APLicative Programming with Naperian Functors

Jeremy Gibbons

University of Oxford

Abstract. Much of the expressive power of array-oriented languages such as Iverson's APL and J comes from their implicit lifting of scalar operations to act on higher-ranked data, for example to add a value to each element of a vector, or to add two compatible matrices pointwise. It is considered a shape error to attempt to combine arguments of incompatible shape, such as a 3-vector with a 4-vector. APL and J are dynamically typed, so such shape errors are caught only at run-time. Recent work by Slepak *et al.* develops a custom type system for an array-oriented language, statically ruling out such errors. We show here that such a custom language design is unnecessary: the requisite compatibility checks can already be captured in modern expressive type systems, as found for example in Haskell; moreover, generative type-driven programming can exploit that static type information constructively to automatically induce the appropriate liftings. We show also that the structure of multi-dimensional data is inherently a matter of *Naperian applicative functors*—lax monoidal functors, with strength, commutative up to isomorphism under composition—that also support *traversal*.

1 Introduction

Array-oriented programming languages such as APL [21] and J [23] pay special attention, not surprisingly, to manipulating *array structures*. These encompass not just rank-one vectors (sequences of values), but also 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* [34]—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:

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

but also pointwise on vectors:

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

and on matrices and cuboids:

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

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

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

and on matrices:

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

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:

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

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

In the right-hand examples above, *sum* and *sums* have been lifted to act on the rows of the matrix. J also provides a *reranking* operator $"_1$, which will make them act instead on the columns—essentially a matter of matrix transposition:

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

$$\begin{aligned} sums\ "_1\ \boxed{\begin{matrix}1&2&3\\4&5&6\end{matrix}} &= transpose\left(sums\left(transpose\ \boxed{\begin{matrix}1&2&3\\4&5&6\end{matrix}}\right)\right)\\ &= transpose\left(sums\ \boxed{\begin{matrix}1&4\\2&5\\3&6\end{matrix}}\right) = transpose\ \boxed{\begin{matrix}1&5\\2&7\\3&9\end{matrix}} = \boxed{\begin{matrix}1&2&3\\5&7&9\end{matrix}} \end{aligned}$$

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

$$3 + \boxed{4\ 5\ 6} = \boxed{3\ 3\ 3} + \boxed{4\ 5\ 6} = \boxed{7\ 8\ 9}$$

or a vector and a matrix:

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

1.1 Static types for multi-dimensional arrays

In recent work [34], Slepak *et al.* 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, Remora's type system makes it a static error if the *shapes* in a given dimension do not match—for

example, when attempting to add a 2-vector to a 3-vector, or a 2×2-matrix to a 2×3-matrix. (Incidentally, we adopt Slepak *et al.*'s terminology: the *shape* of a multi-dimensional array is a sequence of numbers, specifying the extent in each dimension; the *rank* of the array is the length of that list, and hence the number of dimensions; and the *size* of the array is the product of that list, and hence the number of elements.)

Slepak *et al.* model the type and evaluation rules of Remora in PLT Redex [11], 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—with type inference, a compiler, libraries, efficient code generation, and so on. They write that "our hope [for future work] is that we can exploit this type information to compile programs written in the rank-polymorphic array computation model efficiently" and that "Remora is not intended as a language comfortable for human programmers to write array computations. It is, rather, an explicitly typed, 'essential' core language on which such a language could be based" [34, p29]. Moreover, "the transition from a core semantics modeled in PLT Redex to a complete programming system requires a more flexible surface language and a compiler [...] the added code is mostly type and index applications. Type inference would be necessary in order to make a surface language based on Remora practical" [34, p45].

1.2 Embedding static typing

This is the usual trade-off between standalone and embedded domain-specific languages. 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. Sometimes the representation comes entirely naturally; sometimes it takes some degree of encoding.

In this paper, we explore the embedded-DSL approach to capturing the type constraints and implicit lifting and alignment manipulations of rank-polymorphic array computation. We show how to capture these neatly in Haskell, a pure and strongly-typed functional programming language with growing abilities to express and exploit dependent types. To be more precise, we make use of a number of recent extensions to standard Haskell, which are supported in the primary implementation GHC [13]. We do not assume familiarity with fancy Haskell features, but explain them as we go along.

The point is not particularly to promote such fancy features; although the expressive power of modern type systems is quite impressive. Nor is the point to explain to aficionados of dependent types in Haskell how to perform rank-polymorphic array computation; most of our constructions are already folklore.

Rather, the point is to demonstrate to a wider programming language audience that it is often not necessary to invent a new special-purpose language in order to capture a sophisticated feature: we have sufficiently expressive general-purpose languages already.

1.3 The main idea

The main idea is that a rank-n array is essentially a data structure of type $D_1(D_2(\ldots(D_n\,a)))$, where each D_i is a *dimension*: a container type, categorically a functor; one might think in the first instance of lists. However, in order to be able to transpose arrays without losing information, each dimension should be represented by a functor of *fixed shape*; so perhaps vectors, of a fixed length in each dimension, but allowing different lengths in different dimensions.

The vector structure is sufficient to support all the necessary operations discussed above: mapping (for pointwise operations), zipping (for lifting binary operations), replicating (for alignment), transposing (for reranking), folding (for example, for *sum*), and traversing (for *sums*). Moreover, these can also be handled crisply, with static types that both prevent incoherent combinations and explain the implicit lifting required for compatible ones. However, although sufficient, the vector structure is not necessary, and other functors (such as pairs, triples, block vectors, and even perfect binary trees) suffice; we show that the necessary structure is that of a *traversable, Naperian, applicative functor* (and we explain what that means). The richer type structure that this makes available allows us to go beyond Remora, and in particular to explain the relationship between nested and flat representations of multi-dimensional data, leading the way towards higher-performance implementations of bulk operations, for example on multicore chips [24] and on GPUs [5].

Specifically, our novel contributions are as follows:

- formalizing the *lifting* required for rank polymorphism;
- doing so within an *existing type system*, rather than creating a new one;
- identifying *necessary and sufficient structure* for dimensions;
- *implementing* it all (in Haskell), and providing executable code;
- showing how to connect to *flat and sparse representations*.

Although our definitions are asymptotically efficient, or can easily be made so using standard techniques such as accumulating parameters, we do not make performance claims in comparison with serious array libraries such as Repa and Accelerate [24, 5]. Rather, we see this work as providing a flexible but safe front-end, delegating performance-critical computations to such libraries.

1.4 Structure of this paper

The remainder of this paper is structured as follows. Section 2 uses type-level natural numbers for bounds checking of vectors; Section 3 explains the requirements on vectors to support maps, zips, and transposition; and Section 4 similarly for

reductions and scans; these are all fairly standard material, and together show how to generalize the dimensions of an array from concrete vectors to other suitable types. Our contribution starts in Section 5, where we show how to accommodate arrays of arbitrary rank. Section 6 shows how to automatically lift unary and binary operators to higher ranks. Section 7 shows how to avoid manifesting replication and transposition operations by representing them symbolically instead, and Section 8 shows a more efficient representation using flat built-in arrays, while still preserving the shape information in the type. Section 9 concludes.

This paper is a literate Haskell script, and the code in it is all type-checked and executable, albeit with tidier formatting in the PDF for publication purposes. The extracted code is available for experimentation [14]. We exploit a number of advanced type features, explained as we proceed; but we make no use of laziness or undefinedness, treating Haskell as a total programming language.

2 Vectors with bounds checking

Our approach makes essential use of lightweight dependent typing, which is now becoming standard practice in modern functional programming languages such as Haskell. We introduce these ideas gradually, starting with traditional algebraic datatypes, such as lists:

$$\begin{aligned}&\textbf{data}\ \mathit{List} :: * \to * \ \textbf{where}\\ &\quad \mathit{Nil} \quad :: \qquad\qquad\ \ \mathit{List}\ a\\ &\quad \mathit{Cons} :: a \to \mathit{List}\ a \to \mathit{List}\ a\end{aligned}$$

This declaration defines a new datatype constructor *List* of kind $* \to *$. Which is to say, kind $*$ includes all those types with actual values, such as *Int* and *List Int* and $\mathit{Int} \to \mathit{Int}$; and *List* is an operation on types, such that for any type A of kind $*$, there is another type *List A* (also of kind $*$) of lists whose elements are drawn from A. The declaration also introduces two constructors *Nil* and *Cons* of the declared types for the new datatype, polymorphic in the element type.

All lists with elements of a given type have the same type; for example, there is one type *List Int* of lists of integers. This is convenient for operations that combine lists of different lengths; but it does not allow us to guarantee bounds safety by type checking. For example, the tail function

$$\begin{aligned}&\mathit{tail} :: \mathit{List}\ a \to \mathit{List}\ a\\ &\mathit{tail}\ (\mathit{Cons}\ x\ xs) = xs\end{aligned}$$

and the list indexing operator

$$\begin{aligned}&\mathit{lookup} :: \mathit{List}\ a \to \mathit{Int} \to a\\ &\mathit{lookup}\ (\mathit{Cons}\ x\ xs)\ 0 \qquad = x\\ &\mathit{lookup}\ (\mathit{Cons}\ x\ xs)\ (n+1) = \mathit{lookup}\ xs\ n\end{aligned}$$

are partial functions, and there is no way statically to distinguish their safe from their unsafe uses through the types. The way to achieve that end is to partition

the type *List A* into chunks, so that each chunk contains only the lists of a given length, and to index these chunks by their common lengths. The index should be another type parameter, just like the element type is; so we need a type-level way of representing natural numbers. One recent Haskell extension [39] has made this very convenient, by implicitly promoting all suitable datatype constructors from the value to the type level, and the datatypes themselves from the type level to the kind level. For example, from the familiar datatype of Peano naturals

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

we get not only a new type *Nat* with value inhabitants $Z, S\ Z, ...$, but in addition a new kind, also called *Nat*, with type inhabitants $'Z, 'S\ 'Z, ...$ In Haskell, the inhabitants can be distinguished by the initial quote character (which is in fact almost always optional, but for clarity we will make explicit use of it throughout this paper). For convenience, we define synonyms for some small numbers at the type level:

$$\begin{array}{l}\textbf{type}\ \mathit{One} \ \ = 'S\ 'Z\\ \textbf{type}\ \mathit{Two} \ \ = 'S\ \mathit{One}\\ \textbf{type}\ \mathit{Three} = 'S\ \mathit{Two}\\ \textbf{type}\ \mathit{Four} \ \ = 'S\ \mathit{Three}\end{array}$$

We can now define a datatype of length-indexed vectors:

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

The length is encoded in the type: *VNil* yields a vector of length zero, and *VCons* prefixes an element onto an n-vector to yield an $(n + 1)$-vector. For example, *Vector Three Int* is the type of 3-vectors of integers, one of whose inhabitants is the vector $\langle 1, 2, 3\rangle$:

$$\begin{array}{l}\mathit{v123} :: \mathit{Vector\ Three\ Int}\\ \mathit{v123} = \mathit{VCons}\ 1\ (\mathit{VCons}\ 2\ (\mathit{VCons}\ 3\ \mathit{VNil}))\end{array}$$

The first type parameter of *Vector* is called a 'phantom type' [19] or 'type index' [38], because it is not related to the type of any elements: a value of type *Vector Three Int* has elements of type *Int*, but does not in any sense 'have elements of type *Three*'. The type index does not interfere with ordinary recursive definitions, such as the mapping operation that applies a given function to every element, witnessing to *Vector n* being a functor:

$$\begin{array}{l}\mathit{vmap} :: (a \to b) \to \mathit{Vector}\ n\ a \to \mathit{Vector}\ n\ b\\ \mathit{vmap}\ f\ \mathit{VNil} \qquad\quad = \mathit{VNil}\\ \mathit{vmap}\ f\ (\mathit{VCons}\ x\ xs) = \mathit{VCons}\ (f\ x)\ (\mathit{vmap}\ f\ xs)\end{array}$$

For example,

$$\begin{array}{l}\mathit{v456} :: \mathit{Vector\ Three\ Int}\\ \mathit{v456} = \mathit{vmap}\ (\lambda x \to 3 + x)\ \mathit{v123}\end{array}$$

More interestingly, we can now capture the fact that the 'tail' function should be applied only to non-empty vectors, and that it yields a result one element shorter than its argument:

$$\begin{aligned}&vtail :: Vector\ (S\ n)\ a \to Vector\ n\ a\\&vtail\ (VCons\ x\ xs) = xs\end{aligned}$$

Similarly, we can write a 'zip' function that combines two vectors element-by-element using a binary operator, and use the additional type information to restrict it to take vectors of a common length n and to produce a result of the same length:

$$\begin{aligned}&vzipWith :: (a \to b \to c) \to Vector\ n\ a \to Vector\ n\ b \to Vector\ n\ c\\&vzipWith\ f\ VNil\ \ \ \ \ \ \ \ VNil\ \ \ \ \ \ \ \ = VNil\\&vzipWith\ f\ (VCons\ a\ x)\ (VCons\ b\ y) = VCons\ (f\ a\ b)\ (vzipWith\ f\ x\ y)\end{aligned}$$

Because of the type constraints, the patterns on the left-hand side in both examples are exhaustive: it would be ill-typed to take the tail of an empty vector, or to zip two vectors of different lengths.

The functions *vtail* and *vzipWith* consume vectors; the length indices constrain the behaviour, but they are not needed at run-time because the value constructors provide sufficient information to drive the computation. The situation is different when producing vectors from scratch. Consider a function *vreplicate* to construct a vector of a certain length by replicating a given value. The type $a \to Vector\ n\ a$ uniquely determines the implementation of such a function; however, it is the type of the result that contains the length information, and that isn't available for use at run-time. Nevertheless, for each n, there is an obvious implementation of *vreplicate* on $Vector\ n$; it would be nice to be able to state that obvious fact formally. In Haskell, this sort of 'type-driven code inference' is modelled by type classes—it is the same mechanism that determines the appropriate definition of equality or printing for a given type. Similarly, there is an obvious implementation of $vlength :: Vector\ n\ a \to Int$, which in fact does not even need to inspect its vector argument—the length is statically determined. We introduce the class *Count* of those types n (of kind *Nat*) that support these two 'obvious implementations':

$$\begin{aligned}&\textbf{class}\ Count\ (n :: Nat)\ \textbf{where}\\&\quad vreplicate :: a \to Vector\ n\ a\\&\quad vlength\ \ \ :: Vector\ n\ a \to Int\end{aligned}$$

Indeed, every type n of kind *Nat* is in the class *Count*, as we demonstrate by providing those two obvious implementations at each type n:

$$\begin{aligned}&\textbf{instance}\ Count\ 'Z\ \textbf{where}\\&\quad vreplicate\ a = VNil\\&\quad vlength\ xs\ \ = 0\\&\textbf{instance}\ Count\ n \Rightarrow Count\ ('S\ n)\ \textbf{where}\\&\quad vreplicate\ a = VCons\ a\ (vreplicate\ a)\\&\quad vlength\ xs\ \ = 1 + vlength\ (vtail\ xs)\end{aligned}$$

(One might see class *Count* as representing 'natural numbers specifically for vector purposes'; it is possible with some pain to represent 'natural numbers' in Haskell more generally [25].)

The operations *vmap*, *vzipWith*, and *vreplicate* are the essential ingredients for lifting and aligning operations to higher-ranked arguments (albeit not yet sufficient for the other operations). For example, to lift *square* to act on vectors, we can use *vmap square*; to lift (+) to act on two vectors of the same length, we can use *vzipWith* (+); and to align a scalar with a vector, we can use *vreplicate*:

$$v456 = vzipWith\ (+)\ (vreplicate\ 3)\ v123$$

(Note that the types of *vzipWith* and its second argument *v123* together determine which instance of *vreplicate* is required; so no explicit type annotation is needed.) But in order fully to achieve rank polymorphism, we want operators such as squaring and addition to implicitly determine the appropriate lifting and alignment, rather than having explicitly to specify the appropriate amount of replication. We see next how that can be done, without sacrificing static typing and type safety.

3 Applicative and Naperian functors

We have seen that vectors show promise for representing the dimensions of an array, because they support at least three of the essential operations, namely mapping, zipping, and replicating. But vectors are not the only datatype to support such operations; if we can identify the actual requirements on dimensions, then there are other types that would serve just as well. In particular, one of the dimensions of an array might be 'pairs':

$$\textbf{data}\ Pair\ a = P\ a\ a$$

since these too support the three operations discussed above:

$$\begin{aligned} &pmap &&:: (a \to b) \to Pair\ a \to Pair\ b \\ &pzipWith &&:: (a \to b \to c) \to Pair\ a \to Pair\ b \to Pair\ c \\ &preplicate &&:: a \to Pair\ a \end{aligned}$$

Generalizing in this way would allow us to handle vectors of pairs, pairs of triples, and so on.

The first requirement for a type constructor f to be suitable as a dimension is to be a container type, that is, an instance of the type class *Functor* and so providing an *fmap* operator:

$$\begin{aligned} &\textbf{class}\ Functor\ f\ \textbf{where} \\ &\quad fmap :: (a \to b) \to f\ a \to f\ b \end{aligned}$$

The other two operators arise from f being a fortiori an *applicative* functor [26]:

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

Informally, *pure* should yield an f-structure of copies of its argument; this serves as the 'replicate' operation:

$$\begin{aligned}&\mathit{areplicate} :: \mathit{Applicative}\ f \Rightarrow a \to f\ a\\&\mathit{areplicate} = \mathit{pure}\end{aligned}$$

(Here, the context "$\mathit{Applicative}\ f \Rightarrow$" denotes that *areplicate* has type $a \to f\ a$ for any f in type class *Applicative*; in contrast, the type variable a is unconstrained.) The $(\circledast)$ method should combine an f-structure of functions with an f-structure of arguments to yield an f-structure of results. The two methods together give rise to the 'zip' operation:

$$\begin{aligned}&\mathit{azipWith} :: \mathit{Applicative}\ f \Rightarrow (a \to b \to c) \to f\ a \to f\ b \to f\ c\\&\mathit{azipWith}\ h\ \mathit{xs}\ \mathit{ys} = (\mathit{pure}\ h \circledast \mathit{xs}) \circledast \mathit{ys}\end{aligned}$$

Vectors, of course, are applicative functors:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ (\mathit{Vector}\ n)\ \textbf{where}\\&\quad \mathit{fmap} = \mathit{vmap}\\&\textbf{instance}\ \mathit{Count}\ n \Rightarrow \mathit{Applicative}\ (\mathit{Vector}\ n)\ \textbf{where}\\&\quad \mathit{pure} = \mathit{vreplicate}\\&\quad (\circledast)\ = \mathit{vzipWith}\ (\lambda f\ x \to f\ x)\end{aligned}$$

Note that we make the assumption that the length index type n is in type class *Count*, so that we can infer the appropriate definition of *vreplicate*. This assumption is benign, because the length indices are of kind *Nat*, and we have provided a *Count* instance for every type of that kind.

Pairs too are applicative functors:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ \mathit{Pair}\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (P\ x\ y) = P\ (f\ x)\ (f\ y)\\&\textbf{instance}\ \mathit{Applicative}\ \mathit{Pair}\ \textbf{where}\\&\quad \mathit{pure}\ x\ \ \ \ \ \ \ \ = P\ x\ x\\&\quad P\ f\ g \circledast P\ x\ y = P\ (f\ x)\ (g\ y)\end{aligned}$$

However, being an applicative functor is not sufficient for serving as a dimension: that interface is not expressive enough to define transposition, which is needed in order to implement reranking. For that, we need to be able to commute the functors that represent dimensions: that is, to transform an $f\ (g\ a)$ into a $g\ (f\ a)$. The necessary additional structure is given by what Hancock [17] calls a *Naperian* functor, also known as a *representable* functor; that is, a container of fixed shape. Functor f is Naperian if there is a type p of 'positions' such that $f\ a \simeq p \to a$; then p behaves a little like a logarithm of f—in particular, if f and g are both Naperian, then $\mathit{Log}\ (f \times g) \simeq \mathit{Log}\ f + \mathit{Log}\ g$ and $\mathit{Log}\ (f \cdot g) \simeq \mathit{Log}\ f \times \mathit{Log}\ g$.

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Naperian}\ f\ \textbf{where}\\&\quad \textbf{type}\ \mathit{Log}\ f\\&\quad \mathit{lookup}\ \ \ \ :: f\ a \to (\mathit{Log}\ f \to a)\\&\quad \mathit{tabulate}\ \ \ :: (\mathit{Log}\ f \to a) \to f\ a\\&\quad \mathit{positions}\ :: f\ (\mathit{Log}\ f)\end{aligned}$$

$$\begin{aligned} &\mathit{tabulate}\ h = \mathit{fmap}\ h\ \mathit{positions} \\ &\mathit{positions}\ \ = \mathit{tabulate}\ \mathit{id} \end{aligned}$$

Informally, $\mathit{Log}\ f$ is the type of positions for f; $\mathit{lookup}\ \mathit{xs}\ i$ looks up the element of xs at position i; $\mathit{tabulate}\ h$ yields an f-structure where for each position i the element at that position is $h\ i$; and $\mathit{positions}$ yields an f-structure where the element at each position i is i itself. The first two operations should be each other's inverses; they are witnesses to the isomorphism between $f\ a$ and $\mathit{Log}\ f \to a$. The latter two operations are interdefinable, so an instance need only provide one of them; it is often convenient to implement $\mathit{positions}$, but to use $\mathit{tabulate}$. For simplicity, we rule out empty data structures, insisting that the type $\mathit{Log}\ f$ should always be inhabited. Naperian functors are necessarily applicative too:

$$\begin{aligned} &\mathit{pure}\ a = \mathit{tabulate}\ (\lambda i \to a) \\ &\mathit{fs} \circledast \mathit{xs} = \mathit{tabulate}\ (\lambda i \to (\mathit{lookup}\ \mathit{fs}\ i)\ (\mathit{lookup}\ \mathit{xs}\ i)) \end{aligned}$$

Transposition in general consumes an f-structure of g-structures in which all the g-structures have the same shape, and produces a g-structure of f-structures in which all the f-structures have the same shape, namely the outer shape of the input. For general functors f and g, this is a partial function, or at best a lossy one. However, the essential point about Naperian functors is that all inhabitants of a datatype have a common shape. In particular, in an f-structure of g-structures where both f and g are Naperian, all the inner g-structures necessarily have the same (namely, the only possible) shape. Then transposition is total and invertible:

$$\begin{aligned} &\mathit{transpose} :: (\mathit{Naperian}\ f, \mathit{Naperian}\ g) \Rightarrow f\ (g\ a) \to g\ (f\ a) \\ &\mathit{transpose} = \mathit{tabulate} \cdot \mathit{fmap}\ \mathit{tabulate} \cdot \mathit{flip} \cdot \mathit{fmap}\ \mathit{lookup} \cdot \mathit{lookup} \end{aligned}$$

Here, $\mathit{flip} :: (a \to b \to c) \to (b \to a \to c)$ is a standard function that swaps the argument order of a binary function. We use the lookup function for the outer and the inner structures of the input of type $f\ (g\ a)$, yielding a binary function of type $\mathit{Log}\ f \to \mathit{Log}\ g \to a$; we flip the arguments of this function, yielding one of type $\mathit{Log}\ g \to \mathit{Log}\ f \to a$; then we tabulate both structures again, yielding the result of type $g\ (f\ a)$ as required. For example, we have

$$\begin{aligned} \mathit{VCons}\ \mathit{v123}\ (\mathit{VCons}\ \mathit{v456}\ \mathit{VNil}) &= \langle\langle 1,2,3\rangle, \langle 4,5,6\rangle\rangle \\ \mathit{transpose}\ (\mathit{VCons}\ \mathit{v123}\ (\mathit{VCons}\ \mathit{v456}\ \mathit{VNil})) &= \langle\langle 1,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle\rangle \end{aligned}$$

As a consequence, composition of Naperian functors is commutative, up to isomorphism; we will insist on our dimensions being at least Naperian functors.

Of course, pairs are Naperian, with two positions—the usual ordering on booleans in Haskell has $\mathit{False} \leqslant \mathit{True}$, so we use this ordering on the positions too:

$$\begin{aligned} &\textbf{instance}\ \mathit{Naperian}\ \mathit{Pair}\ \textbf{where} \\ &\quad \textbf{type}\ \mathit{Log}\ \mathit{Pair}\ \ = \mathit{Bool} \\ &\quad \mathit{lookup}\ (P\ x\ y)\ b = \textbf{if}\ b\ \textbf{then}\ y\ \textbf{else}\ x \\ &\quad \mathit{positions}\ \ \ \ \ \ \ \ = P\ \mathit{False}\ \mathit{True} \end{aligned}$$

And vectors are Naperian. An n-vector has n positions, so to represent the logarithm we need a type with precisely n inhabitants—the *bounded naturals*:

```
data Fin :: Nat → ∗ where
  FZ ::           Fin ('S n)
  FS :: Fin n → Fin ('S n)
```

Thus, $Fin\ n$ has n inhabitants $FZ, FS\ FZ, ..., FS^{n-1}\ FZ$. Extracting an element from a vector is defined by structural induction simultaneously over the vector and the position—like with zipping, the type constraints make bounds violations a type error:

```
vlookup :: Vector n a → Fin n → a
vlookup (VCons a x) FZ     = a
vlookup (VCons a x) (FS n) = vlookup x n
```

A vector of positions is obtained by what in APL is called the 'iota' function. As with replication, we need to provide the length as a run-time argument; but we can represent this argument as a vector of units, and then infer the appropriate value from the type:

```
viota :: Count n ⇒ Vector n (Fin n)
viota = viota' (vreplicate ()) where
  viota' :: Vector m () → Vector m (Fin m)
  viota' VNil          = VNil
  viota' (VCons () xs) = VCons FZ (fmap FS (viota' xs))
```

With these three components, we are justified in calling vectors Naperian:

```
instance Count n ⇒ Naperian (Vector n) where
  type Log (Vector n) = Fin n
  lookup              = vlookup
  positions           = viota
```

4 Folding and traversing

Another requirement on the dimensions of an array is to be able to reduce along one of them; for example, to sum. In recent versions of Haskell, that requirement is captured in the *Foldable* type class, the essence of which is as follows:

```
class Foldable t where
  foldr :: (a → b → b) → b → t a → b
```

Informally, *foldr* aggregates the elements of a collection one by one, from right to left, using the binary operator and initial value provided. Vectors are foldable in the same way that lists are:

```
instance Foldable (Vector n) where
  foldr f e VNil          = e
  foldr f e (VCons x xs) = f x (foldr f e xs)
```

and pairs are foldable by combining their two elements:

$$\begin{array}{l}\textbf{instance}\ \mathit{Foldable}\ \mathit{Pair}\ \textbf{where}\\ \quad \mathit{foldr}\ f\ e\ (P\ x\ y) = f\ x\ (f\ y\ e)\end{array}$$

A foldable functor imposes a left-to-right ordering on its positions; so we can extract the elements as a list, in that order:

$$\begin{array}{l}\mathit{toList} :: \mathit{Foldable}\ t \Rightarrow t\ a \to [a]\\ \mathit{toList} = \mathit{foldr}\ (:)\ [\,]\end{array}$$

Similarly, we can sum those elements, provided that they are of a numeric type:

$$\begin{array}{l}\mathit{sum} :: (\mathit{Num}\ a, \mathit{Foldable}\ t) \Rightarrow t\ a \to a\\ \mathit{sum} = \mathit{foldr}\ (+)\ 0\end{array}$$

An additional requirement for array dimensions is to be able to *transform* values along a dimension, for example to compute prefix sums. This is captured by the *Traversable* type class:

$$\begin{array}{l}\textbf{class}\ (\mathit{Functor}\ t, \mathit{Foldable}\ t) \Rightarrow \mathit{Traversable}\ t\ \textbf{where}\\ \quad \mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b)\end{array}$$

One way of thinking of *traverse* is as an effectful 'map' function [3], visiting each element in order, precisely once each, and collecting effects in some applicative functor f. For example, stateful computations can be modelled by state-transforming functions:

$$\textbf{data}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \to (a, s)\}$$

(This construction declares $\mathit{State}\ s\ a$ to be a record type, with a data constructor also called *State*, and a single field called *runState*; in this way, the function *runState* extracts the state-transformer from the record.) This datatype forms an applicative functor, in a standard way. Here is a little function to increase and return a numeric state—whatever the current state n, when applied to m, this yields the final state $m + n$, and returns as result the same value $m + n$:

$$\begin{array}{l}\mathit{increase} :: \mathit{Num}\ a \Rightarrow a \to \mathit{State}\ a\ a\\ \mathit{increase}\ m = \mathit{State}\ (\lambda n \to (m + n, m + n))\end{array}$$

Using this, one can compute prefix sums by traversing a data structure, starting with an initial state of 0, increasing the state by each element in turn, preserving the running totals and discarding the final state:

$$\begin{array}{l}\mathit{sums} :: (\mathit{Num}\ a, \mathit{Traversable}\ t) \Rightarrow t\ a \to t\ a\\ \mathit{sums}\ xs = \mathit{fst}\ (\mathit{runState}\ (\mathit{traverse}\ \mathit{increase}\ xs)\ 0)\end{array}$$

so in particular

$$\mathit{sums}\ \mathit{v123} = \mathit{VCons}\ 1\ (\mathit{VCons}\ 3\ (\mathit{VCons}\ 6\ \mathit{VNil}))$$

Vectors and pairs are both traversable, with instances following a common pattern:

$$\begin{array}{l}\textbf{instance}\ \mathit{Traversable}\ \mathit{Pair}\ \textbf{where}\\ \quad \mathit{traverse}\ f\ (P\ x\ y) = (\mathit{pure}\ P \circledast f\ x) \circledast f\ y\end{array}$$

```
instance Traversable (Vector n) where
  traverse f  VNil        = pure VNil
  traverse f (VCons x xs) = (pure VCons ⊛ f x) ⊛ traverse f xs
```

We take these various constraints as our definition of 'dimension':

```
class (Applicative f, Naperian f, Traversable f) ⇒ Dimension f where
  size :: f a → Int
  size = length · toList
```

We have added a *size* method for convenience and with no loss of generality—it is in fact statically determined, so may admit better type-specific definitions:

```
instance                Dimension Pair       where size = const 2
instance Count n ⇒ Dimension (Vector n) where size = vlength
```

But other less obvious datatypes, such as perfect binary trees of a given height, are suitable dimensions too:

```
data Perfect :: Nat → ∗ → ∗ where
  Leaf :: a →                    Perfect 'Z a
  Bin  :: Pair (Perfect n a) → Perfect ('S n) a
```

For example, a *Perfect Three a* is essentially a *Pair* (*Pair* (*Pair a*)). Perhaps more usefully, rather than indexing vectors by a unary representation of the natural numbers, we can use a more compact binary representation:

```
data Binary :: ∗ where
  Unit     ::          Binary
  Twice    :: Binary → Binary
  Twice+1 :: Binary → Binary
```

under the obvious interpretation

```
bin2int :: Binary → Int
bin2int Unit         = 1
bin2int (Twice n)    = 2 × bin2int n
bin2int (Twice+1 n) = 2 × bin2int n + 1
```

Then we can define a datatype of (non-empty) vectors built up via balanced join rather than imbalanced cons:

```
data BVector :: Binary → ∗ → ∗ where
  VSingle  :: a →                                    BVector 'Unit a
  VJoin    ::      BVector n a → BVector n a → BVector ('Twice n) a
  VJoin+1 :: a → BVector n a → BVector n a → BVector ('Twice+1 n) a
```

When used as the dimensions of a matrix, this will allow a quad tree decomposition [12] for recursive functions. We leave the instance definitions as an exercise for the energetic reader.

In fact, one may start from any numeric representation and manufacture a corresponding datatype [29, 18]. Sandberg Eriksson and Jansson [31] use a redundant binary representation of the positive natural numbers (with constructors 1 and +) as the type index in a formalization of block matrices. Each

of these dimension types—pairs, triples, perfect binary trees of a given height, block vectors of a given structure—is equivalent to some vector type, so no additional expressivity is gained; but the alternatives may be more natural in given contexts.

As an example of a generic function, inner product involves summing pairwise products, and so works for any dimension type:

$$\begin{aligned}&\mathit{innerp} :: (\mathit{Num}\ a, \mathit{Dimension}\ f) \Rightarrow f\ a \to f\ a \to a\\&\mathit{innerp}\ \mathit{xs}\ \mathit{ys} = \mathit{sum}\ (\mathit{azipWith}\ (*)\ \mathit{xs}\ \mathit{ys})\end{aligned}$$

Multiplying an $f \times g$-matrix by a $g \times h$-matrix entails lifting both to $f \times h \times g$-matrices then performing pairwise inner product on the g-vectors:

$$\begin{aligned}&\mathit{matrixp} :: (\mathit{Num}\ a, \mathit{Dimension}\ f, \mathit{Dimension}\ g, \mathit{Dimension}\ h) \Rightarrow\\&\qquad f\ (g\ a) \to g\ (h\ a) \to f\ (h\ a)\\&\mathit{matrixp}\ \mathit{xss}\ \mathit{yss} = \mathit{azipWith}\ (\mathit{azipWith}\ \mathit{innerp})\ (\mathit{fmap}\ \mathit{areplicate}\ \mathit{xss})\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{areplicate}\ (\mathit{transpose}\ \mathit{yss}))\end{aligned}$$

Again, this works for any dimension types f, g, h; the *same* definition works for vectors, pairs, block vectors, and any mixture of these.

5 Multidimensionality

Now that we can represent vectors with elements of an arbitrary type, we can of course represent matrices too, as vectors of vectors:

$$\begin{aligned}&\mathit{vv123456} :: \mathit{Vector}\ \mathit{Two}\ (\mathit{Vector}\ \mathit{Three}\ \mathit{Int})\\&\mathit{vv123456} = \mathit{VCons}\ \mathit{v123}\ (\mathit{VCons}\ \mathit{v456}\ \mathit{VNil})\end{aligned}$$

However, with this representation, integer vectors and integer matrices are of quite different types, and there is no immediate prospect of supporting rank polymorphism over them—for example, a single operation that can both add two matrices and add a vector to a matrix. In order to do that, we need one datatype that encompasses both vectors and matrices (and scalars, and arrays of higher rank).

One way to achieve this goal is with a *nested* [4] or *polymorphically recursive* [28] datatype:

$$\begin{aligned}&\mathbf{data}\ \mathit{Hyper}_0 :: * \to * \ \mathbf{where}\quad \text{-- to be refined later}\\&\quad \mathit{Scalar}_0 :: a \to \mathit{Hyper}_0\ a\\&\quad \mathit{Prism}_0 :: \mathit{Count}\ n \Rightarrow \mathit{Hyper}_0\ (\mathit{Vector}\ n\ a) \to \mathit{Hyper}_0\ a\end{aligned}$$

(we make a convention of subscripting definitions that will be refined later). This datatype corresponds to APL's multi-dimensional arrays. We use the name Hyper_0, for 'hypercuboid', so as not to clash with Haskell's *Array* type that we will use later. Thus, Scalar_0 constructs a scalar hypercuboid from its sole element; and Prism_0 yields a hypercuboid of rank $r + 1$ from a hypercuboid of rank r whose elements are all n-vectors (for some n, but crucially, the same n for all elements at this rank). This definition makes essential use of polymorphic recursion, because a composite hypercuboid of as is constructed inductively not

from smaller hypercuboids of as, but from a (single) hypercuboid of vectors of as.

This datatype satisfies the requirement of encompassing hypercuboids of arbitrary rank. However, it is somewhat unsatisfactory, precisely because it lumps together all hypercuboids of a given element type into a single type; for example, a vector and a matrix of integers both have the same type, namely $Hyper_0\ Int$. We have sacrificed any ability to catch rank errors through type checking. Perhaps worse, we have also sacrificed any chance to use the rank statically in order to automatically lift operators. We can solve this problem in much the same way as we did for bounds checking of vectors, by specifying the rank as a type index:

$$\begin{array}{l}\textbf{data}\ Hyper_1 :: Nat \to * \to * \ \textbf{where} \quad \text{-- to be refined later} \\ \quad Scalar_1 :: \qquad a \to \qquad\qquad\qquad\qquad Hyper_1\ 'Z\ a \\ \quad Prism_1 :: Count\ n \Rightarrow Hyper_1\ r\ (Vector\ n\ a) \to Hyper_1\ ('S\ r)\ a\end{array}$$

Now a vector of integers has type $Hyper_1\ One\ Int$, and a matrix of integers has type $Hyper_1\ Two\ Int$; it is a type error simply to try to add them pointwise, and the rank index can be used (we will see how in due course) to lift addition to act appropriately.

That is all well and good for rank, but we have a similar problem with size too: a 3-vector and a 4-vector of integers both have the same type when viewed as hypercuboids, namely $Hyper_1\ One\ Int$; so we can no longer catch size mismatches by type checking. Apparently indexing by the rank alone is not enough; we should index by the size in each dimension—a list of natural numbers. Then the rank is the length of this list. Just as in Section 2 we promoted the datatype Nat to a kind and its inhabitants $Z, S\ Z, ...$ to types $'Z, 'S\ 'Z, ...$, we can also promote the datatype $[\,]$ of lists to the kind level, and its constructors $[\,]$ and $(:)$ to operators $'[\,]$ and $(':)$ at the type level:

$$\begin{array}{l}\textbf{data}\ Hyper_2 :: [Nat] \to * \to * \ \textbf{where} \quad \text{-- to be refined later} \\ \quad Scalar_2 :: \qquad a \to \qquad\qquad\qquad\qquad Hyper_2\ '[\,]\ a \\ \quad Prism_2 :: Count\ n \Rightarrow Hyper_2\ ns\ (Vector\ n\ a) \to Hyper_2\ (n\ ':\ ns)\ a\end{array}$$

Now a 3-vector of integers has type $Hyper_2\ '[Three]\ Int$, a 4-vector has type $Hyper_2\ '[Four]\ Int$, a 2×3-matrix has type $Hyper_2\ '[Three, Two]\ Int$, and so on. (Note that the latter is essentially a 2-vector of 3-vectors, rather than the other way round; it turns out to be most convenient for the first element of the list to represent the extent of the *innermost* dimension.) There is enough information at the type level to catch mismatches both of rank and of size; but still, the indexed types are all members of a common datatype, so can be made to support common operations.

That deals with multi-dimensional vectors. But as we discussed in Section 3, there is no a priori reason to restrict each dimension to be a vector; other datatypes work too, provided that they are instances of the type class $Dimension$. Then it is not enough for the datatype of hypercuboids to be indexed by a type-level list of lengths, because the lengths are no longer sufficient to characterize the dimensions—instead, we should use a type-level list of the dimension types themselves.

We call these type-level lists of dimension types *shapely* [22]. Following the example of vectors in Section 2, we introduce a type class of shapely types, which support replication and size:

```
class Shapely fs where
  hreplicate :: a → Hyper fs a
  hsize      :: Hyper fs a → Int
```

We ensure that every possible type-level list of dimensions is an instance:

```
instance Shapely '[] where
  hreplicate a      = Scalar a
  hsize             = const 1
instance (Dimension f, Shapely fs) ⇒ Shapely (f ': fs) where
  hreplicate a      = Prism (hreplicate (areplicate a))
  hsize (Prism x) = size (first x) × hsize x
```

Here, *first* returns the first element of a hypercuboid, so *first x* is the first 'row' of *Prism x*:

```
first :: Shapely fs ⇒ Hyper fs a → a
first (Scalar a) = a
first (Prism x) = head (toList (first x))
```

and the size of a hypercuboid is of course the product of the lengths of its dimensions.

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.

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

For example, we can wrap up a vector of vectors as a rank-2 hypercuboid:

```
h123456 :: Hyper '[Vector Three, Vector Two] Int
h123456 = Prism (Prism (Scalar vv123456))
```

Hypercuboids are of course functorial:

```
instance Functor (Hyper fs) where
  fmap f (Scalar a) = Scalar (f a)
  fmap f (Prism x) = Prism (fmap (fmap f) x)
```

Furthermore, they are applicative; the type class *Shapely* handles replication, and zipping is simply a matter of matching structures:

```
hzipWith :: (a → b → c) → Hyper fs a → Hyper fs b → Hyper fs c
hzipWith f (Scalar a) (Scalar b) = Scalar (f a b)
hzipWith f (Prism x) (Prism y) = Prism (hzipWith (azipWith f) x y)
```

With these two, we can install shapely hypercuboids as an applicative functor:

```
instance Shapely fs ⇒ Applicative (Hyper fs) where
  pure = hreplicate
  (⊛)  = hzipWith (λf x → f x)
```

(In fact, hypercuboids are also Naperian, foldable, and traversable too, so they can themselves serve as dimensions; but we do not need that power in the rest of this paper.)

Now we can fold along the 'major' (that is, the innermost) axis of a hypercuboid, given a suitable binary operator and initial value:

```
reduceBy :: (a → a → a, a) → Hyper (f ': fs) a → Hyper fs a
reduceBy (f, e) (Prism x) = fmap (foldr f e) x
```

Moreover, we can transpose the hypercuboid in order to be able to fold along the 'minor' (that is, the next-to-innermost) axis:

```
transposeHyper :: Hyper (f ': (g ': fs)) a → Hyper (g ': (f ': fs)) a
transposeHyper (Prism (Prism x)) = Prism (Prism (fmap transpose x))
```

Thus, given a hypercuboid of type $Hyper\ (f\ ':\ (g\ ':\ fs))\ a$, which by construction must be of the form $Prism\ (Prism\ x)$ with x of type $Hyper\ fs\ (g\ (f\ a))$, we *transpose* each of the inner hypercuboids from $g\ (f\ a)$ to $f\ (g\ a)$, then put the two *Prism* constructors back on to yield the result of type $Hyper\ (g\ ':\ (f\ ':\ fs))\ a$ as required. And with multiple transpositions, we can rearrange a hypercuboid to bring any axis into the 'major' position.

6 Alignment

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

```
unary :: Shapely fs ⇒ (a → b) → (Hyper fs a → Hyper fs b)
unary = fmap
```

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

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

Scalar shapes are alignable with each other; alignment is the identity function:

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

Alignments can be extended along a common inner dimension:

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

Finally, and most importantly, a scalar can be aligned with an arbitrary hypercuboid, via replication:

$$\textbf{instance}\ (\mathit{Dimension}\ f, \mathit{Shapely}\ fs) \Rightarrow \mathit{Alignable}\ '[]\ (f\ ':\ fs)\ \textbf{where}$$
$$\quad \mathit{align}\ (\mathit{Scalar}\ a) = \mathit{hreplicate}\ a$$

(Note that, ignoring the accompanying definitions of the *align* function, the heads of the three *Alignable* instance declarations can be read together as a logic program for when one sequence is a prefix of another.)

The *Alignable* relation on shapes is an ordering, and in particular asymmetric. In order to be able to lift a binary operator to act on two compatible hypercuboids, we should treat the two arguments symmetrically: we will align the two shapes with their least common extension, provided that this exists. We express that in terms of the *Max* of two shapes, a type-level function:

$$\begin{aligned}&\textbf{type family}\ \mathit{Max}\ (fs :: [* \to *])\ (gs :: [* \to *]) :: [* \to *]\ \textbf{where}\\ &\quad \mathit{Max}\ '[]\ \quad '[]\ \quad = '[]\\ &\quad \mathit{Max}\ '[]\ \quad (f\ ':\ gs) = (f\ ':\ gs)\\ &\quad \mathit{Max}\ (f\ ':\ fs)\ '[]\ \quad = (f\ ':\ fs)\\ &\quad \mathit{Max}\ (f\ ':\ fs)\ (f\ ':\ gs) = (f\ ':\ \mathit{Max}\ fs\ gs)\end{aligned}$$

For example, a 2×3-matrix can be aligned with a 3-vector:

$$\mathit{Max}\ '[\mathit{Three}, \mathit{Two}]\ '[\mathit{Three}]\ \sim\ '[\mathit{Three}, \mathit{Two}]$$

Here, $\sim$ denotes type compatibility in Haskell. Provided that shapes *fs* and *gs* are compatible, we can align two hypercuboids of those shapes with their least common extension *hs*, and then apply a binary operator to them pointwise:

$$\begin{aligned}&\mathit{binary}_0 :: \quad \text{-- to be refined later}\\ &\quad (\mathit{Max}\ fs\ gs \sim hs, \mathit{Alignable}\ fs\ hs, \mathit{Alignable}\ gs\ hs) \Rightarrow\\ &\quad (a \to b \to c) \to (\mathit{Hyper}\ fs\ a \to \mathit{Hyper}\ gs\ b \to \mathit{Hyper}\ hs\ c)\\ &\mathit{binary}_0\ f\ x\ y = \mathit{hzipWith}\ f\ (\mathit{align}\ x)\ (\mathit{align}\ y)\end{aligned}$$

For example,

$$\mathit{binary}_0\ (+)\ (\mathit{Scalar}\ 3)\ \mathit{h123456} = \langle\langle 4,5,6\rangle, \langle 7,8,9\rangle\rangle$$

Note that as a function on types, *Max* is partial: two shapes $f\ ':\ fs$ and $g\ ':\ gs$ are incompatible when $f \not\equiv g$, and then have no common extension. In that case, it is a type error to attempt to align two hypercuboids of those shapes. However, the type error can be a bit inscrutable. For example, when trying to align a 3-vector with a 4-vector, the compiler cannot simplify $\mathit{Max}\ '[\mathit{Vector}\ \mathit{Three}]\ '[\mathit{Vector}\ \mathit{Four}]$, and GHC (version 7.10.3) gives the following error:

```
No instance for
  (Alignable '[Vector Three] (Max '[Vector Three] '[Vector Four]))
  (maybe you haven't applied enough arguments to a function?)
```

We can use type-level functions to provide more helpful error messages too [33]. We define an additional type function as a predicate on types, to test whether the shapes are compatible:

$$\begin{aligned}&\textbf{type family}\ \mathit{IsCompatible}\ (fs :: [* \to *])\ (gs :: [* \to *]) :: \mathit{IsDefined}\ \mathit{Symbol}\ \textbf{where}\\ &\quad \mathit{IsCompatible}\ '[]\ \quad '[]\ \quad = \mathit{Defined}\\ &\quad \mathit{IsCompatible}\ '[]\ \quad (f\ ':\ gs) = \mathit{Defined}\end{aligned}$$

$$\begin{aligned}
&\mathit{IsCompatible}\ (f\ ' \colon fs)\ '[\,] &&= \mathit{Defined}\\
&\mathit{IsCompatible}\ (f\ ' \colon fs)\ (f\ ' \colon gs) &&= \mathit{IsCompatible}\ fs\ gs\\
&\mathit{IsCompatible}\ (f\ ' \colon fs)\ (g\ ' \colon gs) &&= \mathit{Undefined}\ \texttt{"Mismatching dimensions"}
\end{aligned}$$

Here, *Symbol* is the kind of type-level strings, and *IsDefined* is a type-level version of the booleans, but extended to incorporate also an explanation in the case that the predicate fails to hold:

$$\textbf{data}\ \mathit{IsDefined}\ e = \mathit{Defined} \mid \mathit{Undefined}\ e$$

If we now add this test as a constraint to the type of a lifted binary operator:

$$\begin{aligned}
&\mathit{binary} :: \quad \text{-- final version}\\
&\quad (\mathit{IsCompatible}\ fs\ gs \sim \mathit{Defined}, \mathit{Max}\ fs\ gs \sim hs, \mathit{Alignable}\ fs\ hs, \mathit{Alignable}\ gs\ hs) \Rightarrow\\
&\quad (a \to b \to c) \to (\mathit{Hyper}\ fs\ a \to \mathit{Hyper}\ gs\ b \to \mathit{Hyper}\ hs\ c)\\
&\mathit{binary}\ f\ x\ y = \mathit{binary}_0\ f\ x\ y
\end{aligned}$$

(note that the code is precisely the same, only the type has become more informative) then we get a slightly more helpful error message when things go wrong:

```
Couldn't match type 'Undefined "Mismatching dimensions"
                    with 'Defined
Expected type: 'Defined
  Actual type: IsCompatible '[Vector Three] '[Vector Four]
```

7 Symbolic transformations

Although alignment of arrays of compatible but different shapes morally entails replication, this is an inefficient way actually to implement it; instead, it is better simply to use each element of the smaller structure multiple times. One way to achieve this is perform the replication *symbolically*—that is, to indicate via the type index that an array is replicated along a given dimension, without manifestly performing the replication. This can be achieved by extending the datatype of hypercuboids to incorporate an additional constructor:

$$\begin{aligned}
&\textbf{data}\ \mathit{HyperR} :: [* \to *] \to * \to * \ \textbf{where}\\
&\quad \mathit{ScalarR} :: \qquad\qquad\qquad\qquad a \to \qquad\qquad\quad \mathit{HyperR}\ '[\,]\ a\\
&\quad \mathit{PrismR} :: (\mathit{Dimension}\ f, \mathit{Shapely}\ fs) \Rightarrow \mathit{HyperR}\ fs\ (f\ a) \to \mathit{HyperR}\ (f\ ' \colon fs)\ a\\
&\quad \mathit{ReplR} \quad :: (\mathit{Dimension}\ f, \mathit{Shapely}\ fs) \Rightarrow \mathit{HyperR}\ fs\ a \to \quad\ \mathit{HyperR}\ (f\ ' \colon fs)\ a
\end{aligned}$$

The idea is that *ReplR x* denotes the same array as *Prism* (*fmap areplicate x*), but takes constant time and space to record the replication. It allows us to implement replication to multiple ranks in time and space proportional to the rank, rather than to the size. This would be of no benefit were it just to postpone the actual replication work until later. Fortunately, the work can often be avoided altogether. Mapping is straightforward, since it simply distributes through *ReplR*:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ (\mathit{HyperR}\ fs)\ \textbf{where}\\
&\quad \mathit{fmap}\ f\ (\mathit{ScalarR}\ a) = \mathit{ScalarR}\ (f\ a)
\end{aligned}$$

$$\begin{aligned}
&\mathit{fmap}\ f\ (\mathit{PrismR}\ x) = \mathit{PrismR}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ x)\\
&\mathit{fmap}\ f\ (\mathit{ReplR}\ x) \quad = \mathit{ReplR}\ (\mathit{fmap}\ f\ x)
\end{aligned}$$

Similarly for zipping two replicated dimensions. When zipping a replicated dimension (*ReplR*) with a manifest one (*PrismR*), we end up essentially with a map—that, after all, was the whole point of the exercise. The other cases are as before.

$$\begin{aligned}
&\mathit{rzipWith} :: \mathit{Shapely}\ \mathit{fs} \Rightarrow (a \to b \to c) \to \mathit{HyperR}\ \mathit{fs}\ a \to \mathit{HyperR}\ \mathit{fs}\ b \to \mathit{HyperR}\ \mathit{fs}\ c\\
&\mathit{rzipWith}\ f\ (\mathit{ScalarR}\ a)\ (\mathit{ScalarR}\ b) = \mathit{ScalarR}\ (f\ a\ b)\\
&\mathit{rzipWith}\ f\ (\mathit{PrismR}\ x)\ (\mathit{PrismR}\ y) = \mathit{PrismR}\ (\mathit{rzipWith}\ (\mathit{azipWith}\ f)\ x\ y)\\
&\mathit{rzipWith}\ f\ (\mathit{PrismR}\ x)\ (\mathit{ReplR}\ y) \quad = \mathit{PrismR}\ (\mathit{rzipWith}\ (\mathit{azipWithL}\ f)\ x\ y)\\
&\mathit{rzipWith}\ f\ (\mathit{ReplR}\ x)\ (\mathit{PrismR}\ y) \quad = \mathit{PrismR}\ (\mathit{rzipWith}\ (\mathit{azipWithR}\ f)\ x\ y)\\
&\mathit{rzipWith}\ f\ (\mathit{ReplR}\ x)\ (\mathit{ReplR}\ y) \quad\ = \mathit{ReplR}\ (\mathit{rzipWith}\ f\ x\ y)
\end{aligned}$$

Here, *azipWithL* and *azipWithR* are variants of *azipWith* with one argument constant:

$$\begin{aligned}
&\mathit{azipWithL} :: \mathit{Functor}\ f \Rightarrow (a \to b \to c) \to f\ a \to b \to f\ c\\
&\mathit{azipWithL}\ f\ \mathit{xs}\ y = \mathit{fmap}\ (\lambda x \to f\ x\ y)\ \mathit{xs}\\
&\mathit{azipWithR} :: \mathit{Functor}\ f \Rightarrow (a \to b \to c) \to a \to f\ b \to f\ c\\
&\mathit{azipWithR}\ f\ x\ \mathit{ys} = \mathit{fmap}\ (\lambda y \to f\ x\ y)\ \mathit{ys}
\end{aligned}$$

(note that they only need a *Functor* constraint rather than *Applicative*, since they only use *fmap* and not *pure* and $\circledast$).

Similarly for transposition; if either of the innermost two dimensions is symbolically replicated, it is just a matter of rearranging constructors, and only when they are both manifest do we have to resort to actual data movement:

$$\begin{aligned}
&\mathit{rtranspose} :: (\mathit{Shapely}\ \mathit{fs}, \mathit{Dimension}\ f, \mathit{Dimension}\ g) \Rightarrow\\
&\qquad\qquad \mathit{HyperR}\ (f\ ' : g\ ' : \mathit{fs})\ a \to \mathit{HyperR}\ (g\ ' : f\ ' : \mathit{fs})\ a\\
&\mathit{rtranspose}\ (\mathit{PrismR}\ (\mathit{PrismR}\ x)) = \mathit{PrismR}\ (\mathit{PrismR}\ (\mathit{fmap}\ \mathit{transpose}\ x))\\
&\mathit{rtranspose}\ (\mathit{PrismR}\ (\mathit{ReplR}\ x)) \quad = \mathit{ReplR}\ (\mathit{PrismR}\ x)\\
&\mathit{rtranspose}\ (\mathit{ReplR}\ (\mathit{PrismR}\ x)) \quad = \mathit{PrismR}\ (\mathit{ReplR}\ x)\\
&\mathit{rtranspose}\ (\mathit{ReplR}\ (\mathit{ReplR}\ x)) \quad\ = \mathit{ReplR}\ (\mathit{ReplR}\ x)
\end{aligned}$$

It is only when it comes to folding or traversing a hypercuboid that a symbolic replication really has to be forced. This can be achieved by means of a function that expands a top-level *ReplR* constructor, if one is present, while leaving the hypercuboid abstractly the same:

$$\begin{aligned}
&\mathit{forceReplR} :: \mathit{Shapely}\ \mathit{fs} \Rightarrow \mathit{HyperR}\ \mathit{fs}\ a \to \mathit{HyperR}\ \mathit{fs}\ a\\
&\mathit{forceReplR}\ (\mathit{ReplR}\ x) = \mathit{PrismR}\ (\mathit{fmap}\ \mathit{areplicate}\ x)\\
&\mathit{forceReplR}\ x \qquad\qquad\ = x
\end{aligned}$$

A similar technique can be used to represent transposition itself symbolically, via its own constructor:

$$\begin{aligned}
&\textbf{data}\ \mathit{HyperT} :: [* \to *] \to * \to * \ \textbf{where}\\
&\quad \mathit{ScalarT} :: a \to \qquad\qquad\qquad\quad \mathit{HyperT}\ '[]\ a\\
&\quad \mathit{PrismT} :: (\mathit{Dimension}\ f, \mathit{Shapely}\ \mathit{fs}) \Rightarrow\\
&\qquad\qquad \mathit{HyperT}\ \mathit{fs}\ (f\ a) \to \qquad \mathit{HyperT}\ (f\ ' : \mathit{fs})\ a
\end{aligned}$$

$$\begin{aligned}&\mathit{TransT} :: (\mathit{Dimension}\ f, \mathit{Dimension}\ g, \mathit{Shapely}\ \mathit{fs}) \Rightarrow \\ &\qquad \mathit{HyperT}\ (f' : g' : \mathit{fs})\ a \to \mathit{HyperT}\ (g' : f' : \mathit{fs})\ a\end{aligned}$$

The idea is that $\mathit{TransT}\ x$ represents the transposition of x, without actually doing any work. We can maintain the invariant that there are never two adjacent TransT constructors, by using the following 'smart constructor' in place of the real one, to remove a transposition if one is present and to add one otherwise:

$$\begin{aligned}&\mathit{transT} :: (\mathit{Dimension}\ f, \mathit{Dimension}\ g, \mathit{Shapely}\ \mathit{fs}) \Rightarrow \\ &\qquad \mathit{HyperT}\ (f' : g' : \mathit{fs})\ a \to \mathit{HyperT}\ (g' : f' : \mathit{fs})\ a \\ &\mathit{transT}\ (\mathit{TransT}\ x) = x \\ &\mathit{transT}\ x = \mathit{TransT}\ x\end{aligned}$$

Of course, with the help of this additional constructor, transposition is trivial, and replication is no more difficult than it was with plain *Hyper*; zipping is the only operation that requires any thought. Where the two structures match, zipping simply commutes with them—and in particular, symbolic transpositions may be preserved, as in the third equation for $\mathit{tzipWith}$ below. Only when zipping a TransT with a PrismT does the symbolic transposition need to be forced, for which we provide a function that expands a top-most TransT constructor if one is present, while leaving the hypercuboid abstractly the same:

$$\begin{aligned}&\mathit{forceTransT} :: (\mathit{Dimension}\ f, \mathit{Dimension}\ g, \mathit{Shapely}\ \mathit{fs}) \Rightarrow \\ &\qquad \mathit{HyperT}\ (f' : g' : \mathit{fs})\ a \to \mathit{HyperT}\ (f' : g' : \mathit{fs})\ a \\ &\mathit{forceTransT}\ (\mathit{TransT}\ (\mathit{PrismT}\ (\mathit{PrismT}\ x))) \\ &\qquad = \mathit{PrismT}\ (\mathit{PrismT}\ (\mathit{fmap}\ \mathit{transpose}\ x)) \\ &\mathit{forceTransT}\ (\mathit{TransT}\ (\mathit{PrismT}\ x@(\mathit{TransT}\ \_))) \\ &\qquad = \mathbf{case}\ \mathit{forceTransT}\ x\ \mathbf{of} \\ &\qquad\quad \mathit{PrismT}\ x' \to \mathit{PrismT}\ (\mathit{PrismT}\ (\mathit{fmap}\ \mathit{transpose}\ x')) \\ &\mathit{forceTransT}\ x = x\end{aligned}$$

(Here, the 'as-pattern' $x@p$ binds x to the whole of an argument whilst simultaneously matching against the pattern p, and $\_$ is a wild card. On account of the type constraints, together with the invariant that there are no two adjacent TransT constructors, these three clauses are sufficient to guarantee that the outermost constructor is not a TransT.) Then we have:

$$\begin{aligned}&\mathit{tzipWith} :: \mathit{Shapely}\ \mathit{fs} \Rightarrow \\ &\qquad (a \to b \to c) \to \mathit{HyperT}\ \mathit{fs}\ a \to \mathit{HyperT}\ \mathit{fs}\ b \to \mathit{HyperT}\ \mathit{fs}\ c \\ &\mathit{tzipWith}\ f\ (\mathit{ScalarT}\ a)\ (\mathit{ScalarT}\ b) \quad = \mathit{ScalarT}\ (f\ a\ b) \\ &\mathit{tzipWith}\ f\ (\mathit{PrismT}\ x)\ (\mathit{PrismT}\ y) \quad = \mathit{PrismT}\ (\mathit{tzipWith}\ (\mathit{azipWith}\ f)\ x\ y) \\ &\mathit{tzipWith}\ f\ (\mathit{TransT}\ x)\ (\mathit{TransT}\ y) \quad = \mathit{TransT}\ (\mathit{tzipWith}\ f\ x\ y) \\ &\mathit{tzipWith}\ f\ x@(\mathit{TransT}\ \_)\ (\mathit{PrismT}\ y) = \mathit{tzipWith}\ f\ (\mathit{forceTransT}\ x)\ (\mathit{PrismT}\ y) \\ &\mathit{tzipWith}\ f\ (\mathit{PrismT}\ x)\ y@(\mathit{TransT}\ \_) = \mathit{tzipWith}\ f\ (\mathit{PrismT}\ x)\ (\mathit{forceTransT}\ y)\end{aligned}$$

Again, folding and traversing seem to require manifesting any symbolic transpositions.

We can even combine symbolic replication and transposition in the same datatype, providing trivial implementations of both operations. The only tricky

part then is in zipping, while preserving as much of the symbolic representation as possible. We have all the cases of *rzipWith* for prisms interacting with replication, plus those of *tzipWith* for prisms interacting with transposition, plus some new cases for replication interacting with transposition. The details are not particularly surprising, so are left again to the energetic reader.

8 Flat representation

The various nested representations above precisely capture the shape of a hypercuboid. This prevents dimension and size mismatches, by making them type errors; more constructively, it drives the mechanism for automatically aligning the arguments of heterogeneous binary operators. However, the nested representation is inefficient in time and space; high performance array libraries targetting GPUs arrange the data as a simple, flat, contiguous sequence of values, mediated via coordinate transformations between the nested index space and the flat one. In this section, we explore such flat representations.

Since each dimension of a hypercuboid is Naperian, with a fixed collection of positions, the total size of a hypercuboid is statically determined; so one can rather straightforwardly flatten the whole structure to an immutable linear array. To get the full benefits of the flat representation, that really should be an array of *unboxed* values [30]; for simplicity, we elide the unboxing here, but it should not be difficult to provide that too.

In order to flatten a hypercuboid into a linear array, we need the total size and a list of the elements. The former is provided as the *hsize* method of the type class *Shapely*; for the latter, we define

$$\begin{aligned}&\mathit{elements} :: \mathit{Shapely}\ \mathit{fs} \Rightarrow \mathit{Hyper}\ \mathit{fs}\ a \to [a]\\ &\mathit{elements}\ (\mathit{Scalar}\ a) = [a]\\ &\mathit{elements}\ (\mathit{Prism}\ x) = \mathit{concat}\ (\mathit{map}\ \mathit{toList}\ (\mathit{elements}\ x))\end{aligned}$$

As a representation of flattened hypercuboids, we introduce an indexed version of arrays, preserving the shape *fs* as a type index:

$$\begin{aligned}&\textbf{data}\ \mathit{Flat}\ \mathit{fs}\ a\ \textbf{where}\\ &\quad \mathit{Flat} :: \mathit{Shapely}\ \mathit{fs} \Rightarrow \mathit{Array}\ \mathit{Int}\ a \to \mathit{Flat}\ \mathit{fs}\ a\end{aligned}$$

to which we can transform a hypercuboid:

$$\begin{aligned}&\mathit{flatten} :: \mathit{Shapely}\ \mathit{fs} \Rightarrow \mathit{Hyper}\ \mathit{fs}\ a \to \mathit{Flat}\ \mathit{fs}\ a\\ &\mathit{flatten}\ x = \mathit{Flat}\ (\mathit{listArray}\ (0, \mathit{hsize}\ x - 1)\ (\mathit{elements}\ x))\end{aligned}$$

Here, *listArray* is a standard Haskell function that constructs an array from a pair of bounds and an ordered list of elements. This representation is essentially the same as is used for high-performance array libraries such as Repa [24] for multicore architectures and Accelerate [5] for GPUs; so it should be straightforward to use the abstractions defined here as a front end to guarantee safety, with such a library as a back end for high performance.

The flat contiguous *Array* is one possible representation of the sequence of elements in a hypercuboid, but it is not the only possibility. In particular, we

can accommodate *sparse* array representations too, recording the shape as a type index, and explicitly storing only the non-null elements together with their positions. When the elements are numeric, we could make the convention that the absent ones are zero; more generally, we could provide a single copy of the 'default' element:

```
data Sparse fs a where
  Sparse :: Shapely fs ⇒ a → Array Int (Int, a) → Sparse fs a
```

so that *Sparse e xs* denotes a sparse array with default element e and list xs of proper elements paired with their positions. This can be expanded back to a traditional flat array as follows:

```
unsparse :: ∀fs a . Shapely fs ⇒ Sparse fs a → Flat fs a
unsparse x@(Sparse e xs) = Flat (accumArray second e (0, l − 1) (elems xs))
  where l = hsize (hreplicate () :: Hyper fs ())
        second b a = a
```

Here, *elems* yields the list of elements of an *Array*, which for us will be a list of pairs; and *accumArray f e* (i, j) *xs* constructs a B-array with bounds (i, j) from a list xs of A-elements paired with positions, accumulating the subsequence of elements labelled with the same position using the initial value $e :: B$ and the binary operator $f :: B \to A \to B$. For us, the types A, B coincide, and *second* keeps the second of its two arguments. For simplicity, we compute the size l from a regular *Hyper* of the same shape; more sophisticated approaches are of course possible.

One could similarly provide a run-length-encoded representation, for arrays expected to have long constant sections of different values, and space-efficient representations of diagonal and triangular matrices.

Note that neither the *Flat* nor the *Sparse* representation as shown enforce the bounds constraints. The underlying array in both cases is merely assumed to have the appropriate length for the shape index *fs*. Moreover, for the sparse representation, the positions are additionally assumed to be within range; a more sophisticated representation using bounded naturals *Fin* could be used to enforce the constraints, should that be deemed important. One might also want to maintain the invariant that the elements in the sparse representation are listed in order of position, so that two arrays can easily be zipped via merging without first expanding out to a dense representation; it is straightforward to impose that ordering invariant on the position using dependent typing [27].

In order to provide efficient element access and manipulation, one could combine the array representation with an explicit *index transformation* [16]. Replication and transposition can then be represented by modifying the index transformation, without touching the array elements. We leave the pursuit of this possibility for future work.

9 Conclusions

We have shown how to express the rank and size constraints on multidimensional APL array operations statically, by embedding into a modern strongly typed functional programming language. This means that we benefit for free from all the infrastructure of the host language, such as the type checking, compilation, code optimizations, libraries, and development tools—all of which would have to be built from scratch for a standalone type system such as that of Remora [34].

The embedding makes use of lightweight dependently typed programming features, as exhibited in Haskell. What is quite remarkable is that there is no need for any sophisticated solver for size constraints; the existing traditional unification algorithm suffices (with admittedly many extensions since the days of Hindley and Milner, for example to encompass generalized algebraic datatypes, polymorphic recursion, type families, and so on). This is perhaps not surprising when all one does with sizes is to compare them, but it still applies for certain amounts of size arithmetic. For example, there is no difficulty at all in defining addition and multiplication of type-level numbers,

$$\textbf{type family } \mathit{Add}\ \ (m :: \mathit{Nat})\ (n :: \mathit{Nat}) :: \mathit{Nat}\ \textbf{where}\ ...$$
$$\textbf{type family } \mathit{Mult}\ (m :: \mathit{Nat})\ (n :: \mathit{Nat}) :: \mathit{Nat}\ \textbf{where}\ ...$$

and then writing functions to append and split vectors, and to concatenate and group vectors of vectors:

$$\begin{aligned}
&\mathit{vappend} :: (\mathit{Vector}\ m\ a, \mathit{Vector}\ n\ a) \to \mathit{Vector}\ (\mathit{Add}\ m\ n)\ a\\
&\mathit{vsplit} \quad :: \mathit{Count}\ m \Rightarrow\\
&\qquad\quad \mathit{Vector}\ (\mathit{Add}\ m\ n)\ a \to (\mathit{Vector}\ m\ a, \mathit{Vector}\ n\ a)\\
&\mathit{vconcat}\ :: \mathit{Vector}\ m\ (\mathit{Vector}\ n\ a) \to \mathit{Vector}\ (\mathit{Mult}\ m\ n)\ a\\
&\mathit{vgroup} \quad :: (\mathit{Count}\ m, \mathit{Count}\ n) \Rightarrow\\
&\qquad\quad \mathit{Vector}\ (\mathit{Mult}\ m\ n)\ a \to \mathit{Vector}\ m\ (\mathit{Vector}\ n\ a)
\end{aligned}$$

We have shown how the approach supports various important optimizations, such as avoiding unnecessary replication of data, and flat storage of multidimensional data. In future work, we plan to integrate this approach with existing libraries for high-performance GPU execution, notably Repa [24] and Accelerate [5].

9.1 Related work

We are, of course, not the first to use a type system to guarantee size constraints on dimensions of array operations. The length-indexed vector example is the 'hello, world' of lightweight approaches to dependently typed programming, dating back at least to Xi's Dependent ML [38]. The particular case in which the shape is data-independent, as we address here, can also be traced back to Jay's work [22] on *shapely types*, in which data structures may be factored into 'shape' and 'contents', the latter a simple sequence of elements; then *shapely operations* are those for which the shape of the output depends only on the shape of the input, and not on its contents.

Jay already considered the example of two-dimensional matrices; many others have also used size information at the type level to statically constrain multi-dimensional array operations. Eaton [7] presented a demonstration of *statically typed linear algebra*, by which he meant "any tool which makes [static guarantees about matching dimensions] possible". Scholz's Single-Assignment C [32] represents the extents of multi-dimensional arrays in their types, and Trojahner and Grelck [36] discuss *shape-generic functional array programming* in SaC/Qube. Abe and Sumii [1] present an array interface that enforces shape consistency through what they call *generative phantom types*, and conclude that "practical size checking for a linear algebra library can be constructed on the simple idea of verifying mostly the equality of sizes without significantly restructuring application programs".

Elsman and Dybdal's subset of APL [10] and Veldhuizen's Blitz++ [37] have array types indexed by rank but not size. Chakravarty *et al.*'s Haskell libraries Repa [24] and Accelerate [5] similarly express the rank of a multi-dimensional array in its type, but represent its shape only as part of the value, so that can only be checked at run-time (note that they use both "rank" and "shape" in their papers to refer to what we call rank). Thiemann and Chakravarty [35] explore the trade-offs in developing a front-end to Accelerate in the true dependently typed language Agda in order (among other things) to statically capture shape information; we have shown that it is not necessary to leave the more familiar world of Haskell to achieve that particular end.

None of these works cover full rank polymorphism as in APL and Remora and as in our work: although operations such as map may be applied at arbitrary shape, binary operations such as zip require both arguments to have the same shape—there is no lifting and alignment.

The representation of an array in terms of its *lookup* function, as in our *Naperian* class and our basis for transposition, also has quite a long history. The representation is known as *pull arrays* in the Chalmers work on the digital signal processing language Feldspar [2] and the GPU language Obsidian [6], and *delayed* arrays in Repa [24]. But it is also the essence of *functional reactive animation*, for example in Elliott's Pan [8] and his and Hudak's Fran [9], and in Hudak and Jones's earlier experiment on *geometric region servers* [20].

9.2 Acknowledgements

This paper has benefitted from helpful suggestions from Tim Zakian, Matthew Pickering, Sam Lindley, Andres Löh, Wouter Swierstra, Conor McBride, Simon Peyton Jones, the participants at IFIP WG2.1 Meeting #74 and WG2.11 Meeting #16, and the anonymous reviewers, to all of whom I am very grateful. The work was partially supported by EPSRC grant EP/K020919/1 on *A Theory of Least Change for Bidirectional Transformations*.

O, my offence is rank, it smells to heaven;
It hath the primal eldest curse upon 't.
Shakespeare, "Hamlet", Act III Scene 3

References

1. Akinori Abe and Eijiro Sumii. A simple and practical linear algebra library interface with static size checking. In Oleg Kiselyov and Jacques Garrigue, editors, *ML Family Workshop*, volume 198 of *EPTCS*, pages 1–21, 2014.
2. Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of Feldspar. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2011.
3. Richard Bird, Jeremy Gibbons, Stefan Mehner, Janis Voigtländer, and Tom Schrijvers. Understanding idiomatic traversals backwards and forwards. In *Haskell Symposium*. ACM, 2013.
4. Richard S. Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 1998.
5. Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Declarative Aspects of Multicore Programming*, pages 3–14. ACM, 2011.
6. Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Declarative Aspects of Multicore Programming*, pages 21–30. ACM, 2012.
7. Frederik Eaton. Statically typed linear algebra in Haskell (demo). In *Haskell Workshop*, pages 120–121. ACM, 2006.
8. Conal Elliott. Functional images. In Gibbons and de Moor [15], pages 131–150.
9. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*. ACM, 1997.
10. Martin Elsman and Martin Dybdal. Compiling a subset of APL into a typed intermediate language. In *Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 101–106. ACM, 2014.
11. Matthias Felleisen, Robby B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
12. Raphael A. Finkel and Jon L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
13. GHC Team. Glasgow Haskell Compiler. `https://www.haskell.org/ghc/`.
14. Jeremy Gibbons. APLicative programming with Naperian functors: Haskell code. `http://www.cs.ox.ac.uk/jeremy.gibbons/publications/aplicative.hs`, January 2017.
15. Jeremy Gibbons and Oege de Moor, editors. *The Fun of Programming*. Cornerstones in Computing. Palgrave, 2003.
16. Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *Principles of Programming Languages*, pages 1–8. ACM, 1978.
17. Peter Hancock. What is a Naperian container? `http://sneezy.cs.nott.ac.uk/containers/blog/?p=14`, June 2005.
18. Ralf Hinze. Manufacturing datatypes. *Journal of Functional Programming*, 11(5):493–524, 2001.
19. Ralf Hinze. Fun with phantom types. In Gibbons and de Moor [15], pages 245–262.
20. Paul Hudak and Mark P. Jones. Haskell vs Ada vs C++ vs Awk vs ...: An experiment in software prototyping productivity. Department of Computer Science, Yale, July 1994.

21. Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.
22. C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In Donald Sannella, editor, *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer-Verlag, 1994.
23. Jsoftware, Inc. Jsoftware: High performance development platform. `http://www.jsoftware.com`, 2016.
24. Gabrielle Keller, Manuel Chakravarty, Roman Leschchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic parallel arrays in Haskell. In *International Conference on Functional Programming*, pages 261–272. ACM, 2010.
25. Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed Haskell programming. In *Haskell Symposium*, pages 81–92. ACM, 2013.
26. Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.
27. James McKinna. Why dependent types matter. In *Principles of Programming Languages*, page 1. ACM, 2006. Full paper available at `http://www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf`.
28. Alan Mycroft. Polymorphic type schemes and recursive definitions. In Manfred Paul and Bernard Robinet, editors, *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.
29. Chris Okasaki. *Purely Functional Data Structures*. CUP, 1998.
30. Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, pages 636–666. ACM, 1991.
31. Adam Sandberg Eriksson and Patrik Jansson. An Agda formalisation of the transitive closure of block matrices (extended abstract). In *Type-Driven Development*, pages 60–61. ACM, 2016.
32. Sven-Bodo Scholz. Functional array programming in SaC. In Zoltan Horváth, editor, *CEFP 2005*, volume 4164 of *Lecture Notes in Computer Science*, pages 62–99. Springer, 2006.
33. Alejandro Serrano, Jurriaan Hage, and Patrick Bahr. Type families with class, type classes with family. In *Haskell Symposium*, pages 129–140. ACM, 2015.
34. Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In Zhong Shao, editor, *European Symposium on Programming*, volume 8410 of *Lecture Notes in Computer Science*, pages 27–46. Springer, 2014.
35. Peter Thiemann and Manuel M. T. Chakravarty. Agda meets Accelerate. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2012.
36. Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009.
37. Todd L. Veldhuizen. Arrays in Blitz++. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *International Symposium on Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998.
38. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Programming Language Design and Implementation*. ACM, 1998.
39. Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012.