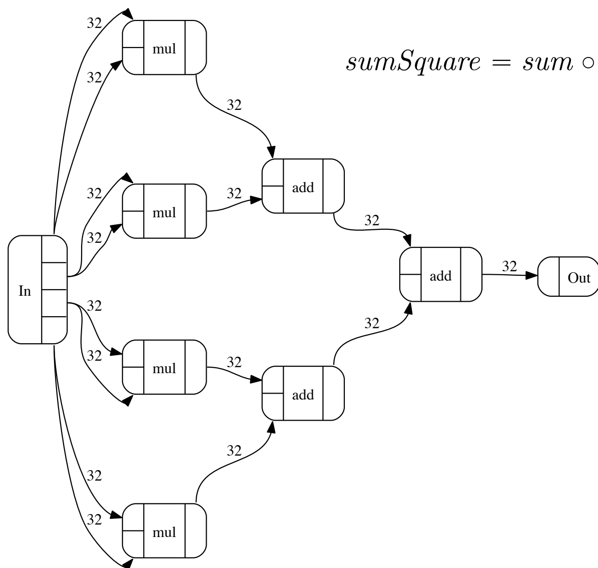
- Founded by Steve Teig about 10 years ago.
- Post-FPGA reconfigurable hardware.
- Spacetime architecture:
  - 3D for shorter paths
  - Implemented by rapid reconfiguration (2GHz)
  - Minkowski spacetime (special relativity)
  - Spacetime layout with causality constraints
  - Very high sustained throughput
- Tremendous flexibility for moving computations in space & time
- Program in a non-sequential language: Haskell
- Compiler developed openly and shared freely

# Example

$$square :: Num\ a \Rightarrow a \rightarrow a$$
$$square\ a = a * a$$

$$sumSquare :: (Functor\ f, Foldable\ f, Num\ a) \Rightarrow f\ a \rightarrow a$$
$$sumSquare = sum \circ fmap\ square$$

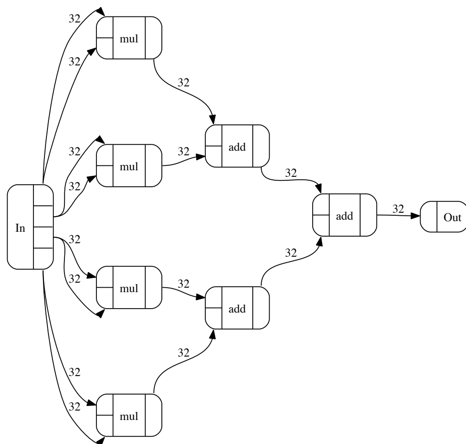
# $sumSquare :: Tree_2\ Int \rightarrow Int$



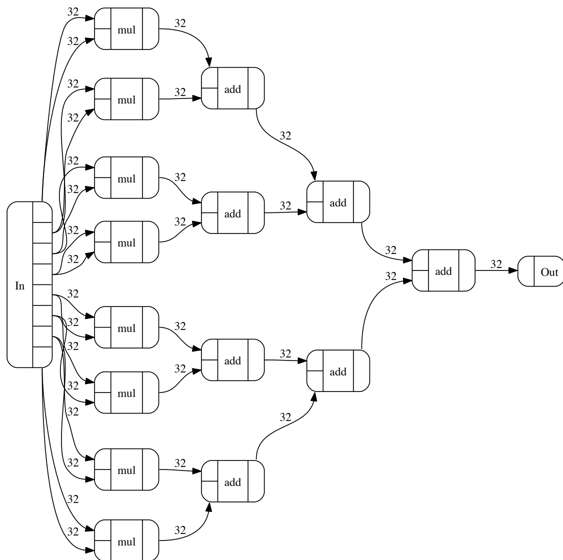

$$sumSquare = sum \circ fmap\ square$$

# $sumSquare :: Tree_2\ Int \rightarrow Int$

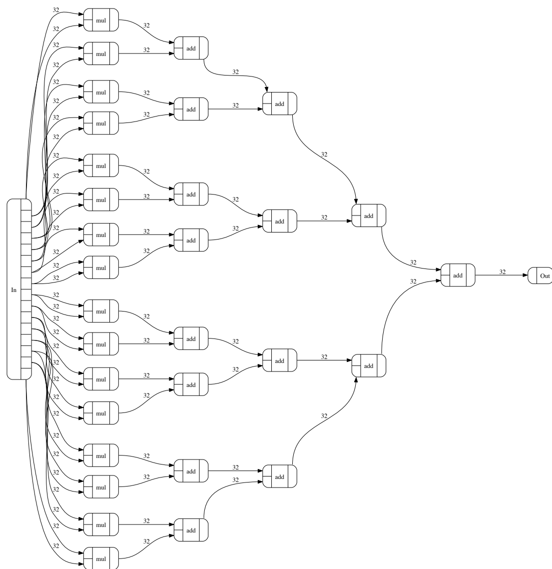
```
module sumSquare-t2 (In_0, In_1, In_2, In_3, Out);
  input [0:31] In_0;
  input [0:31] In_1;
  input [0:31] In_2;
  input [0:31] In_3;
  output [0:31] Out;
  wire [0:31] w_mul_I1;
  wire [0:31] w_mul_I2;
  wire [0:31] w_mul_I3;
  wire [0:31] w_mul_I4;
  wire [0:31] w_add_I5;
  wire [0:31] w_add_I6;
  wire [0:31] w_add_I7;
  assign w_mul_I1 = In_0 * In_0;
  assign w_mul_I2 = In_1 * In_1;
  assign w_mul_I3 = In_2 * In_2;
  assign w_mul_I4 = In_3 * In_3;
  assign w_add_I5 = w_mul_I1 + w_mul_I2;
  assign w_add_I6 = w_mul_I3 + w_mul_I4;
  assign w_add_I7 = w_add_I5 + w_add_I6;
  assign Out = w_add_I7;
endmodule
```



# $sumSquare :: Tree_3\ Int \rightarrow Int$



# $sumSquare :: Tree_4\ Int \rightarrow Int$



# *How it works*

# Overall plan

- Convert Haskell to Core (GHC).
- Monomorphize.
- Convert to abstract vocabulary.
- Interpret as circuits.
- Synthesize & optimize with existing HDL machinery.

Initial simplifications:

- Shape-typed data
- Combinational

# GHC Core

```
data Expr b   -- "b" for the type of binders,
  = Var  Id
  | Lit  Literal
  | App  (Expr b) (Expr b)
  | Lam  b (Expr b)
  | Let  (Bind b) (Expr b)
  | Case (Expr b) b Type [Alt b]
  | Cast (Expr b) Coercion
  | Type Type

type Alt b = (AltCon, [b], Expr b)

data AltCon = DataAlt DataCon | LitAlt Literal | DEFAULT

data Bind b = NonRec b (Expr b) | Rec [(b, Expr b)]
```

# Overloading lambda

Powerful abstraction mechanisms:

- Lambda/application
- Type classes

Can we use type classes to generalize lambda & application?

# (Bi-)Cartesian closed categories

- *Category*: identity and composition
- *Cartesian*: products
- *Co-Cartesian*: coproducts ("sums")
- *Closed*: exponentials (arrows as "values")

Suffices for translating typed lambda calculus (J. Lambek, 1980).

# Category

Interface:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Category}\ (\leadsto)\ \textbf{where} \\
\quad \mathit{id}\ \ :: a \leadsto a \\
\quad (\circ) :: (b \leadsto c) \to (a \leadsto b) \to (a \leadsto c)
\end{array}
$$

Laws:

$$
\begin{array}{ll}
\mathit{id} \circ f & \equiv f \\
g \circ \mathit{id} & \equiv g \\
(h \circ g) \circ f & \equiv h \circ (g \circ f)
\end{array}
$$

# Products

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ (\leadsto) \Rightarrow \mathit{ProductCat}\ (\leadsto)\ \textbf{where} \\
\quad \textbf{type } a \times_{\leadsto} b \\
\quad \mathit{exl} \quad :: (a \times_{\leadsto} b) \leadsto a \\
\quad \mathit{exr} \quad :: (a \times_{\leadsto} b) \leadsto b \\
\quad (\vartriangle) \quad :: (a \leadsto c) \to (a \leadsto d) \to (a \leadsto (c \times_{\leadsto} d)) \\
\quad (\times) \quad :: (a \leadsto c) \to (b \leadsto d) \to ((a \times_{\leadsto} b) \leadsto (c \times_{\leadsto} d)) \\
\quad f \times g = (f \circ \mathit{exl}) \vartriangle (g \circ \mathit{exr})
\end{array}
$$

Laws:

$$
\begin{array}{ll}
\mathit{exl} \circ (f \vartriangle g) & \equiv f \\
\mathit{exr} \circ (f \vartriangle g) & \equiv g \\
\mathit{exl} \circ h \vartriangle \mathit{exr} \circ h & \equiv h \\
\mathit{exl} \vartriangle \mathit{exr} & \equiv \mathit{id}
\end{array}
$$

$$
\begin{array}{ll}
(f \times g) \circ (h \vartriangle k) & \equiv (f \circ h) \vartriangle (g \circ k) \\
\mathit{id} \times \mathit{id} & \equiv \mathit{id} \\
(f \times g) \circ (h \times k) & \equiv (f \circ h) \times (g \circ k) \\
(f \vartriangle g) \circ h & \equiv (f \circ h) \vartriangle (g \circ h)
\end{array}
$$

# Coproducts

$$
\begin{array}{l}
\textbf{class}\ \mathit{Category}\ (\leadsto) \Rightarrow \mathit{CoproductCat}\ (\leadsto)\ \textbf{where} \\
\quad \textbf{type}\ a +_{\leadsto} b \\
\quad \mathit{inl} \quad :: a \leadsto (a +_{\leadsto} b) \\
\quad \mathit{inr} \quad :: b \leadsto (a +_{\leadsto} b) \\
\quad (\triangledown) \quad :: (a \leadsto c) \to (b \leadsto c) \to ((a +_{\leadsto} b) \leadsto c) \\
\quad (+) \quad :: (a \leadsto c) \to (b \leadsto d) \to ((a +_{\leadsto} b) \leadsto (c +_{\leadsto} d)) \\
\quad f + g = (\mathit{inl} \circ f) \triangledown (\mathit{inr} \circ g)
\end{array}
$$

Laws (dual to product):

$$
\begin{array}{ll}
(f \triangledown g) \circ \mathit{inl} & \equiv f \\
(f \triangledown g) \circ \mathit{inr} & \equiv g \\
h \circ \mathit{inl} \triangledown h \circ \mathit{inr} & \equiv h \\
\mathit{inl} \triangledown \mathit{inr} & \equiv \mathit{id}
\end{array}
$$

$$
\begin{array}{ll}
(h \triangledown k) \circ (f + g) & \equiv (h \circ f) \triangledown (k \circ g) \\
\mathit{id} + \mathit{id} & \equiv \mathit{id} \\
(h + k) \circ (f + g) & \equiv (h \circ f) + (k \circ g) \\
h \circ (f \triangledown g) & \equiv (h \circ f) \triangledown (h \circ g)
\end{array}
$$

# Exponentials

$$
\begin{array}{l}
\textbf{class } \mathit{ProductCat}\ (\leadsto) \Rightarrow \mathit{ClosedCat}\ (\leadsto)\ \textbf{where} \\
\quad \textbf{type } a \Rightarrow_{\leadsto} b \\
\quad \mathit{apply} \quad\ :: ((a \Rightarrow_{\leadsto} b) \times_{\leadsto} a) \leadsto b \\
\quad \mathit{curry} \quad\ :: ((a \times_{\leadsto} b) \leadsto c) \to (a \leadsto (b \Rightarrow_{\leadsto} c)) \\
\quad \mathit{uncurry} :: (a \leadsto (b \Rightarrow_{\leadsto} c)) \to (a \times_{\leadsto} b) \leadsto c
\end{array}
$$

# Lambda terms

$$
\begin{aligned}
&\textbf{data}\ E :: * \to *\ \ \textbf{where}\\
&\quad \mathit{Var} \ \ \ :: V\ a \to E\ a\\
&\quad \mathit{Const} :: \mathit{Prim}\ a \to E\ a\\
&\quad \mathit{App} \ \ :: E\ (a \to b) \to E\ a \to E\ b\\
&\quad \mathit{Lam} \ \ :: \mathit{Pat}\ a \to E\ b \to E\ (a \to b)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ \mathit{Pat} :: * \to *\ \ \textbf{where}\\
&\quad \mathit{UnitPat} :: \mathit{Pat}\ ()\\
&\quad \mathit{VarPat} \ :: V\ a \to \mathit{Pat}\ a\\
&\quad \mathit{PairPat} :: \mathit{Pat}\ a \to \mathit{Pat}\ b \to \mathit{Pat}\ (a \times b)
\end{aligned}
$$

# Lambda to CCC

$$
\begin{array}{lll}
(\lambda p \rightarrow k) & \dashrightarrow \textit{const}\ k & \text{-- v not free in k} \\
(\lambda p \rightarrow v) & \dashrightarrow \ldots & \text{-- accessor} \\
(\lambda p \rightarrow u\ v) & \dashrightarrow \textit{apply} \circ ((\lambda p \rightarrow u) \vartriangle (\lambda p \rightarrow v)) & \\
(\lambda p \rightarrow \lambda q \rightarrow u) & \dashrightarrow \textit{curry}\ (\lambda(p, q) \rightarrow u) &
\end{array}
$$

# Lambda to CCC

$$
\begin{array}{lll}
(\lambda p \to k) & \dashrightarrow \mathit{const}\ k & \text{-- v not free in k} \\
(\lambda p \to v) & \dashrightarrow \ldots & \text{-- accessor} \\
(\lambda p \to u\ v) & \dashrightarrow \mathit{apply} \circ ((\lambda p \to u) \vartriangle (\lambda p \to v)) & \\
(\lambda p \to \lambda q \to u) & \dashrightarrow \mathit{curry}\ (\lambda (p, q) \to u) &
\end{array}
$$

$$
\begin{array}{l}
\mathit{convert} :: \mathit{CCC}\ (\leadsto) \Rightarrow \mathit{Pat}\ a \to E\ b \to (a \leadsto b) \\
\mathit{convert}\ _\ (\mathit{Const}\ x) = \mathit{constArrow}\ x \\
\mathit{convert}\ p\ (\mathit{Var}\ v) \quad = \mathit{convertVar}\ p\ v \\
\mathit{convert}\ p\ (\mathit{App}\ u\ v) = \mathit{apply} \circ (\mathit{convert}\ p\ u \vartriangle \mathit{convert}\ p\ v) \\
\mathit{convert}\ p\ (\mathit{Lam}\ q\ e) = \mathit{curry}\ (\mathit{convert}\ (\mathit{PairPat}\ p\ q)\ e)
\end{array}
$$

# Circuit CCC

$$\mathbf{data}\ Comp = \forall a\ b. Comp\ (Prim\ a\ b)\ (Buses\ a)\ (Buses\ b)$$

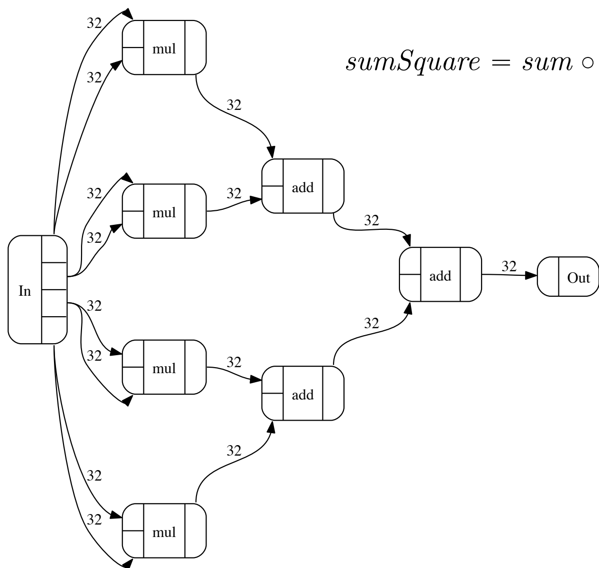
$$\mathbf{type}\ CircuitM = WriterT\ (Seq\ Comp)\ (State\ BusSupply)$$

$$\mathbf{newtype}\ a \rightarrowtail b = Circ\ (Buses\ a \rightarrow CircuitM\ (Buses\ b))$$

$$\mathbf{instance}\ Category\ (\rightarrowtail)\ \mathbf{where} \ldots$$
$$\mathbf{instance}\ ProductCat\ (\rightarrowtail)\ \mathbf{where} \ldots$$
$$\mathbf{instance}\ ClosedCat\ (\rightarrowtail)\ \mathbf{where} \ldots$$

# *Examples*

# $sumSquare :: Tree_2\ Int \rightarrow Int$

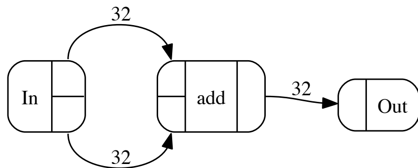



$$sumSquare = sum \circ fmap\ square$$

# $sumSquare :: Tree_2\ Int \to Int$

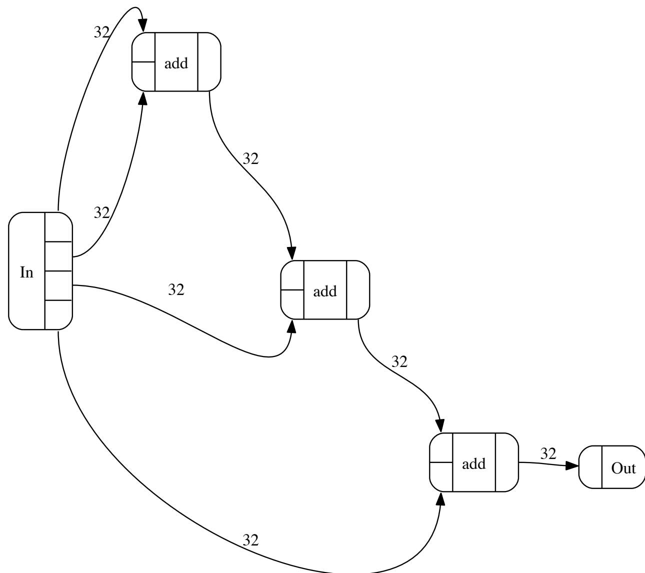
```
uncurry (apply . (curry (apply . (curry (apply . (curry (apply . (curry (curry
(apply . (curry (apply . (apply . (curry (curry (uncurry add) . exr) . it &&&
apply . (curry (repr . exr) . it &&& apply . (((((id . exr) . exl) . exl) . exl)
. id *** (id . exl) . id))) &&& apply . (curry (repr . exr) . it &&& apply .
(((((id . exr) . exl) . exl) . exl) . id *** (id . exr) . id)))) &&& apply .
(curry (repr . exr) . it &&& apply . (curry (apply . (curry (abst . exr) . it
&&& apply . (apply . (curry (curry id . exr) . it &&& apply . (((id . exr) .
exl) . id *** (id . exl) . id)) &&& apply . (((id . exr) . exl) . id *** (id .
exr) . id)))) &&& apply . (curry (repr . exr) . it &&& apply . (curry (repr .
exr) . it &&& id . exr))))))) &&& curry (apply . (curry (abst . exr) . it &&&
apply . (curry (apply . (curry (abst . exr) . it &&& apply . (apply . (curry
(curry id . exr) . it &&& apply . (((id . exr) . exl) . id *** (id . exl) . id))
&&& apply . (((id . exr) . exl) . id *** (id . exr) . id)))) &&& apply . (curry
(repr . exr) . it &&& apply . (curry (repr . exr) . it &&& id . exr))))))) &&&
curry (apply . (curry (apply . (curry (abst . exr) . it &&& apply . (apply .
(curry (curry (uncurry mul) . exr) . it &&& id . exr) &&& id . exr))) &&& apply
. (curry (repr . exr) . it &&& id . exr))))) &&& curry (apply . (curry (apply .
(curry (abst . exr) . it &&& apply . (apply . (curry (curry (uncurry add) . exr)
. it &&& apply . (curry (repr . exr) . it &&& apply . (((id . exr) . exl) . id
*** (id . exl) . id))) &&& apply . (curry (repr . exr) . it &&& apply . (((id .
exr) . exl) . id *** (id . exr) . id))))) &&& apply . (curry (repr . exr) . it
&&& apply . (curry (repr . exr) . it &&& id . exr)))))) &&& curry (apply .
(curry (abst . exr) . it &&& apply . (curry (repr . exr) . it &&& id . exr)))))
. (it &&& id)
```

# $\lambda(a, b) \rightarrow a + b :: Int$

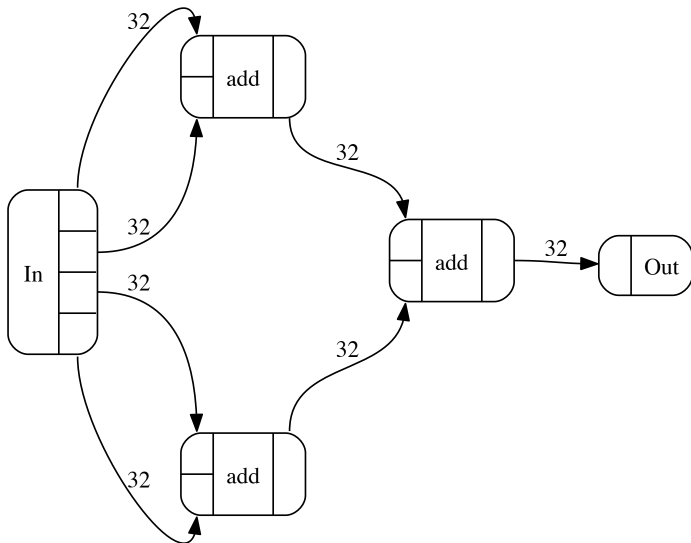



```
module sum-2 (In_0, In_1, Out);
  input [0:31] In_0;
  input [0:31] In_1;
  output [0:31] Out;
  wire [0:31] w_add_I1;
  assign w_add_I1 = In_0 + In_1;
  assign Out = w_add_I1;
endmodule
```

# $\lambda(a, b, c, d) \rightarrow a + b + c + d :: Int$



# $\lambda(a, b, c, d) \rightarrow (a + b) + (c + d) :: Int$

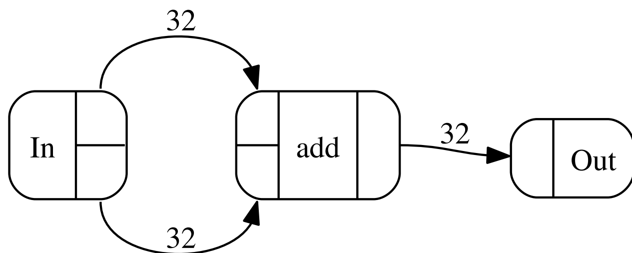


# Uniform pairs

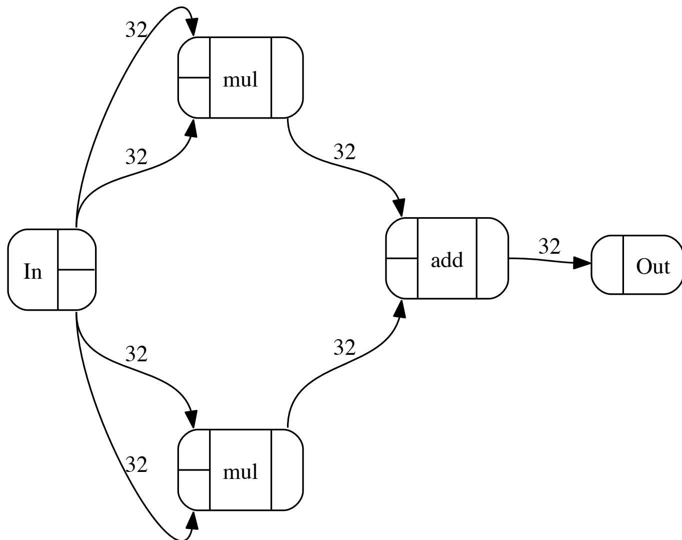
$\textbf{data}\ \mathit{Pair}\ a = a :\!\#\ a$

*Functor*, *Applicative*, *Monad*, *Foldable*, *Traversable* (*transpose*).

# $sum :: Pair\ Int \rightarrow Int$



## $sumSquare :: Pair\ Int \rightarrow Int$



# Length-typed vectors

$$
\begin{aligned}
&\textbf{data } \mathit{Vec} :: \mathit{Nat} \to * \to * \ \textbf{ where} \\
&\quad \mathit{ZVec} :: \mathit{Vec}_0\ a \\
&\quad (\lhd) \quad :: a \to \mathit{Vec}_n\ a \to \mathit{Vec}_{1+n}\ a
\end{aligned}
$$

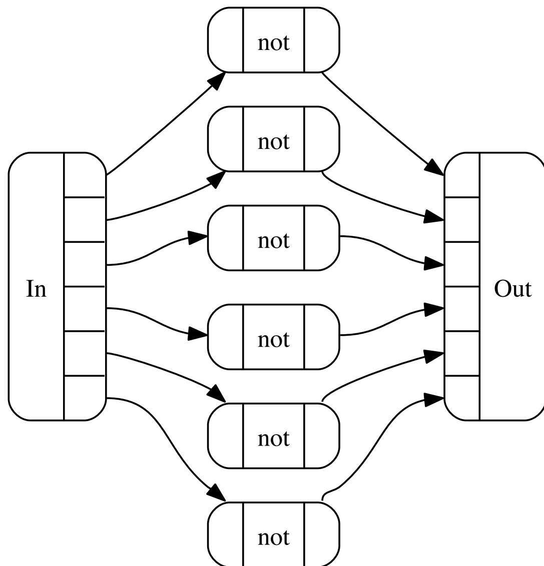
# Length-typed vectors

$$
\begin{array}{l}
\textbf{data}\ \mathit{Vec} :: \mathit{Nat} \to * \to *\ \ \textbf{where} \\
\quad \mathit{ZVec} :: \mathit{Vec}_0\ a \\
\quad (\lhd)\ \ :: a \to \mathit{Vec}_n\ a \to \mathit{Vec}_{1+n}\ a
\end{array}
$$

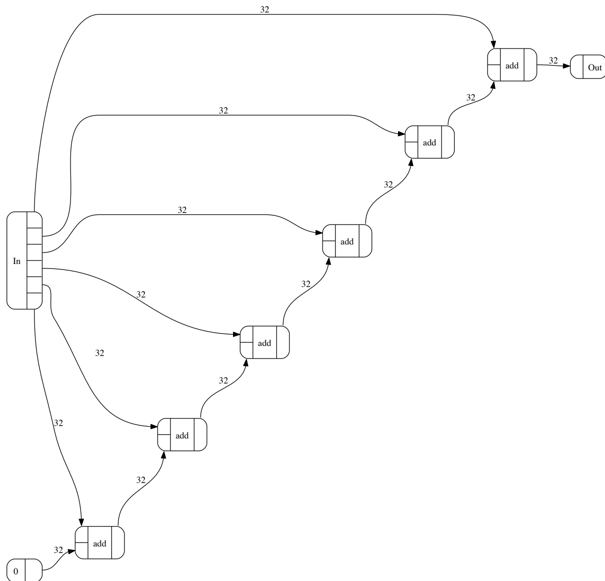
$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ (\mathit{Vec}_n)\ \textbf{where} \\
\quad \mathit{fmap}\ _\ \mathit{ZVec} \quad\ = \mathit{ZVec} \\
\quad \mathit{fmap}\ f\ (a \lhd u) = f\ a \lhd \mathit{fmap}\ f\ u \\
\textbf{instance}\ \mathit{Foldable}\ (\mathit{Vec}_n)\ \textbf{where} \\
\quad \mathit{foldMap}\ _\ \mathit{ZVec} \quad\ \ = \varepsilon \\
\quad \mathit{foldMap}\ h\ (a \lhd as) = h\ a \oplus \mathit{foldMap}\ h\ as
\end{array}
$$

And *Applicative*, *Monad*, *Traversable*.

# $fmap\ not :: Vec_6\ Bool \to Vec_6\ Bool$



# $sum :: Vec_6\ Int \rightarrow Int$



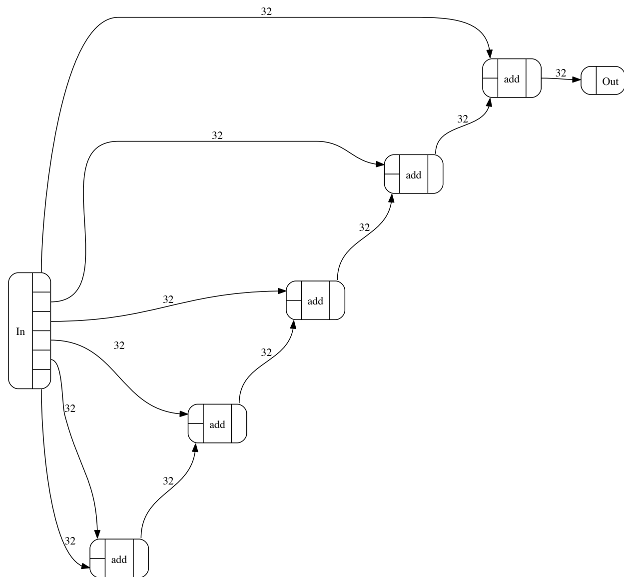
# Improved folding

Bypass $\varepsilon$ when possible.

$$
\begin{array}{l}
\textbf{instance } \mathit{Foldable}\ (\mathit{Vec}_n)\ \textbf{where} \\
\quad \mathit{foldMap}\ _\ \mathit{ZVec} \qquad\quad\ = \varepsilon \\
\quad \mathit{foldMap}\ h\ \mathit{as}@(_ \lhd _) = \mathit{foldMapS}\ h\ \mathit{as}
\end{array}
$$

$$
\begin{array}{l}
\mathit{foldMapS} :: \mathit{Monoid}\ m \Rightarrow (a \rightarrow m) \rightarrow \mathit{Vec}_{1+n}\ a \rightarrow m \\
\mathit{foldMapS}\ h\ (a \lhd \mathit{as}) = \\
\quad \textbf{case } \mathit{as} \textbf{ of} \\
\qquad \mathit{ZVec} \quad\ \rightarrow h\ a \\
\qquad (_ \lhd _) \rightarrow h\ a \oplus \mathit{foldMapS}\ h\ \mathit{as}
\end{array}
$$

# $sum :: Vec_6\ Int \rightarrow Int$



# Depth-typed trees

$$\textbf{data}\ \mathit{Tree} :: \mathit{Nat} \rightarrow * \rightarrow *\ \ \textbf{where}$$
$$L :: a \rightarrow \mathit{Tree}_0\ a$$
$$B :: \mathit{Pair}\ (\mathit{Tree}_n\ a) \rightarrow \mathit{Tree}_{1+n}\ a$$

# Depth-typed trees

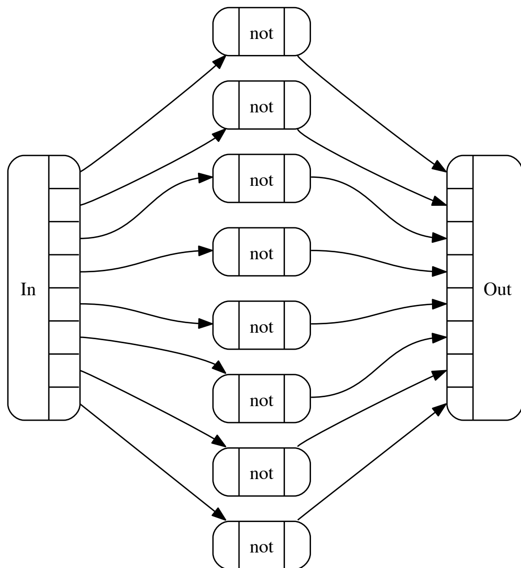
$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree} :: \mathit{Nat} \rightarrow * \rightarrow *\ \ \textbf{where}\\
&\quad L :: a \rightarrow \mathit{Tree}_0\ a\\
&\quad B :: \mathit{Pair}\ (\mathit{Tree}_n\ a) \rightarrow \mathit{Tree}_{1+n}\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ (\mathit{Tree}_n)\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (L\ a\ \ ) = L\ (f\ \ a)\\
&\quad \mathit{fmap}\ f\ (B\ \mathit{ts}) = B\ ((\mathit{fmap} \circ \mathit{fmap})\ f\ \ \mathit{ts})\\
&\textbf{instance}\ \mathit{Foldable}\ (\mathit{Tree}_n)\ \textbf{where}\\
&\quad \mathit{foldMap}\ f\ (L\ a\ \ ) = f\ \ a\\
&\quad \mathit{foldMap}\ f\ (B\ \mathit{ts}) = (\mathit{foldMap} \circ \mathit{foldMap})\ f\ \ \mathit{ts}
\end{aligned}
$$

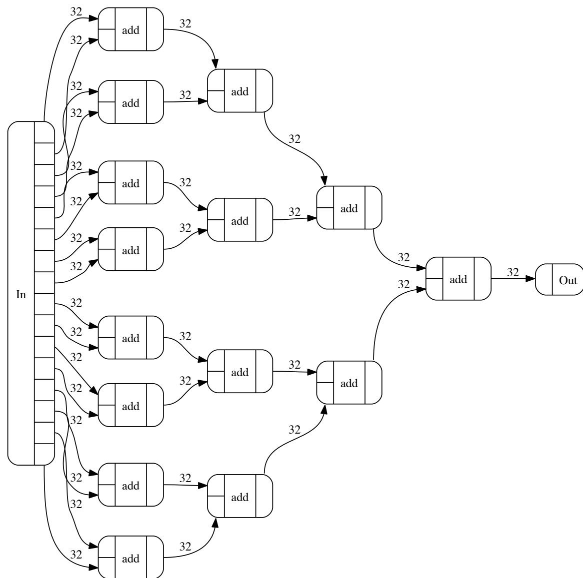
And *Applicative*, *Monad*, *Traversable*.

Easily generalize beyond *Pair*.

# $fmap\ not :: Tree_3\ Bool \rightarrow Tree_3\ Bool$



# $sum :: Tree_4\ Int \rightarrow Int$

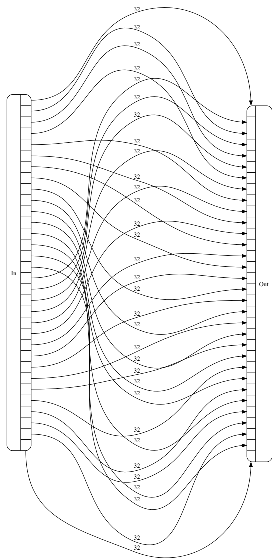


# $sum :: Tree_3\ Int \rightarrow Int$

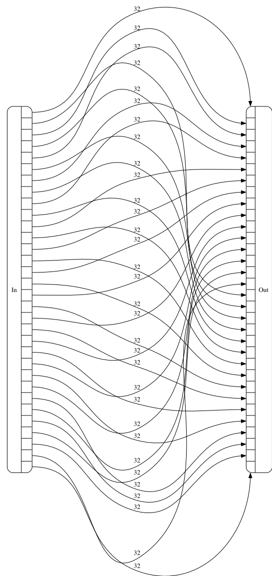
Monomorphized & simplified GHC Core:

```
let f0 :: Tree0 Int → Sum Int
    f0 = λds →
      abst $fRepSum (repr $fRepTree0 ds)
    f1 :: Tree1 Int → Sum Int
    f1 = λds →
      case repr $fRepPair (repr $fRepTree ds) of
        (, ) a b →
          abst $fRepSum
              ($fNumInt+ (repr $fRepSum (f0 a))
                         (repr $fRepSum (f0 b)))
    f2 :: Tree2 Int → Sum Int
    f2 = λds →
      case repr $fRepPair (repr $fRepTree ds) of
        (, ) a b →
          abst $fRepSum
              ($fNumInt+ (repr $fRepSum (f1 a))
                         (repr $fRepSum (f1 b)))
in λη →
   repr $fRepSum
         (case repr $fRepPair (repr $fRepTree η) of
            (, ) a b →
              abst $fRepSum
                ($fNumInt+ (repr $fRepSum (f2 a))
                           (repr $fRepSum (f2 b))))
```

# $transpose :: Pair\ (Tree_4\ Int) \rightarrow Tree_4\ (Pair\ Int)$



# $transpose :: Tree_4\ (Pair\ Int) \rightarrow Pair\ (Tree_4\ Int)$



# Dot products

$$
\begin{aligned}
&dot :: (Foldable\ g, Foldable\ f, Num\ (f\ \ a), Num\ a) \Rightarrow \\
&\qquad g\ (f\ \ a) \rightarrow a \\
&dot = sum \circ product
\end{aligned}
$$

Typically, $g \equiv Pair$.

# Dot products

$$
\begin{aligned}
&dot :: (Foldable\ g, Foldable\ f, Num\ (f\ \ a), Num\ a) \Rightarrow \\
&\qquad g\ (f\ \ a) \rightarrow a \\
&dot = sum \circ product
\end{aligned}
$$

Typically, $g \equiv Pair$.

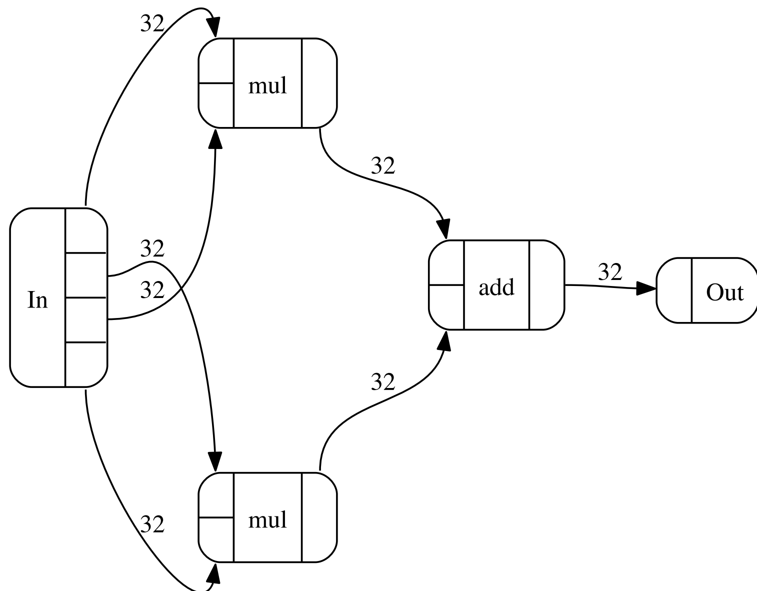
Alternatively,

$$
\begin{aligned}
&dot :: (Traversable\ g, Foldable\ f, Applicative\ f, Num\ a) \Rightarrow \\
&\qquad g\ (f\ \ a) \rightarrow a \\
&dot = sum \circ fmap\ product \circ transpose
\end{aligned}
$$

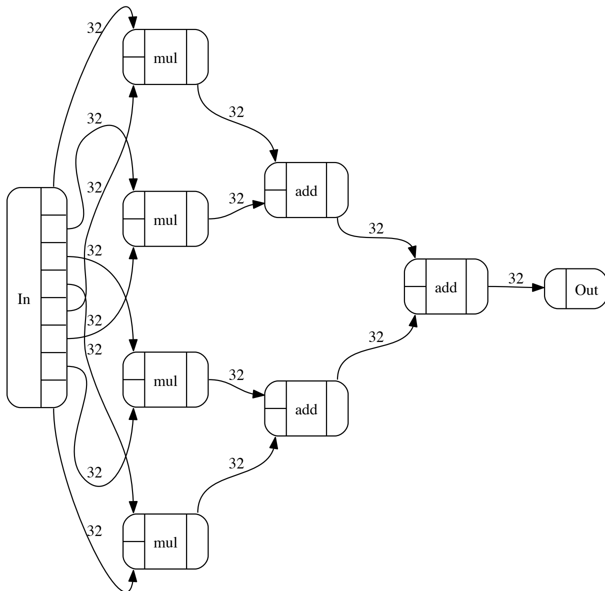
where

$$transpose :: (Traversable\ t, Applicative\ f) \Rightarrow t\ (f\ \ a) \rightarrow f\ (t\ \ a)$$

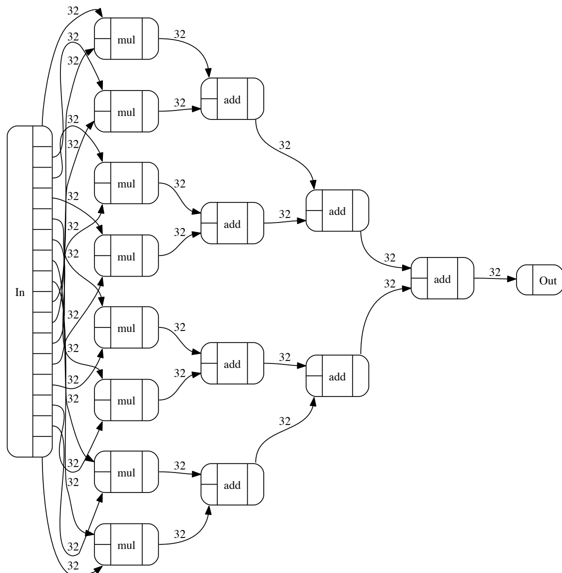
# $dot :: Pair\ (Tree_1\ Int) \to Int$



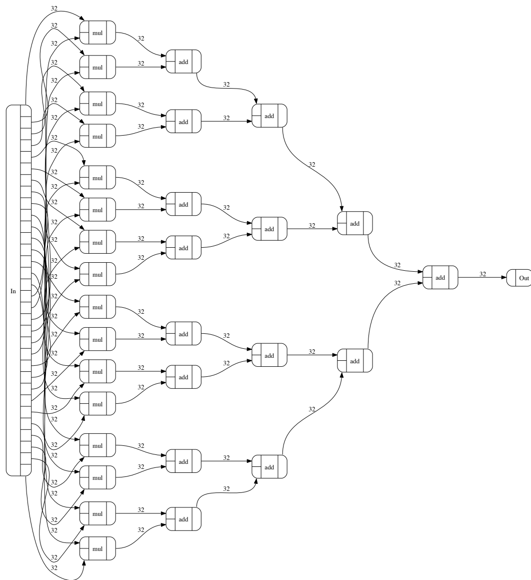
# $dot :: Pair\ (Tree_2\ Int) \rightarrow Int$



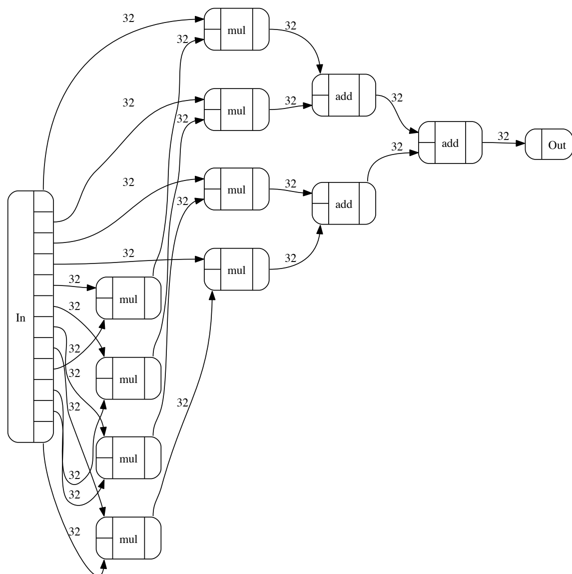
# $dot :: Pair\ (Tree_3\ Int) \rightarrow Int$



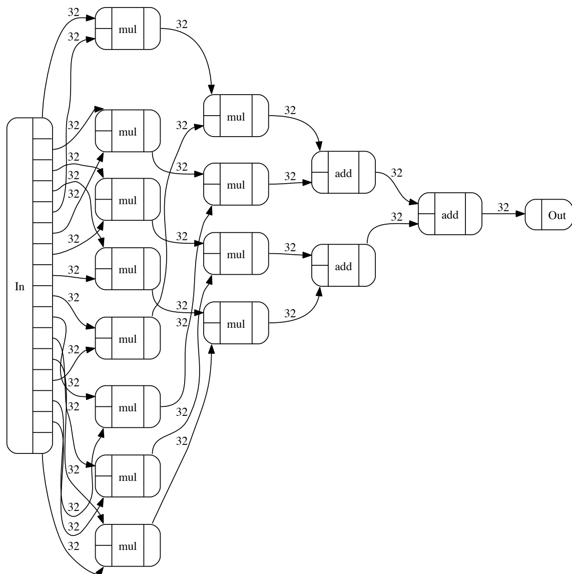
# $dot :: Pair\ (Tree_4\ Int) \rightarrow Int$



# $dot :: Vec_3\ (Tree_2\ Int) \rightarrow Int$



# $dot :: Tree_2\ (Tree_2\ Int) \to Int$



# Linear transformations

$$(\cdot) :: (Foldable\ f, Num\ (f\ a), Num\ a) \Rightarrow f\ a \rightarrow f\ a \rightarrow a$$
$$u \cdot v = dot\ (u :\# \ v)$$

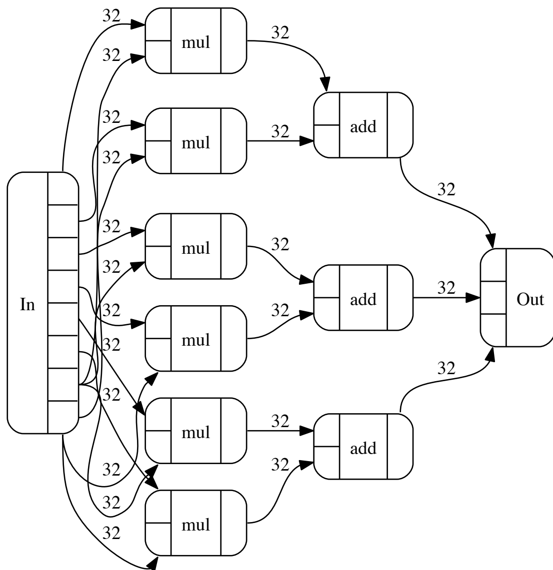
# Linear transformations

$$(\cdot) :: (\mathit{Foldable}\ f, \mathit{Num}\ (f\ a), \mathit{Num}\ a) \Rightarrow f\ a \to f\ a \to a$$
$$u \cdot v = \mathit{dot}\ (u :\#\ v)$$

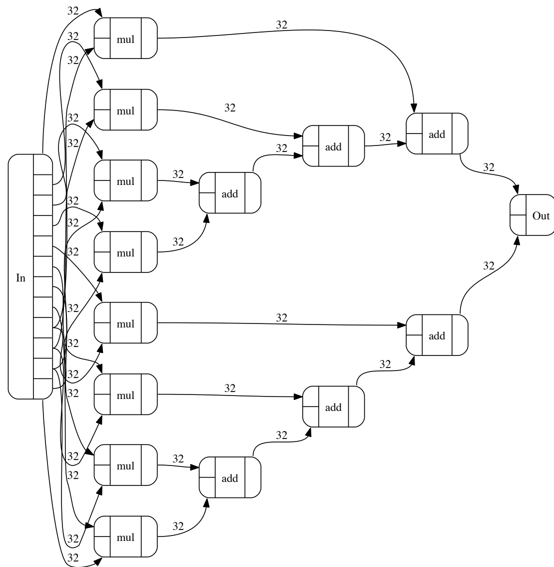
$$\textbf{type}\ \mathit{Mat}_{m,n}\ a = \mathit{Vec}_n\ (\mathit{Vec}_m\ a)$$

$$(\hat{\$}) :: (\mathit{IsNat}\ m, \mathit{Num}\ a) \Rightarrow \mathit{Mat}_{m,n}\ a \to \mathit{Vec}_m\ a \to \mathit{Vec}_n\ a$$
$$\mathit{mat}\ \hat{\$}\ \mathit{vec} = (\cdot \mathit{vec}) <\$> \mathit{mat}$$

# $(\hat{\$}) :: Mat_{2,3}\ Int \rightarrow Vec_2\ Int \rightarrow Vec_3\ Int$



# $(\hat{\$}) :: \mathit{Mat}_{4,2}\ \mathit{Int} \rightarrow \mathit{Vec}_4\ \mathit{Int} \rightarrow \mathit{Vec}_2\ \mathit{Int}$



# *Generalizing linear transformations*

$$
\begin{aligned}
&(\hat{\$}) :: (IsNat\ m, Num\ a) \Rightarrow \\
&\qquad Vec_n\ (Vec_m\ a) \rightarrow Vec_m\ a \rightarrow Vec_n\ a \\
&mat\ \hat{\$}\ vec = (\cdot vec) \mathbin{<\$>} mat
\end{aligned}
$$

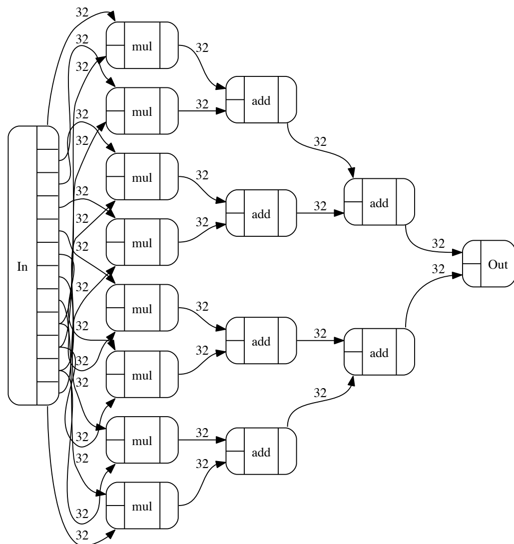
More simply and generally,

$$
\begin{aligned}
&(\hat{\$}) :: (Foldable\ m, Applicative\ m, Functor\ n, Num\ a) \Rightarrow \\
&\qquad n\ (m\ a) \rightarrow m\ a \rightarrow n\ a
\end{aligned}
$$

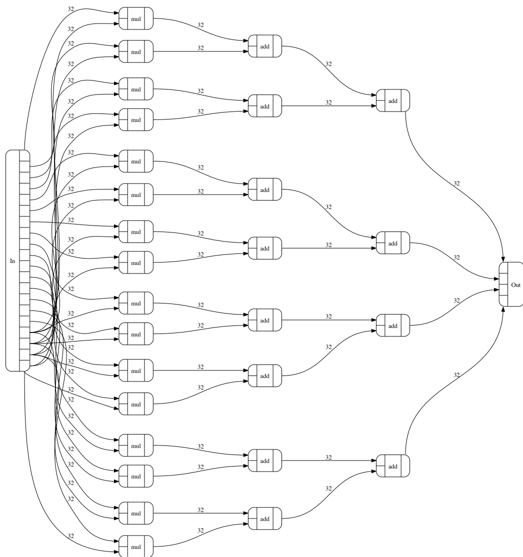
For instance,

$$\mathbf{type}\ MatT_{m,n}\ a = Tree_n\ (Tree_m\ a)$$

# $(\hat{\$}) :: \mathit{MatT}_{2,1}\ \mathit{Int} \to \mathit{Tree}_2\ \mathit{Int} \to \mathit{Tree}_1\ \mathit{Int}$



# $(\hat{\$}) :: MatT_{2,2}\ Int \to Tree_2\ Int \to Tree_2\ Int$



# Composing linear transformations

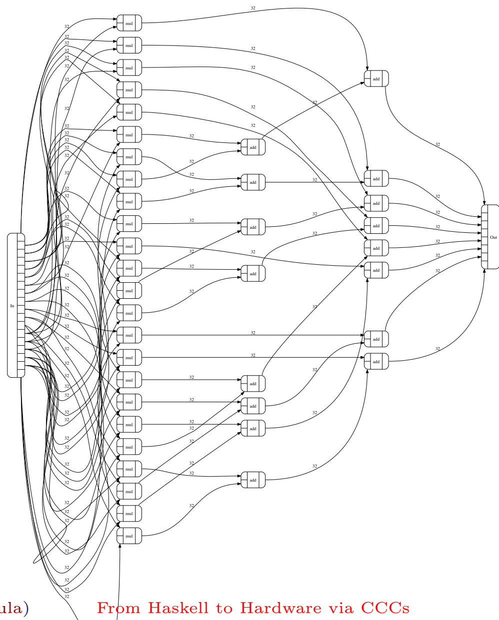
Transform columns:

$$
\begin{aligned}
&(\hat{\circ}) :: (\mathit{IsNat}\ m, \mathit{IsNat}\ n, \mathit{IsNat}\ o, \mathit{Num}\ a) \Rightarrow \\
&\qquad \mathit{Mat}_{n,o}\ a \rightarrow \mathit{Mat}_{m,n}\ a \rightarrow \mathit{Mat}_{m,o}\ a \\
&no\ \hat{\circ}\ mn = \mathit{transpose}\ ((no\ \hat{\$}) \mathbin{<\$>} \mathit{transpose}\ mn)
\end{aligned}
$$

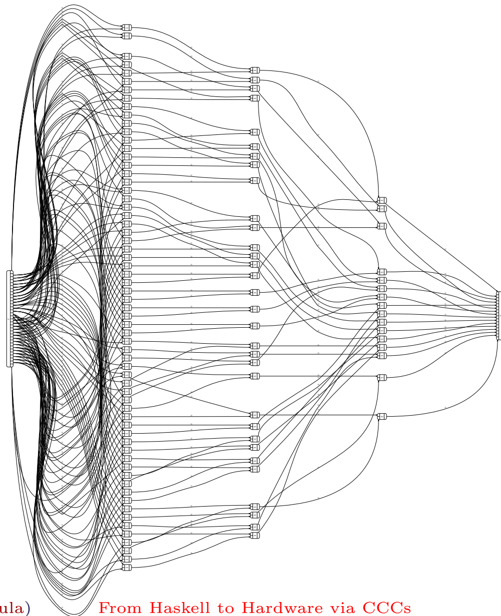
More simply and generally,

$$
\begin{aligned}
&(\hat{\circ}) :: (\mathit{Applicative}\ o, \mathit{Traversable}\ n, \mathit{Applicative}\ n \\
&\qquad , \mathit{Traversable}\ m, \mathit{Applicative}\ m, \mathit{Num}\ a) \Rightarrow \\
&\qquad o\ (n\ a) \rightarrow n\ (m\ a) \rightarrow o\ (m\ a)
\end{aligned}
$$

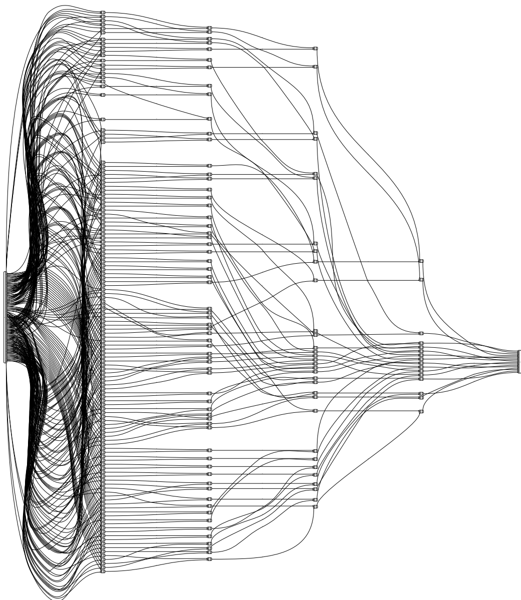
# $(\hat{\circ}) :: \mathit{Mat}_{3,4}\ \mathit{Int} \rightarrow \mathit{Mat}_{2,3}\ \mathit{Int} \rightarrow \mathit{Mat}_{2,4}\ \mathit{Int}$



# $(\hat{\circ}) :: MatT_{2,2}\ Int \rightarrow MatT_{2,2}\ Int \rightarrow MatT_{2,2}\ Int$



# $(\hat{\circ}) :: \mathit{MatT}_{3,2}\ \mathit{Int} \rightarrow \mathit{MatT}_{2,3}\ \mathit{Int} \rightarrow \mathit{MatT}_{2,2}\ \mathit{Int}$



# Status and future

- GitHub repository
- Looking for collaboration and hiring recommendations
- To do:
  - Improve performance
  - More examples
  - Genuine sums for circuits
  - Memory and computation management
  - More interpretations (CCCs)