APLicative Programming with Naperian Functors

Extended abstract

Jeremy Gibbons
University of Oxford, UK

1. APL Arrays

Array-oriented programming languages such as APL [1] and J [2] pay special attention to manipulating *array structures*: rank-one vectors (sequences of values), rank-two matrices (which can be seen as rectangular sequences of sequences), rank-three cuboids (sequences of sequences of sequences), rank-zero scalars, and so on.

One appealing consequence of this unification is the prospect of *rank polymorphism* [7]—that a scalar function may be automatically lifted to act element-by-element on a higher-ranked array, a scalar binary operator to act pointwise on pairs of arrays, and so on. For example, numeric function *square* acts not only on scalars, but also on vectors:

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

Similarly, binary operators act not only on scalars, but also on vectors and matrices:

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

The same lifting can be applied to operations that do not simply act pointwise. For example, the *sum* and prefix *sums* functions on vectors can also be applied to matrices:

$$sum\ \boxed{\begin{matrix}1&2&3\\4&5&6\end{matrix}} = \boxed{\begin{matrix}6\\15\end{matrix}} \qquad sums\ \boxed{\begin{matrix}1&2&3\\4&5&6\end{matrix}} = \boxed{\begin{matrix}1&3&6\\4&9&15\end{matrix}}$$

In the examples above, *sum* and *sums* have been lifted to act on the rows of the matrix; J also provides a *reranking* operator $"_1$, making them act instead on the columns—essentially a matter of matrix transposition:

$$sum\ "_1\ \boxed{\begin{matrix}1&2&3\\4&5&6\end{matrix}} = sum\ \boxed{\begin{matrix}1&4\\2&5\\3&6\end{matrix}} = \boxed{5\ 7\ 9}$$

Furthermore, the arguments of binary operators need not have the same rank; the lower-ranked argument is implicitly lifted to align with the higher-ranked one—essentially a matter of replication. For example, one can add a scalar and a vector, or a vector and a matrix:

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

2. Embedding Static Typing

Slepak *et al.* [7] present static and dynamic semantics for a typed core language Remora. Their semantics clarifies the axes of variability illustrated above; in particular, it makes explicit the implicit control structures and data manipulations required for lifting operators to higher-ranked arguments and aligning arguments of different ranks. Moreover, the type system makes it a static error if the *shapes* in a given dimension do not match. They model the type and evaluation rules of Remora in PLT Redex, and use this model to prove important properties such as type safety. PLT Redex provides complete freedom to model whatever semantics the language designer chooses; but the *quid pro quo* for this freedom is that it does not directly lead to a full language implementation.

This is the usual trade-off between standalone and embedded DSLs. If the type rules of Remora had been embedded instead in a sufficiently expressive typed host language, then the surrounding ecosystem of that host language—type inference, the compiler, libraries, code generation—could be leveraged immediately to provide a practical programming vehicle. The challenge then becomes to find the right host language, and to work out how best to represent the rules of the DSL within the features available in that host language. We show how to capture the type constraints and implicit lifting and alignment manipulations of rank-polymorphic array computation neatly in Haskell (plus recent extensions).

3. The Core Idea

The core idea behind implicit lifting of operations turns out to be a matter of *applicative functors* [6], which provide precisely the necessary 'map' and 'zip' operators. That insight points to an intriguing generalization of array dimensions: they need not necessarily be vectors, but might take the form of some other applicative functor than vectors, such as pairs, or perfect binary trees—the necessary structure seems to be that of a *traversable Naperian functor*. The richer type structure that this makes available allows us to explain the relationship between nested and flat representations of multi-dimensional data, leading the way towards much more efficient implementations of bulk operations [3].

4. Vectors with Bounds Checking

To a first approximation, each dimension of an array is a vector of some length. We can capture these in languages like Haskell by 'faking' [5] dependently typed programming through promoting [8] value-level data to the type level. Thus, from the definition

$$\begin{array}{l}\textbf{data}\ Nat :: * \ \textbf{where}\\ \quad Z :: \quad Nat\\ \quad S :: Nat \to Nat\end{array}$$

we get not only a new type *Nat* with value inhabitants $Z, S\,Z, \ldots$, but in addition a new kind also called *Nat* with type inhabitants $'Z, 'S\,'Z, \ldots$. We can now define a datatype of length-indexed vectors:

$$\begin{array}{l}\textbf{data}\ Vector :: Nat \to * \to * \ \textbf{where}\\ \quad VNil \quad :: \qquad\qquad Vector\ 'Z\ a\\ \quad VCons :: a \to Vector\ n\ a \to Vector\ ('S\ n)\ a\end{array}$$

supporting useful operations:

$$vmap :: (a \to b) \to Vector\ n\ a \to Vector\ n\ b$$
$$vzipWith :: (a \to b \to c) \to Vector\ n\ a \to Vector\ n\ b \to Vector\ n\ c$$

For functions that generate vectors, we need to make available run-time analogues of compile-time type information [4]:

$$\textbf{data}\ Natty :: Nat \to * \ \textbf{where}\ ...$$
$$vreplicate :: Natty\ n \to a \to Vector\ n\ a$$

The operations *vmap*, *vzipWith*, and *vreplicate* are the key to lifting and aligning operations to higher-ranked arguments.

5. Applicative, Naperian, Foldable, Traversable

We need not restrict the dimensions of an array to be vectors: other types, such as pairs, can serve just as well. Each dimension should be a type constructor f with *applicative* structure [6]:

$$\textbf{class}\ Functor\ f\ \textbf{where}$$
$$\quad fmap :: (a \to b) \to f\ a \to f\ b$$
$$\textbf{class}\ Functor\ f \Rightarrow Applicative\ f\ \textbf{where}$$
$$\quad pure :: a \to f\ a$$
$$\quad (\circledast) :: f\ (a \to b) \to f\ a \to f\ b$$

In order to support transposition, it should also be what Peter Hancock calls *Naperian*, that is, a container of fixed shape:

$$\textbf{class}\ Applicative\ f \Rightarrow Naperian\ f\ \textbf{where}$$
$$\quad \textbf{type}\ Log\ f$$
$$\quad lookup :: f\ a \to (Log\ f \to a)$$
$$\quad table :: (Log\ f \to a) \to f\ a$$

On such types, transposition is total and invertible:

$$transpose :: (Naperian\ f, Naperian\ g) \Rightarrow f\ (g\ a) \to g\ (f\ a)$$
$$transpose = table \circ fmap\ table \circ flip \circ fmap\ lookup \circ lookup$$

In order to reduce along an array dimension, for example to sum, the functor should be *Foldable*; and in order to scan along a dimension, for example to compute prefix sums, it should be *Traversable*:

$$\textbf{class}\ Foldable\ t\ \textbf{where}$$
$$\quad foldMap :: Monoid\ m \Rightarrow (a \to m) \to (t\ a \to m)$$
$$\textbf{class}\ (Functor\ t, Foldable\ t) \Rightarrow Traversable\ t\ \textbf{where}$$
$$\quad traverse :: Applicative\ f \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b)$$

Dimensions have to satisfy all these constraints:

$$\textbf{class}\ (Naperian\ f, Traversable\ f) \Rightarrow Dim\ f$$

6. Multidimensionality

The shape of a multidimensional array is a composition of dimension functors (innermost first):

$$\textbf{class}\ Shapely\ fs \qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ ...$$
$$\textbf{instance} \qquad\qquad\qquad\qquad Shapely\ '[] \qquad \textbf{where}\ ...$$
$$\textbf{instance}\ (Dim\ f, Shapely\ fs) \Rightarrow Shapely\ (f\ ':fs) \ \textbf{where}\ ...$$

Now, a hypercuboid of type *Hyper fs a* has shape *fs* (a list of dimensions) and elements of type *a*. The rank zero hypercuboids are scalars; at higher ranks, one can think of them as geometrical 'right prisms'—congruent stacks of lower-rank hypercuboids:

$$\textbf{data}\ Hyper :: [* \to *] \to * \to * \ \textbf{where}$$
$$\quad Scalar :: a \to Hyper\ '[]\ a$$
$$\quad Prism :: (Dim\ f, Shapely\ fs) \Rightarrow$$
$$\qquad Hyper\ fs\ (f\ a) \to Hyper\ (f\ ':fs)\ a$$

These are again *Applicative*, *Naperian*, *Foldable*, and *Traversable*.

7. Alignment

We can easily lift a unary operator to act on a hypercuboid:

$$unary :: Shapely\ fs \Rightarrow (a \to b) \to (Hyper\ fs\ a \to Hyper\ fs\ b)$$
$$unary = fmap$$

We can similarly lift a binary operator to act on hypercuboids of matching shapes, using *liftA2*. But what about when the shapes don't match? A shape *fs* is *alignable* with another shape *gs* if the type-level list of dimensions *fs* is a prefix of *gs*; in that case, we can replicate the *fs*-hypercuboid to yield a *gs*-hypercuboid.

$$\textbf{class}\ (Shapely\ fs, Shapely\ gs) \Rightarrow Align\ fs\ gs\ \textbf{where}$$
$$\quad align :: Hyper\ fs\ a \to Hyper\ gs\ a$$
$$\textbf{instance} \qquad\qquad Align\ '[]\ '[] \qquad\qquad \textbf{where}\ ...$$
$$\textbf{instance}\ (Dim\ f, Align\ fs\ gs) \Rightarrow Align\ (f\ ':fs)\ (f\ ':gs)\ \textbf{where}\ ...$$
$$\textbf{instance}\ (Dim\ f, Shapely\ fs) \Rightarrow Align\ '[]\ (f\ ':fs) \qquad \textbf{where}\ ...$$

When applying a binary operator to two arrays, we align the two shapes with their least common extension, provided that this exists:

$$\textbf{type family}\ Max\ (fs :: [* \to *])\ (gs :: [* \to *]) :: [* \to *]$$
$$binary :: (Max\ fs\ gs \sim hs, Align\ fs\ hs, Align\ gs\ hs) \Rightarrow$$
$$\qquad (a \to b \to c) \to (Hyper\ fs\ a \to Hyper\ gs\ b \to Hyper\ hs\ c)$$
$$binary\ f\ x\ y = liftA2\ f\ (align\ x)\ (align\ y)$$

8. Better Representations

Manifest replication is wasteful; it can be represented symbolically instead, via an additional constructor:

$$PrismR :: (Dim\ f, Shapely\ fs) \Rightarrow Hyper\ fs\ (f\ a) \to Hyper\ (f\ ':fs)\ a$$

A similar technique can be used for transposition:

$$TransR :: (Dim\ f, Dim\ g, Shapely\ fs) \Rightarrow$$
$$\qquad Hyper\ (f\ ':g\ ':fs)\ a \to Hyper\ (g\ ':f\ ':fs)\ a$$

Array sizes are statically known, so we can represented a multidimensional structure as a flat sequence:

$$\textbf{data}\ Flat\ fs\ a\ \textbf{where}$$
$$\quad Flat :: Shapely\ fs \Rightarrow Array\ Int\ a \to Flat\ fs\ a$$

to which we can transform a hypercuboid:

$$flatten :: Shapely\ fs \Rightarrow Hyper\ fs\ a \to Flat\ fs\ a$$
$$flatten\ xs = Flat\ (listArray\ (0, sizeHyper\ xs - 1)\ (elements\ xs))$$
$$\quad \textbf{where}\ sizeHyper :: Shapely\ fs \Rightarrow Hyper\ fs\ a \to Int$$
$$\qquad elements :: Shapely\ fs \Rightarrow Hyper\ fs\ a \to [a]$$

References

[1] Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.

[2] Jsoftware, Inc. Jsoftware: High performance development platform. `http://www.jsoftware.com`, 2016.

[3] Gabrielle Keller, Manuel Chakravarty, Roman Leschchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic parallel arrays in Haskell. In *ICFP*, pages 261–272, 2010.

[4] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed Haskell programming. In *Haskell Symposium*, pages 81–92. ACM, 2013.

[5] Conor McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Prog.*, 12(4-5):375–392, 2002.

[6] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Prog.*, 18(1):1–13, 2008.

[7] Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In Zhong Shao, editor, *ESOP*, volume 8410 of *LNCS*, pages 27–46. Springer, 2014.

[8] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *TLDI*, pages 53–66. ACM, 2012.