Accelerating Naperian Functors

Nick Hu
nick.hu@cs.ox.ac.uk

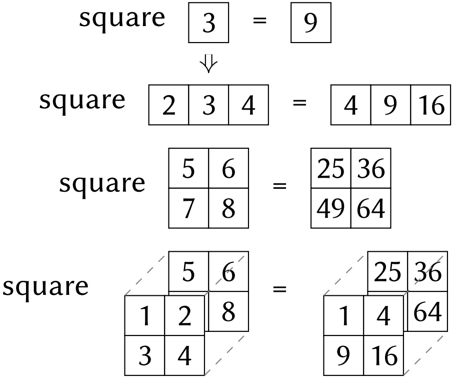
September 22, 2017

UNIVERSITY OF OXFORD
DEPARTMENT OF COMPUTER SCIENCE

The fundamental operations of arrays:
map, zip, fold, traverse, transpose,
replicate

Motivation — going up

- APL is a programming language centred on multidimensional arrays
- It provides lots of seamless adhoc lifting to multiple dimensions, i.e.



Motivation — going up

- Similarly for binary operations

$$\boxed{2} + \boxed{3} = \boxed{5}$$

$$\Downarrow$$

$$\begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline\end{array} + \begin{array}{|c|c|c|}\hline 2 & 3 & 4 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 3 & 3 & 5 \\ \hline\end{array}$$

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline\end{array} + \begin{array}{|c|c|}\hline 5 & 6 \\ \hline 7 & 8 \\ \hline\end{array} = \begin{array}{|c|c|}\hline 6 & 8 \\ \hline 10 & 12 \\ \hline\end{array}$$

- Replicates to satisfy shape constraints — *alignment*

$$\boxed{1} + \begin{array}{|c|c|c|}\hline 2 & 3 & 4 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline\end{array} + \begin{array}{|c|c|c|}\hline 2 & 3 & 4 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline 3 & 4 & 5 \\ \hline\end{array}$$

$$\begin{array}{|c|c|}\hline 2 & 3 \\ \hline\end{array} + \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline\end{array} = \begin{array}{|c|c|}\hline 2 & 3 \\ \hline 2 & 3 \\ \hline\end{array} + \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline\end{array} = \begin{array}{|c|c|}\hline 3 & 5 \\ \hline 5 & 7 \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|}\hline 2 & 3 & 4 \\ \hline\end{array} + \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline\end{array} = \text{Runtime error — unalignable!}$$

Motivation — going up

- A dimensional ordering must be imposed to represent such structures in memory — e.g. *row-major order*
- How can we formalise this and make it type safe?

Static sized vectors

- Using `DataKinds`, we define a type `KnownNat n => Vector n a` to be a vector of `n` elements, each of which are of type `a`

```
newtype Vector (n :: Nat) a = Vector
                                  (Data.Vector.Vector a)
let xs = [3, 4] :: Vector 2 Int -- OverloadedLists
```

$$\simeq \quad \boxed{3} \boxed{4}$$

- Allows the typechecker to catch pre-alignment size-mismatch

```
zipWith :: (a -> b -> c)
        -> Vector n a -> Vector n b -> Vector n c
```

Post-align size-mismatch

- Each `Vector n` gives rise to an `Applicative` functor, with pure given by replication

```
instance KnownNat n => Applicative (Vector n) where
  pure = Vector . Data.Vector.replicate s
    -- black magic Haskell type-to-term cast
    where s = fromIntegral $
                natVal' (proxy# :: Proxy# n) :: Int
```

```
pure xs :: Vector 2 (Vector 2 Int)
```

$\simeq$

3	4
3	4

Alignment is given by applicative functors

Motivation — coming down

- APL also provides operations to reduce along dimensions — reductions and scans, which perform *sequencing*

$$\mathrm{sum}\ \begin{array}{|c|c|c|}\hline 2 & 4 & 6 \\ \hline \end{array} = \begin{array}{|c|}\hline 12 \\ \hline \end{array} \qquad \mathrm{sums}\ \begin{array}{|c|c|c|}\hline 2 & 4 & 6 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 2 & 6 & 12 \\ \hline \end{array}$$

$$\Downarrow$$

$$\mathrm{sum}\ \begin{array}{|c|c|c|}\hline 2 & 4 & 6 \\ \hline 8 & 10 & 12 \\ \hline \end{array} = \begin{array}{|c|}\hline 12 \\ \hline 30 \\ \hline \end{array} \qquad \mathrm{sums}\ \begin{array}{|c|c|c|}\hline 2 & 4 & 6 \\ \hline 8 & 10 & 12 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline 2 & 6 & 12 \\ \hline 8 & 18 & 30 \\ \hline \end{array}$$

Formalising reductions and scans

- Reductions are perfectly captured by `Foldable`

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b

sum :: (Num a, Foldable t) => t a -> a
sum = foldr (+) 0
```

- Scans are perfectly captured by `Traversable` (which is a `Foldable`)

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f
           => (a -> f b) -> t a -> f (t b)
```

Sequencing is given by traversables

Motivation — back around

- Which dimension do we want to sum along?

$$\text{sum}\ \begin{array}{|c|c|c|}\hline 2 & 4 & 6 \\ \hline 8 & 10 & 12 \\ \hline \end{array} \stackrel{?}{=} \begin{array}{|c|}\hline 12 \\ \hline 30 \\ \hline \end{array}$$

$$\begin{array}{c} \rotatebox{90}{$\stackrel{?}{=}$} \\ \begin{array}{|c|c|c|}\hline 10 & 14 & 18 \\ \hline \end{array} \end{array}$$

Motivation — back around

- Refer to the dimensional order imposed and always reduce along the innermost

$$\mathrm{sum}\ \begin{array}{|c|c|c|}\hline 2 & 4 & 6 \\ \hline 8 & 10 & 12 \\ \hline\end{array} = \begin{array}{|c|}\hline 12 \\ \hline 30 \\ \hline\end{array}$$

$$\mathrm{sum} \cdot \mathrm{transpose}\ \begin{array}{|c|c|c|}\hline 2 & 4 & 6 \\ \hline 8 & 10 & 12 \\ \hline\end{array} = \mathrm{sum}\ \begin{array}{|c|c|}\hline 2 & 8 \\ \hline 4 & 10 \\ \hline 6 & 12 \\ \hline\end{array} = \begin{array}{|c|}\hline 10 \\ \hline 14 \\ \hline 18 \\ \hline\end{array}$$

$$= \mathrm{transpose}\ \begin{array}{|c|c|c|}\hline 10 & 14 & 18 \\ \hline\end{array}$$

- *Transposition* is key

Formalising transposition

- For the most general definition, note that there is a type with precisely the same number of inhabitants as the indices of a `Vector n` — the finitely bounded naturals $[0, n)$, `Fin n`
- Thus every `Vector n a` is isomorphic to function `Fin n -> a`

Enter Naperian

- A Naperian functor generalises this notion to any statically sized data structure

```
class Applicative f => Naperian f where
  type Log f -- using TypeFamilies
  lookup :: f a -> (Log f -> a)
  tabulate :: (Log f -> a) -> f a
  positions :: f (Log f)
  tabulate h = fmap h positions
  positions = tabulate id
```

such that `lookup` and `tabulate` are each other's inverse.

- For `Naperian (Vector n)`, `Log f = Fin n`

Naperian transpose

```
transpose :: (Naperian f, Naperian g)
          => f (g a) -> g (f a)
transpose = tabulate . fmap tabulate . flip
          . fmap lookup . lookup
```

… the `fmap`s are function composition

Selection is really transposition, and is given by Naperian functors

Pointwise combinations

- It's just a zip!

```
nzipWith :: Naperian f => (a -> b -> c)
         -> f a -> f b -> f c
nzipWith f xs ys = tabulate (\i -> f
                               (lookup xs i)
                               (lookup ys i)
                             )
```

- Can also get here from `<*>`...

Combination is zipping, and is also given by Naperian functors

Multidimensionality with rank polymorphism

Hypercuboids

- Need a single type containing scalars, vectors, matrices, etc. to define rank-polymorphic operators on

```
data Hyper :: [Type -> Type] -> Type -> Type where
  Scalar :: a -> Hyper '[] a
  Prism :: (Dimension f, Shapely fs)
        => Hyper fs (f a) -> Hyper (f ': fs) a
```

- Contains rank and extent along each dimension at the type level

Accelerate types

- Accelerate is a Haskell DSL for GPU programming, centred around its `Array` type

```
fromList :: (Shape sh, Elt a)
         => sh -> [a] -> Array sh a
-- some shapes
Z :: Z
(Z :. 2) :: Z :. Int
(Z :. 2 :. 3) :: Z :. Int :. Int
```

- `Shape` corresponds to the type-level list of dimensions of `Hyper`...

Rosetta Stone

- Concrete example:

1	2	3
4	5	6

```
m :: Vector 2 (Vector 3 Int)
m = [ [ 1, 2, 3 ],
      [ 4, 5, 6 ] ]

h :: Hyper '[Vector 3, Vector 2] Int
h = Prism . Prism $ Scalar m

a :: Array (Z :. Int :. Int) Int
a = fromList (Z :. 2 :. 3) [1 .. 6]
```

- …but this correspondence is not perfect — Shape lacks information!

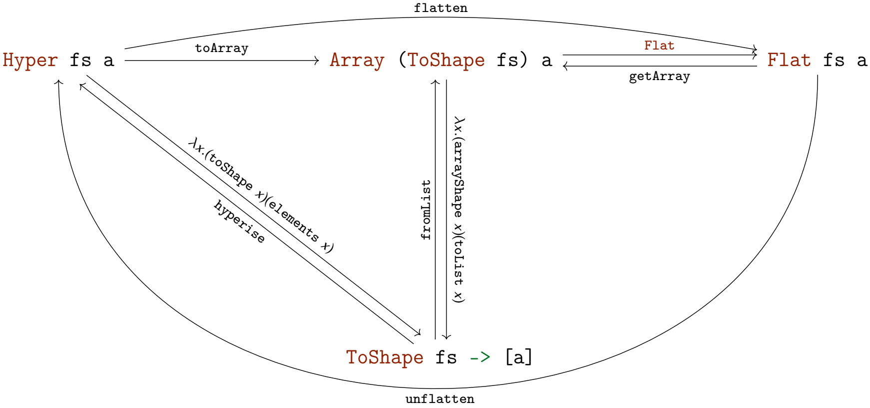
Introducing Flat

```
data Hyper :: [Type -> Type] -> Type -> Type where
  Scalar :: a -> Hyper '[] a
  Prism :: (Dimension f, Shapely fs)
        => Hyper fs (f a) -> Hyper (f ': fs) a

type family ToShape (f :: [Type -> Type]) where
  ToShape '[] = Z
  ToShape (x ': xs) = ToShape xs :. Int

data Flat fs a where
  Flat :: (Shape (ToShape fs))
       => Array (ToShape fs) a -> Flat fs a
```

Hyper-Array-Flat correspondence



Summary

- Modern Haskell facilitates APL features with type safety
- Accelerate provides an interface to the GPU with reasonably nice types
- Plenty of room for improvement
    - Empirical benchmarking required
    - Deal with the boxing — MonoFunctors?
    - Translation between Hyper operators and Accelerate operators

Questions?