APLicative Programming with Naperian Functors

Jeremy Gibbons

WG2.11#16, August 2016

1. Arrays in APL and J

Scalar operation

$$square\ \boxed{3} = \boxed{9}$$

is lifted implicitly to vectors:

$$square\ \boxed{1\ \ 2\ \ 3} = \boxed{1\ \ 4\ \ 9}$$

and to matrices:

$$square\ \boxed{\begin{matrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{matrix}} = \boxed{\begin{matrix}1 & 4 & 9\\ 16 & 25 & 36\\ 49 & 64 & 81\end{matrix}}$$

and to cuboids, etc:

$$square\ \boxed{\begin{matrix}\ & 5 & 6\\ 1 & 2 & 8\\ 3 & 4 & \end{matrix}} = \boxed{\begin{matrix}\ & 25 & 36\\ 1 & 4 & 64\\ 9 & 16 & \end{matrix}}$$

Binary operators

Similarly, binary operators act not only on scalars:

$$\boxed{1} + \boxed{4} = \boxed{5}$$

but also on vectors:

$$\boxed{1\ 2\ 3} + \boxed{4\ 5\ 6} = \boxed{5\ 7\ 9}$$

and on matrices:

$$\boxed{\begin{matrix}1 & 2\\ 3 & 4\end{matrix}} + \boxed{\begin{matrix}5 & 6\\ 7 & 8\end{matrix}} = \boxed{\begin{matrix}6 & 8\\ 10 & 12\end{matrix}}$$

and so on.

Reductions and scans

Similarly for operations that are not simply pointwise.

The *sum* and prefix *sums* functions on vectors:

$$sum \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 6 \end{bmatrix}$$

$$sums \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 6 \end{bmatrix}$$

lift to act on the rows of a matrix:

$$sum \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 6 \\ 15 \end{bmatrix}$$

$$sums \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 6 \\ 4 & 9 & 15 \end{bmatrix}$$

Reranking

J provides a reranking operator $"_1$ allowing action instead on the columns of a matrix:

$$sum\ "_1 \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = sum\ (transpose \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}) = sum \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 5 & 7 & 9 \end{bmatrix}$$

$$sums\ "_1 \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = transpose\ (sums\ (transpose \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}))$$

$$= transpose\ (sums \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}) = transpose \begin{bmatrix} 1 & 5 \\ 2 & 7 \\ 3 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \end{bmatrix}$$

Alignment

The arguments of a binary operator need not have the same rank: lower-ranked argument is implicitly lifted to align with higher-ranked.

For example, one can add a scalar and a vector:

$$3 + \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 3 & 3 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 7 & 8 & 9 \end{bmatrix}$$

or a vector and a matrix:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 5 & 7 & 9 \\ 8 & 10 & 12 \end{bmatrix}$$

The *shapes* at common ranks must match.

2. Typing rank polymorphism

In APL and J, shape checking is *dynamic*.

Recent work by Slepak *et al.* on *Remora*, a language with a *static type system* for shape checking.

An Array-Oriented Language with Static Rank Polymorphism

Justin Slepak, Olin Shivers, and Panagiotis Manolios

Northeastern University
{jrslepak,shivers,pete}@ccs.neu.edu

Abstract. The array-computational model pioneered by Iverson's languages APL and J offers a simple and expressive solution to the "von Neumann bottleneck." It includes a form of rank, or dimensional, polymorphism, which renders much of a program's control structure implicit by lifting base operators to higher-dimensional array structures. We present the first formal semantics for this model, along with the first static type system that captures the full power of the core language.

The formal dynamic semantics of our core language, Remora, illuminates several of the murkier corners of the model. This allows us to resolve some of the model's *ad hoc* elements in more general, regular ways. Among these, we can generalise the model from SIMD to MIMD computations, by extending the semantics to permit functions to be lifted to higher-dimensional arrays in the same way as their arguments.

Our static semantics, a dependent type system of carefully restricted power, is capable of describing array computations whose dimensions cannot be determined statically. The type-checking problem is decidable and the type system is accompanied by the usual soundness theorems. Our type system's principal contribution is that it serves to extract the implicit control structure that provides so much of the language's expressive power, making this structure explicitly apparent at compile time.

1 The Promise of Rank Polymorphism

Behind every interesting programming language is an interesting model of computation. For example, the lambda calculus, the relational calculus, and finite-state automata are the computational models that, respectively, make Scheme, SQL and regular expressions interesting programming languages. Iverson's language APL [7], and its successor J [10], are interesting for this very reason. That is, they provide a notational interface to an interesting model of computation: loop-free, recursion-free array processing, a model that is becoming increasingly relevant as we move into an era of parallel computation.

APL and J's array-computation model is important for several reasons. First, the model provides a solution to Backus's "von Neumann bottleneck" [1]. Instead of using iteration or recursion, all operations are automatically aggregate operations. This lifting is the fundamental control flow mechanism. The iteration space associated with array processing is reified as the shape of the arrays being

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 27–46, 2014.

$$
\begin{array}{rcll}
e & ::= & \alpha \mid x \mid (e\ e'\ \ldots) \mid (\mathtt{T}\lambda\,[x\ \ldots]\,e) \mid (\mathtt{T\text{-}APP}\ e\ \tau\ \ldots) & \textit{(exressions)} \\
 & & \mid (\mathtt{I}\lambda\,[(x\ \gamma)\ \ldots]\,e) \mid (\mathtt{I\text{-}APP}\ e\ \iota\ \ldots) \mid (\mathtt{PACK}\ \iota\ \ldots\ e)^{\tau} \mid (\mathtt{UNPACK}\ (\langle x\ \ldots\ |y\rangle = e)\ e') & \\
\alpha & ::= & [l\ \ldots]^{\tau} \mid [l\ l'\ \ldots]^{\iota} & \textit{(arrays)} \\
l & ::= & b \mid f \mid e \mid (\mathtt{T}\lambda\,[x\ \ldots]\,l) \mid (\mathtt{T\text{-}APP}\ l\ \tau\ \ldots) \mid (\mathtt{I}\lambda\,[(x\ \gamma)\ \ldots]\,l) & \textit{(array elements)} \\
 & & \mid (\mathtt{I\text{-}APP}\ l\ \iota\ \ldots) & \\
f & ::= & \pi \mid (\lambda\ [(x\ \tau)\ \ldots]\ e) & \textit{(functions)} \\
\tau, \sigma & ::= & B \mid x \mid \mathtt{A}_{\iota}\tau \mid (\tau\ \ldots \to \sigma) \mid (\forall\,[x\ \ldots]\,\tau) \mid (\Pi\,[(x\ \gamma)\ \ldots]\,\tau) & \textit{(types)} \\
 & & \mid (\Sigma\,[(x\ \gamma)\ \ldots]\,\tau) & \\
\iota, \kappa & ::= & n \mid x \mid (\mathtt{S}\ \iota\ \ldots) \mid (+\ \iota\ \kappa) & \textit{(indices)} \\
\gamma & ::= & \mathtt{Nat} \mid \mathtt{Shape} & \textit{(index sorts)} \\
z & \in & \mathbb{Z} & \textit{(numbers)} \\
n, m & \in & \mathbb{N} & \\
v & ::= & [b\ \ldots]^{\tau} \mid [f\ \ldots]^{\tau} \mid b \mid f \mid (\mathtt{T}\lambda\,[x\ \ldots]\,l) \mid (\mathtt{I}\lambda\,[(x\ \gamma)\ \ldots]\,l) & \textit{(value forms)} \\
 & & \mid (\mathtt{PACK}\ \iota\ \ldots\ v) \mid [(\mathtt{PACK}\ \iota\ \ldots\ v)\ \ldots]^{\mathtt{A}(\mathtt{S}\ m\ n\ \ldots)\,\tau} & \\
E & ::= & \square \mid (v\ \ldots\ E\ e\ \ldots) \mid [v\ \ldots\ E\ l\ \ldots]^{\tau} \mid (\mathtt{T\text{-}APP}\ E\ \tau\ \ldots) & \textit{(evaluation contexts)} \\
 & & \mid (\mathtt{I\text{-}APP}\ E\ \iota\ \ldots) \mid (\mathtt{PACK}\ \iota\ \ldots\ E)^{\tau} \mid (\mathtt{UNPACK}\ (\langle x\ \ldots\ |y\rangle = E)\ e) & \\
\Gamma & ::= & \cdot \mid \Gamma, (x : \tau) & \textit{(type environments)} \\
\Delta & ::= & \cdot \mid \Delta, x & \textit{(kind environments)} \\
\Theta & ::= & \cdot \mid \Theta, (x :: \gamma) & \textit{(sort environments)}
\end{array}
$$

Fig. 6. Syntax for Remora

$\boxed{\Gamma; \Delta; \Theta \vdash l : \tau}$

$$\frac{}{\Gamma; \Delta; \Theta \vdash num : \texttt{Num}} \quad \text{(T-Num)} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma; \Delta; \Theta \vdash x : \tau} \quad \text{(T-Var)}$$

$$\frac{\tau \cong \sigma \qquad \Gamma; \Delta; \Theta \vdash l : \tau}{\Gamma; \Delta; \Theta \vdash l : \sigma} \quad \text{(T-Equiv)} \qquad \frac{\begin{array}{c}\Gamma; \Delta; \Theta \vdash l_j : \tau \quad \text{for each } l_j \in l \ldots \\ \mathit{Product}\,[\![n \ldots]\!] = \mathit{Length}\,[\![elt \ldots]\!]\end{array}}{\Gamma; \Delta; \Theta \vdash [l \ldots]^{\texttt{A}(\texttt{S}\ n \ldots)\tau} : \texttt{A}_{(\texttt{S}\ n \ldots)}\tau} \quad \text{(T-Array)}$$

$$\frac{\Gamma, (x : \tau) \ldots ; \Delta; \Theta \vdash e : \sigma}{\begin{array}{c}\Gamma; \Delta; \Theta \vdash (\lambda\,[(x\ \tau) \ldots]\ e) : \\ (\tau \ldots \to \sigma)\end{array}} \quad \text{(T-Abst)} \qquad \frac{\begin{array}{c}\Gamma; \Delta; \Theta \vdash e : \texttt{A}_\iota\,(\sigma \ldots \to \tau) \\ \Gamma; \Delta; \Theta \vdash e'_j : \texttt{A}_{\kappa_j}\sigma_j \quad \text{for each } j \\ \iota' = \mathit{Max}\,[\![\iota, \kappa \ldots]\!]\end{array}}{\Gamma; \Delta; \Theta \vdash (e\ e' \ldots) : \texttt{A}_{\iota'}\tau} \quad \text{(T-App)}$$

$$\frac{\Gamma; \Delta, x \ldots ; \Theta \vdash e : \tau}{\begin{array}{c}\Gamma; \Delta; \Theta \vdash (\texttt{T}\lambda\ [x \ldots]\ e) : \\ (\forall\,[x \ldots]\,\tau)\end{array}} \quad \text{(T-TAbst)} \qquad \frac{\begin{array}{c}\Gamma; \Delta; \Theta \vdash l : (\forall\,[x \ldots]\,\sigma) \\ \Delta; \Theta \vdash \tau_j \quad \text{for each } j \\ \text{no } \tau_j \text{ is an array type}\end{array}}{\begin{array}{c}\Gamma; \Delta; \Theta \vdash (\texttt{T-APP}\ l\ \tau \ldots) : \\ \sigma[(x \leftarrow_t \tau) \ldots]\end{array}} \quad \text{(T-TApp)}$$

$$\frac{\Gamma; \Delta; \Theta, (x :: \gamma) \ldots \vdash e : \tau}{\begin{array}{c}\Gamma; \Delta; \Theta \vdash (\texttt{I}\lambda\ [(x) \ldots]\ e) : \\ (\Pi\,[(x\ \gamma) \ldots]\,\tau)\end{array}} \quad \text{(T-IAbst)} \qquad \frac{\begin{array}{c}\Gamma; \Delta; \Theta \vdash e : (\Pi\,[(x\ \gamma) \ldots]\,\tau) \\ \Gamma; \Delta; \Theta \vdash \iota_j :: \gamma_j \quad \text{for each } j\end{array}}{\begin{array}{c}\Gamma; \Delta; \Theta \vdash (\texttt{I-APP}\ e\ \iota \ldots) : \\ \tau[(x \leftarrow_i \iota) \ldots]\end{array}} \quad \text{(T-IApp)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \Theta \vdash e : \tau\,[(x \leftarrow \iota) \ldots] \\ \Gamma; \Delta; \Theta \vdash \iota_j :: \gamma_j \quad \text{for each } j\end{array}}{\Gamma; \Delta; \Theta \vdash (\texttt{PACK}\ \iota \ldots\ e) : (\Sigma\,[(x\ \gamma) \ldots]\,\tau)} \quad \text{(T-Pack)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \Theta \vdash e : (\Sigma\ [(x\ \gamma) \ldots]\,\sigma) \\ \Gamma, y : \sigma; \Delta; \Theta, (x :: \gamma) \ldots \vdash e' : \tau \\ \Delta; \Theta \vdash \tau\end{array}}{\Gamma; \Delta; \Theta \vdash (\texttt{UNPACK}\ (\langle x \ldots | y \rangle = e)\ e') : \tau} \quad \text{(T-Unpack)}$$

Fig. 7. Type judgment for Remora

$\boxed{\Delta;\Theta \vdash \tau}$

$$\frac{}{\Delta;\Theta \vdash B}\ (\text{K-Base}) \qquad \frac{x \in \Delta}{\Delta;\Theta \vdash x}\ (\text{K-Var}) \qquad \frac{\Delta;\Theta \vdash \tau \quad \Theta \vdash \iota :: \texttt{Shape}}{\Delta;\Theta \vdash \texttt{A}_\iota \tau}\ (\text{K-Array})$$

$$\frac{\Delta;\Theta \vdash \tau_j \text{ for each } j \qquad \Delta;\Theta \vdash \sigma}{\Delta;\Theta \vdash (\tau \ \ldots \ \to \sigma)}\ (\text{K-Fun}) \qquad \frac{\Delta;\Theta,(x :: \gamma) \ \ldots \vdash \tau}{\Delta;\Theta \vdash (\Pi\ [(x\ \gamma)\ \ldots]\ \tau)}\ (\text{K-DProd})$$

$$\frac{\Delta;\Theta,(x :: \gamma) \ \ldots \vdash \tau}{\Delta;\Theta \vdash (\Sigma\ [(x\ \gamma)\ \ldots]\ \tau)}\ (\text{K-DSum}) \qquad \frac{\Delta, x\ \ldots\ ;\Theta \vdash \tau}{\Delta;\Theta \vdash (\forall\ [x\ \ldots]\ \tau)}\ (\text{K-Univ})$$

$\boxed{\Theta \vdash \iota :: \gamma}$

$$\frac{n \in \mathbb{N}}{\Theta \vdash n :: \texttt{Nat}}\ (\text{S-Nat}) \qquad \frac{(x :: \gamma) \in \Theta}{\Theta \vdash x :: \gamma}\ (\text{S-Var}) \qquad \frac{\Theta \vdash \iota_j :: \texttt{Nat} \text{ for each } j}{\Theta \vdash (\texttt{S}\ \iota\ \ldots) :: \texttt{Shape}}\ (\text{S-Shape})$$

$$\frac{\Theta \vdash \iota :: \texttt{Nat} \qquad \Theta \vdash \kappa :: \texttt{Nat}}{\Theta \vdash (+\ \iota\ \kappa) :: \texttt{Nat}}\ (\text{S-Plus})$$

Fig. 8. Kind and index sort judgments for Remora

Pointwise application:

$$\left([f\ \ldots]^{\mathtt{A}(\mathtt{S}\ n_f\ \ldots)(\mathtt{A}(\mathtt{S}\ n_a\ \ldots)\tau\ \ldots\to\tau')}\ v^{\mathtt{A}(\mathtt{S}\ n_f\ \ldots\ n_a\ \ldots)\tau}\ \ldots\right)^{\mathtt{A}(\mathtt{S}\ n_f\ \ldots\ n_c\ \ldots)\tau'}$$

$$\mapsto_{map}\ \left[\left([f]^{\mathtt{A}(\mathtt{S})(\mathtt{A}(\mathtt{S}\ n_a\ \ldots)\tau\ \ldots\to\tau')}\ \alpha^{\mathtt{A}(\mathtt{S}\ n_a\ \ldots)\tau}\ \ldots\right)^{\tau'}\ \ldots\right]^{\mathtt{A}(\mathtt{S}\ n_f\ \ldots)\tau'}$$

where $\rho = \mathit{length}\ (n_f\ \ldots) > 0$

$((\alpha\ \ldots)\ \ldots) = ((\mathit{Cells}_\rho\ [\![v]\!])\ \ldots)^\top$

Duplicating cells:

$$\left([f\ \ldots]^{\mathtt{A}(\mathtt{S}\ m\ \ldots)(\mathtt{A}(\mathtt{S}\ n\ \ldots)\tau\ \ldots\to\tau')}\ v^{\mathtt{A}(\mathtt{S}\ m'\ \ldots)\tau}\ \ldots\right)^\sigma$$

$$\mapsto_{lift}\ \left(\mathit{Dup}_{(\mathtt{A}(\mathtt{S}\ n\ \ldots)\tau\ \ldots\to\tau'),\iota}\ [\![[f\ \ldots]]\!]\ \mathit{Dup}_{\mathtt{A}(\mathtt{S}\ m'\ \ldots)\tau,\iota}\ [\![v]\!]\ \ldots\right)^\sigma$$

where $(m\ \ldots), (m'\ \ldots)\ \ldots$ not all equal

$\iota = \mathit{Max}\ [\![(m\ \ldots), (m'\ \ldots)\ \ldots]\!]$

Applying a type abstraction:

$$\left(\mathtt{T\text{-}APP}\ (\mathtt{T}\lambda\ [x\ \ldots]\ e^\tau)^{(\forall[x\ \ldots]\tau)}\ \sigma\ \ldots\right)^{\tau[(x\ \leftarrow_t\ \sigma)\ \ldots]}\ \mapsto_{T\beta}\ e^\tau\ [(x\ \leftarrow_t\ \sigma)\ \ldots]$$

Applying an index abstraction:

$$\left(\mathtt{I\text{-}APP}\ (\mathtt{I}\lambda\ [(x\ \gamma)\ \ldots]\ e^\tau)^{(\Pi[(x\ \gamma)\ \ldots]\tau)}\ \iota\ \ldots\right)^{\tau[(x\ \leftarrow_i\ \iota)\ \ldots]}\ \mapsto_{I\beta}\ e^\tau\ [(x\ \leftarrow_i\ \iota)\ \ldots]$$

Projecting from a dependent sum:

$$\left(\mathtt{UNPACK}\ \left(\langle x\ \ldots\ |y\rangle = (\mathtt{PACK}\ \iota\ \ldots\ v^\tau)^{\tau'}\right)\ e^\sigma\right)^\sigma\ \mapsto_{proj}\ e^\sigma\ [(x\ \leftarrow_i\ \iota)\ \ldots\ (y\ \leftarrow_e\ v)]$$

Fig. 9. Small-step operational semantics for Remora

3. Vectors

Automatic promotion of datatype

```
data Nat :: * where
  Z ::          Nat
  S :: Nat → Nat
```

to kind *Nat* and type-level naturals $'Z, 'S\ 'Z, \ldots$.

Then we can define

```
data Vector :: Nat → * → * where
  VNil  ::                        Vector 'Z a
  VCons :: a → Vector n a → Vector ('S n) a
```

It is now straightforward to define

```
vmap     :: (a → b) → Vector n a → Vector n b
vzipWith :: (a → b → c) → Vector n a → Vector n b → Vector n c
```

Hasochism

That's not quite enough for *vreplicate*, crucial for alignment.

```
class Natural (n :: Nat)
  where vreplicate :: a → Vector n a

instance Natural 'Z
  where vreplicate a = VNil
instance Natural n ⇒ Natural ('S n)
  where vreplicate a = VCons a (vreplicate a)
```

4. Applicative functors

Vectors are an *applicative functor*:

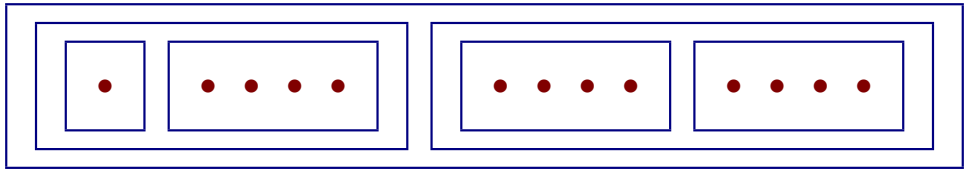
```
class Functor f ⇒ Applicative f where
  pure :: a → f a                    -- think "replicate"
  (⊛)  :: f (a → b) → f a → f b      -- think "zip with apply"
```

Not just vectors as dimensions; also eg pairs:

```
data Pair a = P a a

instance Functor Pair where
  fmap f (P x y) = P (f x) (f y)

instance Applicative Pair where
  pure x         = P x x
  P f g ⊛ P x y = P (f x) (g y)
```

. . . or block-structured matrices

```
data Block :: ∗ where
  Single :: Block
  Join   :: Block → Block → Block

data BlockVec :: Block → ∗ → ∗ where
  One  :: a →                                  BlockVec Single a
  Plus :: BlockVec m a → BlockVec n a → BlockVec (Join m n) a

instance Functor (BlockVec p) where ...
instance Applicative (BlockVec p) where ...
```



Dimensions can have structure!

5. Naperian functors

The *Applicative* interface is not enough to define *transpose*, needed for reranking. Instead:

```
class Applicative f ⇒ Naperian f where
  type Log f
  lookup    :: f a → (Log f → a)   -- each other's...
  tabulate  :: (Log f → a) → f a   -- ...inverses
  positions :: f (Log f)

  tabulate h = fmap h positions
  positions  = tabulate id
```

Then

```
transpose :: (Naperian f, Naperian g) ⇒ f (g a) → g (f a)
transpose = tabulate ∘ fmap tabulate ∘ flip ∘ fmap lookup ∘ lookup
```

6. Folding and traversing

To define things like *sum*, we need

```
class Foldable t where
  foldMap :: Monoid m ⇒ (a → m) → (t a → m)
```

And to define things like *sums*, we need

```
class (Functor t, Foldable t) ⇒ Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
```

So we represent acceptable array dimensions as:

```
class (Naperian f, Traversable f) ⇒ Dimension f
```

7. Multidimensionality

Hypercuboids are scalars, vectors, matrices... a *nested* datatype:

```
data Hyper :: ∗ → ∗ where
  Scalar ::                    a →                          Hyper a
  Prism  :: Natural n ⇒ Hyper (Vector n a) →                Hyper a
```

or

```
data Hyper :: Nat → ∗ → ∗ where
  Scalar ::                    a →                          Hyper 'Z a
  Prism  :: Natural n ⇒ Hyper r (Vector n a) →              Hyper ('S r) a
```

or (*innermost* extent first)

```
data Hyper :: [Nat] → ∗ → ∗ where
  Scalar ::                    a →                          Hyper '[] a
  Prism  :: Natural n ⇒ Hyper ns (Vector n a) →             Hyper (n': ns) a
```

But what about non-vector dimensions?

Beyond vectors

Type index is a *type-level list of dimensions*:

```
class Shapely fs                where ...
instance Shapely '[]            where ...
instance (Dimension f, Shapely fs) ⇒
                Shapely (f ': fs) where ...
```

Then (innermost first again)

```
data Hyper :: [∗ → ∗] → ∗ → ∗ where
  Scalar ::                                    a →                  Hyper '[] a
  Prism  :: (Dimension f, Shapely fs) ⇒ Hyper fs (f a) → Hyper (f ': fs) a
```

Beyond vectors

Type index is a *type-level list of dimensions*:

```
class Shapely fs            where rank :: Rank fs
instance Shapely '[]        where rank = RZ
instance (Dimension f, Shapely fs) ⇒
         Shapely (f ': fs) where rank = RS rank
```

Then (innermost first again)

```
data Hyper :: [∗ → ∗] → ∗ → ∗ where
  Scalar ::                                  a →              Hyper '[] a
  Prism  :: (Dimension f, Shapely fs) ⇒ Hyper fs (f a) → Hyper (f ': fs) a
```

where a *Rank* denotes the rank of a shape:

```
data Rank :: [∗ → ∗] → ∗ where
  RZ ::                                          Rank '[]
  RS :: (Dimension f, Shapely fs) ⇒ Rank fs → Rank (f ': fs)
```

Hypercuboid operations

Replication and zipping (and hence applicative):

$$
\begin{aligned}
&\mathit{hreplicate} :: \mathit{Rank}\ \mathit{fs} \to a \to \mathit{Hyper}\ \mathit{fs}\ a \\
&\mathit{hreplicate}\ \mathit{RZ}\ a = \mathit{Scalar}\ a \\
&\mathit{hreplicate}\ (\mathit{RS}\ r)\ a = \mathit{Prism}\ (\mathit{hreplicate}\ r\ (\mathit{pure}\ a))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hzipWith} :: (a \to b \to c) \to \mathit{Hyper}\ \mathit{fs}\ a \to \mathit{Hyper}\ \mathit{fs}\ b \to \mathit{Hyper}\ \mathit{fs}\ c \\
&\mathit{hzipWith}\ f\ (\mathit{Scalar}\ a)\ (\mathit{Scalar}\ b) = \mathit{Scalar}\ (f\ a\ b) \\
&\mathit{hzipWith}\ f\ (\mathit{Prism}\ x)\ \ (\mathit{Prism}\ y)\ \ = \mathit{Prism}\ (\mathit{hzipWith}\ (\mathit{azipWith}\ f)\ x\ y)
\end{aligned}
$$

Reduction:

$$
\begin{aligned}
&\mathit{reduce} :: \mathit{Monoid}\ m \Rightarrow (a \to m) \to \mathit{Hyper}\ (f' : \mathit{fs})\ a \to \mathit{Hyper}\ \mathit{fs}\ m \\
&\mathit{reduce}\ f\ (\mathit{Prism}\ x) = \mathit{fmap}\ (\mathit{foldMap}\ f)\ x
\end{aligned}
$$

and transposition:

$$
\begin{aligned}
&\mathit{transposeHyper} :: \mathit{Hyper}\ (f' : (g' : \mathit{fs}))\ a \to \mathit{Hyper}\ (g' : (f' : \mathit{fs}))\ a \\
&\mathit{transposeHyper}\ (\mathit{Prism}\ (\mathit{Prism}\ x)) = \mathit{Prism}\ (\mathit{Prism}\ (\mathit{fmap}\ \mathit{transpose}\ x))
\end{aligned}
$$

7. Alignment

Unary operators via *fmap*, homogeneous binary via *hzipWith*.

Heterogeneous binary operators (eg vector with matrix) entail *alignment*:

```
class (Shapely fs, Shapely gs) ⇒ Alignable fs gs where
  align :: Hyper fs a → Hyper gs a
```

Shape *fs* alignable with *gs* if it is a *prefix*:

```
instance Alignable '[ ] '[ ] where
  align = id

instance (Dimension f, Alignable fs gs) ⇒ Alignable (f ': fs) (f ': gs) where
  align (Prism x) = Prism (align x)

instance (Dimension f, Shapely fs) ⇒ Alignable '[ ] (f ': fs) where
  align (Scalar a) = hreplicate rank a
```

Lifting

Two arguments can be aligned with their common maximum shape:

```
type family Max (fs :: [* → *]) (gs :: [* → *]) :: [* → *]
type instance Max '[]       '[]       =' []
type instance Max '[]       (f ': gs) = (f ': gs)
type instance Max (f ': fs) '[]       = (f ': fs)
type instance Max (f ': fs) (f ': gs) = (f ': Max fs gs)
```

Then

```
binary :: (Shapely fs, Shapely gs, Max fs gs ~ hs,
           Alignable fs hs, Alignable gs hs) ⇒
          (a → b → c) → (Hyper fs a → Hyper gs b → Hyper hs c)
binary f x y = hzipWith f (align x) (align y)
```

8. Symbolic replication and transposition

```
data HyperR :: [* → *] → * → * where
  ScalarR :: a →                                  HyperR' [] a
  PrismR  :: (Dimension f, Shapely fs) ⇒
             HyperR fs (f a) →                    HyperR (f' : fs) a
  ReplR   :: (Dimension f, Shapely fs) ⇒
             HyperR fs a →                        HyperR (f' : fs) a
  TransR  :: (Dimension f, Dimension g, Shapely fs) ⇒
             HyperR (f' : g' : fs) a → HyperR (g' : f' : fs) a
```

then

```
rzipWith :: Shapely fs ⇒
            (a → b → c) → HyperR fs a → HyperR fs b → HyperR fs c
rzipWith f (PrismR x) (ReplR y) = PrismR (rzipWith (azipWith1 f) x y)
  where azipWith1 f xs y = fmap (`f` y) xs
...
```

9. Flat representation

```
data Flat fs a where
  Flat :: Shapely fs ⇒ Array Int a → Flat fs a

flatten :: Shapely fs ⇒ Hyper fs a → Flat fs a
flatten xs = Flat (listArray (0, sizeHyper xs − 1) (elements xs))

where

sizeHyper :: Shapely fs ⇒ Hyper fs a → Int
elements  :: Shapely fs ⇒ Hyper fs a → [a]
```

10. Summary

- typing of APL and J array operations
- explicating the implicit alignment and lifting
- symbolic (constant-time) replication and transposition
- type-safe flat representations

- no need for hand-rolled type system
- not too much pain

> O, my offence is rank, it smells to heaven;
> It hath the primal eldest curse upon 't,
> A brother's murder.
> *William Shakespeare (1564–1616), "Hamlet", Act III Scene 3*