Multi-dimensional Arrays with Levels

Artjoms Šinkarovs
School of Mathematical and Computer Sciences
Heriot-Watt University
Scotland
tema@pm.me

We explore a data structure that generalises rectangular multi-dimensional arrays. The shape of an n-dimensional array is typically given by a tuple of n natural numbers. Each element in that tuple defines the length of the corresponding axis. If we treat this tuple as an array, the shape of that array is described by the single natural number n. A natural number itself can be also treated as an array with the shape described by the natural number 1 (or the element of any singleton set). This observation gives rise to the hierarchy of array types where the shape of an array of level $l+1$ is a level-l array of natural numbers. Such a hierarchy occurs naturally when treating arrays as containers, which makes it possible to define both rank- and level-polymorphic operations. The former can be found in most array languages, whereas the latter gives rise to partial selections on a large set of hyperplanes, which is often useful in practice. In this paper we present an Agda formalisation of arrays with levels. We show that the proposed formalism supports standard rank-polymorphic array operations, while type system gives static guarantees that indexing is within bounds. We generalise the notion of ranked operator so that it becomes applicable on arrays of arbitrary levels and we show why this may be useful in practice.

1 Introduction

A large number of high-performance numerical problems use multi-dimensional arrays (often referred as tensors) as a key data structure. On the one hand, the multi-dimensional array is a natural abstraction of a space with a regular structure; on the other hand, computations on arrays can be efficiently implemented on conventional computing architectures.

In functional programming, arrays typically do not get a lot of attention, as most of computations on arrays can be expressed as computations on nested lists or vectors, both of which are simpler data structures. While the latter is true, nested vectors miss a very essential feature of many array languages — rank polymorphism. This is the ability to define operations on arrays of arbitrarily many dimensions.

The tradition of rank-polymorphic programming starts from APL [9] and is picked-up by a number of descendants such as J [14], K [16], FISH [10] and others. Rank-polymorphic array programming has found its way into functional langauges as well. For example, SaC [11], Remora [13], Qube [15] are all array-based languages supporting rank-polymorphism. However, enforcing static safety guarantees such as lack of out-of-bound indexing turns out to be a very challenging problem. None of the functional languages above are capable of enforcing such guarantees for the full range of APL operators. The reason for this is that a number of these operators introduce a dependency between the value of the input and the shape of the output. For example, the *take* operator selects a subarray from the given array, and the shape of the subarray comes as an argument. Therefore, a rank-polymorphic language of APL's expressive power which guarantees correct indexing has to support dependent types. Most of practical languages find such a constraint too harsh, due mainly to the fact that one has to give-up global type inference.

Sam Lindley and Max S. New (Eds.): Eighth
Workshop on Mathematically Structured
Functional Programming (MSFP 2020)
EPTCS 317, 2020, pp. 57–71, doi:10.4204/EPTCS.317.4

Consequently, these languages make compromises, either with the range of supported primitives, or with type safety.

When designing a type system for a rank-polymorphic array language the notion of containers [3] comes in very handy. An arrays can be thought of as a tabulated index-value functions, where the set of valid indices into the array is defined by the array shape — exactly the abstraction that containers are designed to handle.

While formalising rectangular multi-dimensional arrays using containers (see Section 2.1), we discovered a new container operation that gives rise to the desired structure:

$$(A \lhd B) \diamond (C \lhd D) = [\![A \lhd B]\!]_{\lhd} C \lhd \lambda(a,s) \to \prod_{B\ a} D \circ s \tag{1}$$

Using this operation, we can define a multi-dimensional array with elements of type X as:

$$Array\ X = [\![(\mathbb{N} \lhd Fin) \diamond (\mathbb{N} \lhd Fin)]\!]_{\lhd}\ X \tag{2}$$

We explain the derivation of the operation, and the array structure in Section 2.

As $\diamond$ is a general container operation, we notice that $- \diamond (\mathbb{N} \lhd Fin)$ can be iterated. By doing so, we get a hierarchy of array types that to our knowledge have not been studied before. Intuitively they can be described as follows. The container $\mathbb{N} \lhd Fin$ describes a finite vector, the shape of which is given by a natural number n. The indices into this vector are natural numbers that are less than n. We call these objects level-1 arrays. If we apply $- \diamond (\mathbb{N} \lhd Fin)$ to a level-1 array we get level-2 arrays. The shape of such a thing is a vector of natural numbers, and the indices are vectors of natural numbers of the same size as the shape, where each element is less than the corresponding element in the shape vector. At the next application, level-3 arrays have shapes that are described by level-2 arrays of natural numbers, and level-3 indices are level-2 arrays of natural numbers, where each element is less than the corresponding element in the shape. And so on.

Beyond simple curiosity, it turns out that these higher-level arrays can be useful in practice. To understand why this is the case, consider the following intuition. The main advantage of multi-dimensional arrays is the availability of proximity metrics for a given element. Within a vector, we can refer only to the left and right neighbours of a cell, whereas within an n-dimensional array we can refer to $2n$ neighbours. Within a multi-level arrays, it is possible to do exactly the same at the level of shapes. The shapes of level-2 arrays are always vectors, therefore one can only talk about left and right shape neighbours. With level-n arrays we can talk about $2n$ shape neighbours, applying all the arsenal of multi-dimensional array operations at the level of shapes. This additional information in the shape makes it possible to define a new class of generic array operations that reshuffle array elements or perform non-trivial partial selections, both of which lie at the very core of array programming.

The paper is a literate Agda script. The contributions are as follows:

- Description of novel data structures that generalise multi-dimensional arrays;
- Formal definition of the data structure and the standard array operations in Agda (available at [12]);
- Generalisation of rank-polymorphic array operations to level-polymorphism, and demonstration of the benefits in practice.

2 Arrays as Containers

In this section we briefly introduce container types and explain how we derived the previously mentioned $\diamond$ operation.

2.1 Containers

Containers can be seen as a conceptual tool to describe "collections of things" such as lists or trees in a uniform way. Mathematically, containers are endofunctors on a category of types, that are coproducts of type-indexed families of representable functors.

We define containers by a type of shapes *Sh* and an *Sh*-indexed type family *Po*. The interpretation (sometimes called extension) of a container type is a dependent pair type where the first element is the shape of type *Sh*, and the second element is a function from positions of that shape to the element type. Following Conor McBride's syntax, we specify containers in Agda as follows[1]:

```
record Con : Set₁ where
  constructor _◁_
  field
    Sh : Set
    Po : Sh → Set
⟦_⟧◁ : Set → Set
⟦_⟧◁ X = Σ Sh λ s → Po s → X
```

To develop an intuition, consider lists of Xs, expressed as a container.

```
List X = ⟦ ℕ ◁ Fin ⟧◁ X  -- ≡ Σ ℕ λ n → Fin n → X
```

For any given length n, the data of the list is modeled by a function of type `Fin n → X`. While the type does not carry the length as an argument, each item of the list container type carries the length in the first element of the dependent pair.

Consider binary trees defined as containers.

```
data Tr : Set where
  Empty : Tr
  Node : Tr → Tr → Tr

data Tx : Tr → Set where
  Done : ∀ {l r} → Tx (Node l r)
  R_ : ∀ {l r} → Tx r → Tx (Node l r)
  L_ : ∀ {l r} → Tx l → Tx (Node l r)

Tree X = ⟦ Tr ◁ Tx ⟧◁ X
```

The shape of the tree is given by a tree that does not store any data in its nodes. Positions into this tree are all the valid paths leading from the root to some node in the shape tree.

For further details on numerous container properties refer to [3, 1, 4, 2].

2.2 Rectangular Arrays

Now let us explore how containers can be used to define rectangular multi-dimensional arrays. The shape of a d-dimensional rectangular array can be represented as a d-element tuple of natural numbers. For a

[1] For presentational purposes we avoid level polymorphism and force shapes to be elements of `Set`. A level-polymorphic version of containers can be found in Agda's standard library.

tuple $(s_1,\ldots,s_d)$, every value s_i represents the number of elements over the axis i in the array with that shape. All the array elements are of type X:

$$Array\ X = \prod_{d\,:\,\mathbb{N}} \prod_{s\,:\,Fin\ d \to \mathbb{N}} \left(\left(\prod_{i\,:\,Fin\ d} Fin\ (s\ i) \right) \to X \right)$$

where d is the number of dimensions, s is a d-element tuple of $\mathbb{N}$ that we model as a function, and the content of the array is a function from indices to values of type X. Indices are d-element tuples of natural numbers where every i-th element is bound by the corresponding position in the shape vector $s\ i$.

Properties The type of indices ensures that out-of-bound access is not possible. Arrays with empty shapes inhabit the *Array* type as there exists a function from $Fin\ 0$ (empty set) to $\mathbb{N}$. Arrays of this kind are often called scalars.

$$s : Fin\ 0 \to \mathbb{N} \qquad \prod_{i\,:\,Fin\ 0} Fin\ (s\ i) \cong \top \qquad X \cong Array\ X\ 0\ s\ (f : \top \to X)$$

Most array processing languages treat scalars as degenerate (0-dimensional) arrays.

By similar reasoning, the *Array* type allows an infinite number of empty arrays (arrays with no elements). An empty array can be characterised by a shape function that evaluates to zero at one of its inputs. (There are no indices permitted in that dimension.)

$$\exists (i : Fin\ d)\ \ s\ i = 0 \implies \prod_{i\,:\,Fin\ d} Fin\ (s\ i) \cong \bot$$

Empty arrays do not contain any elements, but they do exist, due to existence of a function of type $\bot \to X$, whatever the type X. Empty arrays are often found useful in practice, for example as neutral elements for array concatenations.

2.3 The $\diamond$ Operation

Let us find a container formulation for the *Array* type. To do so, we uncurry the first two arguments d and s as follows:

$$Array\ X = \left[\!\!\left[\left(\sum_{d\,:\,\mathbb{N}} Fin\ d \to \mathbb{N} \right) \lhd \lambda (d,s) \to \prod_{i\,:\,Fin\ d} Fin\ (s\ i) \right]\!\!\right]_{\lhd} X$$

We can notice that the first sigma can be represented as a container as well:

$$\sum_{d\,:\,\mathbb{N}} Fin\ d \to \mathbb{N} = [\![\mathbb{N} \lhd Fin]\!]_{\lhd}\, \mathbb{N}$$

Using this observation let us generalise *Array* as a result of the following container operation:

$$(A \lhd B) \diamond (C \lhd D) = [\![A \lhd B]\!]_{\lhd}\, C \lhd \lambda (a,s) \to \prod_{B\ a} D \circ s$$

Using $\diamond$ we describe homogeneous rectangular arrays as:

$$Array\ X = [\![(\mathbb{N} \lhd Fin) \diamond (\mathbb{N} \lhd Fin)]\!]_{\lhd}\ X$$

Discussion One may think about the $\diamond$ operation as of a generalisation of the container operation that is often referred as Hancock's tensor. The tensor operation on containers is defined as:

$$(A \lhd B) \otimes (C \lhd D) = (A \times C) \lhd \lambda\,(a,c) \to B\ a \times D\ c$$

If we want to compute an n-fold tensor product of the container $C \lhd D$:

$$(C \lhd D) \otimes (C \lhd D) \otimes (C \lhd D) \otimes \cdots$$

we need to specify the boundaries of the product. Instead of giving a number, we can use another container, and use its index-space to encode the count. One might write:

$$(A \lhd B) \diamond (C \lhd D) = \bigotimes [\![A \lhd B]\!]_{\lhd}\,(C \lhd D)$$

We "set the bounds" of the iterated tensor product using a "count container" $A \lhd B$. This gives us a family of containers $(a : A, f : B\ a \to (C \lhd D))$ and we compute tensor product of all the elements produced by f.

Further, we notice the following analogy. Similarly to the way $\otimes$ replaces[2] $+$ with $\times$ in the container coproduct:

$$(A \lhd B) + (C \lhd D) = (A + C) \lhd (B + D)$$
$$(A \lhd B) \otimes (C \lhd D) = (A \times C) \lhd (B \times D)$$

in the same way $\diamond$ replaces $\sum$ with $\prod$ in the container composition:

$$(A \lhd B) \circ (C \lhd D) = [\![A \lhd B]\!]_{\lhd} C \lhd \lambda\,(a,\gamma) \to \textstyle\sum_{B\ a} D \circ \gamma$$
$$(A \lhd B) \diamond (C \lhd D) = [\![A \lhd B]\!]_{\lhd} C \lhd \lambda\,(a,\gamma) \to \textstyle\prod_{B\ a} D \circ \gamma$$

Iteration Let us now explore iterated applications of $- \diamond (\mathbb{N} \lhd Fin)$ and $(\mathbb{N} \lhd Fin) \diamond -$ treating $\diamond$ first as a as left-associative and then as a right-associative operation.

$$\begin{aligned}((\mathbb{N} \lhd Fin) \diamond (\mathbb{N} \lhd Fin)) \diamond (\mathbb{N} \lhd Fin) &= \left([\![\mathbb{N} \lhd Fin]\!]_{\lhd}\, \mathbb{N} \lhd \lambda\,(d,s) \to \prod_{Fin\ d} (Fin \circ s) \right) \diamond (\mathbb{N} \lhd Fin) \\ &= \left[\!\!\left[[\![\mathbb{N} \lhd Fin]\!]_{\lhd}\, \mathbb{N} \lhd \lambda\,(d,s) \to \prod_{Fin\ d} (Fin \circ s) \right]\!\!\right]_{\lhd} \mathbb{N} \\ &\quad \lhd \lambda\,((d,s),v) \to \textstyle\prod_{\prod_{Fin\ d}(Fin \circ s)} (Fin \circ v)\end{aligned}$$

In this left-associative case we see that we obtain a generalisation of multi-dimensional arrays which we call level-3 arrays. The shape of a level-2 array is given by a pair (d,s), where d is the dimensionality and s is the shape. The indices into level-2 arrays are vectors of Fin-s of the same length as s. The shape of a level-3 array is a level-2 array v of $\mathbb{N}$. The indices into such an array are level-2 arrays of Fin-s with the same shape as v. We can get even higher levels by further application of $- \diamond (\mathbb{N} \lhd Fin)$.

[2] We have overloaded + and $\times$ for B and D.

When $\diamond$ is right-associative, *i.e.* we successively apply $(\mathbb{N} \lhd Fin) \diamond -$ (on the left) we get:

$$\begin{aligned}(\mathbb{N} \lhd Fin) \diamond ((\mathbb{N} \lhd Fin) \diamond (\mathbb{N} \lhd Fin)) &= (\mathbb{N} \lhd Fin) \diamond \left([\![\mathbb{N} \lhd Fin]\!]_{\lhd} \mathbb{N} \lhd \lambda (d, s) \to \prod_{Fin\ d} (Fin \circ s) \right) \\ &= [\![\mathbb{N} \lhd Fin]\!]_{\lhd} \left([\![\mathbb{N} \lhd Fin]\!]_{\lhd} \mathbb{N} \right) \\ &\quad \lhd \lambda (m, f) \to \prod_{Fin\ m} \left(\lambda (n, s) \to \prod_{Fin\ n} Fin \circ s \right) \circ f\end{aligned}$$

Note that this is well-formed because:

$$f : Fin\ m \to \sum_{n:\mathbb{N}} (Fin\ n \to \mathbb{N})$$

The shape of such an array is a vector of vectors of $\mathbb{N}$. The shape is a 2-dimensional array, but it does not have a rectangular structure, as its rows could be of different lengths. The array itself still has a rectangular structure and is indexed by the $Fin\ x$ "tuples" where x iterates over the elements of the shape. For example, we may create a shape $\begin{pmatrix} 3 & \\ 4 & 5 \end{pmatrix}$ that is encoded with the type $\sum_m (Fin\ m \to \sum_n (Fin\ n \to \mathbb{N}))$ where $m = 2$ and the corresponding two sigmas are $(1, \lambda\ \_ \to 3)$ and $(2, \lambda\ \{0 \to 4; 1 \to 5\})$. The array itself would be isomorphic to the 3-d array of shape $3 \times 4 \times 5$. Further applications of $(\mathbb{N} \lhd Fin) \diamond -$ (on the left) will turn shapes into 3-d irregular arrays, as it simply generates a vector of the shapes from the previous level.

This difference is somewhat natural if we recall the explanation of the $\diamond$ operation via "count containers" and tensor product. Left application of $\diamond$ acts on the original "count container" enriching its structure. The right application turns the structure obtained at the level l into the count container for the level $l + 1$.

3 Array Levels

With the $\diamond$ operation in hand, we can define a hierarchy of arrays with levels in the following way.

```
_◇_ : Con → Con → Con
(S ◁ P) ◇ (S₁ ◁ P₁) = ⟦ S ◁ P ⟧◁ S₁ ◁ λ { (s , γ) → (s₁ : P s) → P₁ (γ s₁) }

A : ℕ → Con
A zero = ⊤ ◁ λ _ → ⊤
A (suc x) = (A x) ◇ (ℕ ◁ Fin)
```

$[\![A\ n]\!]_{\lhd}$ is a level-n array. Our iteration begins with level-0 arrays, where all the shapes are singletons. Level-0 arrays are often referred to as scalars in array calculi. Even though $[\![A\ 0]\!]_{\lhd}\ X$ is isomorphic to X, it still makes sense to have both: a level-polymorphic array operation is applicable to scalars and is not applicable to X.

Unfortunately, this data structure is not very convenient for observing shape relations in array operations. Consider a regular `cons` operator on a level-1 array.

```
cons : ∀ {X} → X → ⟦ A 1 ⟧◁ X → ⟦ A 1 ⟧◁ X
cons {X} x ((_ , s) , p) = (tt , suc ∘ s) , ix-val where
```

```
ix-val : ((x : ⊤) → Fin (suc (s x))) → X
ix-val iv with iv tt
... | zero = x
... | (suc j) = p λ _ → j
```

We would like to observe from the type signature that the resulting array is one element longer than the input. We can surely encode this information as follows:

```
cons+inv : ∀ {X} → X → (v : ⟦ A 1 ⟧◁ X)
                     → Σ (⟦ A 1 ⟧◁ X)
                       λ r → proj₂ (proj₁ r) tt ≡ 1 + proj₂ (proj₁ v) tt
cons+inv x s = cons x s , refl
```

However, as this is a frequent case, we find it more natural to break the structure of the container apart, and lift the shape information into the type. We end up with the following array type.

```
data Ar {a} (l : ℕ) (X : Set a) (s : Sh (A l)) : Set a where
  imap : (Po (A l) s → X) → Ar l X s
```

Note that instead of fixing `A` in `Ar`, we can have a generic definition:

```
data TC {C : Con}{a} (X : Set a) (s : Sh C) : Set a where
  imap : (Po C s → X) → TC X s
```

in which case `Ar l X s` would be defined as `TC {C = A l} X s`. In some sense `C` and `TC` (or `A` and `Ar`) are related in a similar way as `List` and `Vec`. We lift a commonly used invariant (shape in case of containers and length in case of lists) into a type-level argument.

With the help of `Ar` we can express `cons` as:

```
cons-ar : ∀ {X s} → X → Ar 1 X (_ , s) → Ar 1 X (_ , suc ∘ s)
```

Also, such a formulation naturally gives rise to the `imap` construct that is a basic building block of the SaC programming language. In SaC arrays are treated as tabulated index-value functions. The `imap` construct can be thought of as an abstract tag that indicates that a chosen function has to be eventually tabulated. However, the exact details on how this function is to be tabulated are not specified. As a result, when producing an executable for the given program, a compiler has a lot of freedom to choose storage formats for arrays, based on the information that is being accumulated during optimisation phases.

The two data structures are isomorphic, which is indicated by the following conversion functions

```
c→ar : ∀ {n X} → (c : ⟦ A n ⟧◁ X) → Ar n X (proj₁ c)
c→ar c = imap (proj₂ c)

ar→c : ∀ {n X s} → Ar n X s → ⟦ A n ⟧◁ X
ar→c {s = s} (imap x) = s , x
```

Finally, if we consider `imap` as an array constructor, there has to be an eliminator. The eliminator for the array is a selection operation, and our model, this is simply a function application.

```
sel : ∀ {a}{X : Set a}{n s} → Ar n X s → Po (A n) s → X
sel (imap x) iv = x iv
```

3.1 Lack of Extensionality

While the above model gives us all the fundamental array primitives, it has a serious flaw when we come to reasoning about array equalities. One of the fundamental assumptions in array calculi is that the same indices select the same values. Indices of level-n arrays, where $n > 0$ are functions, and therefore we define index equality extensionally. It should not matter how exactly the elements within the index are computed, as long as two indices are element-wise equal, they should select the same element. Unfortunately, in Agda this cannot be shown. The code below demonstrates the problem.

```
po-eq : ∀ {l s} → (iv jv : Po (A l) s) → Set
po-eq {zero} iv jv = iv ≡ jv
po-eq {suc l} iv jv = ∀ i → iv i ≡ jv i

sel-eq : ∀ {a}{X : Set a}{l s}
       → (a : Ar l X s)
       → (iv jv : Po (A l) s)
       → po-eq {l = l} iv jv
       → sel a iv ≡ sel a jv
```

Recall that even at the first level the indices have a type $\top \to$ `Fin (s tt)`:

```
sanity : ∀ s → Po (A 1) (_ , s) ≡ (⊤ → Fin (s tt))
sanity s = refl
```

Without the `sel-eq` property we cannot show very fundamental array facts such as element preservation under flattening/unflattening, reshaping, transposition, *etc*. This problem can be worked around in a number of ways including defining custom equality relation and working in setoids, defining custom selection operation or using cubical Agda. In this paper we introduces a non-functional representation for indices so that the `sel-eq` becomes provable. The details are described in the next sections.

4 Alternative Encoding

The main idea here is to define an alternative representation for `A` from the previous section that makes it possible to prove that indices with the same components select the same array elements. We achieve this by using a non-function based representation for shapes and indices, while still representing contents of arrays as imap-tagged index-value functions.

We start with an alternative definition for Fin in a refinement type [7, 8] style. While this is not strictly necessary, it helps to simplify a number of proofs later. The main difference from the regular Fin is that we keep the actual value of an index as an element of type $\mathbb{N}$, and make irrelevant (in Agda sense) the proof that this value is less than the chosen upper bound. Note the dot in the `v<u` field name.

```
record BFin (u : ℕ) : Set where
  constructor _bounded_
  field
    v : ℕ
    .v<u : v < u
```

As we are defining an array and its representation at the same time, and we make these definitions interdependent, we sometimes have to provide types first and only later provide the actual definition. We start with leveled types for shapes, indices and array representations.

```
ShType : (l : ℕ) → Set
IxType : (l : ℕ) → ShType l → Set
ReprAr : (l : ℕ) → (X : Set) → Set
```

Note that for simplicity we do not impose any requirements on the validity of array representation. We can define these properties later extrinsically.

As will be seen later, IxType may have the same representation for indices into arrays of different shapes. To avoid this we define the Ix type that wraps IxType and carries array shape as a type parameter.

```
record Ix (l : ℕ) (s : ShType l) : Set where
  constructor ix
  field
    flat-ix : IxType l s
```

As a result, we will not be able to index an array of shape s with an index bound by shape s_1 without explicit cast. In the same way as one cannot pass the term of type `Fin 10` to the function `Fin 15 → X`.

As before, arrays are tabulated index-value functions where `imap` is a constructor.

```
data Ar {a} (l : ℕ) (X : Set) (s : ShType l) : Set a where
  imap : (Ix l s → X) → Ar l X s
```

Our array representation needs the information on how many elements does the array contain, which is just a product of the shape elements. Again, we cannot yet provide the body of the function, as we have not yet defined how we represent shapes.

```
prod : ∀ {l} → ShType l → ℕ
```

Finally, we get to the definition of the shape representation, which we chose to be the unit type for level-0 arrays and ReprAr otherwise. While the latter is not strictly necessary, it gives us a nice symmetry between the arrays and their shapes.

```
ShType zero = ⊤
ShType (suc l) = ReprAr l ℕ
```

Our array representation is a dependent pair where the first element is the representation of the shape, and the second element is a linearisation of array elements — a vector that has as many elements as an array of the given shape.

```
ReprAr l X = Σ (ShType l) λ s → Vec X (prod {l = l} s)
```

As before, indices into an array of shape s have the same structure as s, except each valid index is component-wise less than s. As ShType keeps shape elements in a linearised form, the index type mimics the structure of the shape:

```
infixr 5 _::_
data FlatIx : (d : ℕ) → (s : Vec ℕ d) → Set where
```

```
[] : FlatIx 0 []
_::_ : ∀ {d s x} → BFin x → (ix : FlatIx d s) → FlatIx (suc d) (x :: s)
```

That is, a linearised shape is given by a vector of d elements, therefore an index contains BFin elements where the upper bounds refer to the corresponding elements of the linearised shape.

The only valid index type for level-0 shapes is a singleton type, for which we use the unit type, and for higher level arrays we use FlatIx.

```
IxType zero tt = ⊤
IxType (suc l) (s , v) = FlatIx (prod s) v
```

The product is a fold of multiplication with the neutral element 1.

```
flat-prod : ∀ {n} → Vec ℕ n → ℕ
flat-prod = foldr _ _*_ 1

prod {zero} sh = 1
prod {suc l} (s , v) = flat-prod v
```

4.1 Examples

In order to develop a better intuition of the above data structures, let us define a few simple examples. We start with defining some arrays of levels zero, one and two.

```
sca : Ar 0 ℕ tt                      -- A scalar
vec : Ar 1 ℕ (tt , 5 :: [])          -- Vector of 5 elements
mat : Ar 2 ℕ ((tt , 2 :: []) , 2 :: 2 :: [])  -- Matrix of 2×2 elements
```

Let us now define the values.

```
sca = imap λ _ → 42
vec = imap λ {(ix (0 bounded _ :: [])) → 42 ; _ → 0}
mat = imap λ {(ix (1 bounded _ :: 1 bounded _ :: [])) → 42 ; _ → 0}
```

We defined a scalar with a value 42; a vector of 5 elements with the value 42 at the index zero and value 0 elsewhere; a matrix with the value 42 at index $[1,1]$ and zeroes elsewhere.

For a more realistic example consider a level-polymorphic array addition.

```
plus : ∀ {l s} → Ar l ℕ s → Ar l ℕ s → Ar l ℕ s
plus (imap a) (imap b) = imap λ iv → a iv + b iv
```

Generally speaking, any element-level function can be lifted to the level of arrays: Ar with a fixed level and shape is a functor. As another realistic example consider matrix multiplication. We formulate it in a way so that it operates on 2-dimensional arrays of any size, given that the outer dimension of the first matrix is identical to the inner dimension of the second one.

```
matmul : ∀ {m n p} → let Sh x y = ((_ , 2 :: []) , x :: y :: []) in
         Ar 2 ℕ $ Sh m p → Ar 2 ℕ $ Sh p n → Ar 2 ℕ $ Sh m n
```

```
matmul {p = p} (imap a) (imap b) = imap mat-content
  where mat-content : _
        mat-content (ix (i :: j :: [])) = let
          t : Ar 1 ℕ (_ , p :: [])
          t = imap λ {(ix (k :: [])) → a (ix $ i :: k :: []) * b (ix $ k :: j :: [])}
          in sum t
```

5 Practical Applications

Most of the typical array-based problems use arrays up to level two. Therefore, it is reasonable to ask whether there are any applications where availability of higher levels is handy. As a motivating example, we consider an instance of the average pooling problem [6] that is commonly used in machine learning applications. Given a matrix of size $(2m)\times(2n)$ we produce an $m\times n$ matrix by averaging 2×2 subarrays, for example:

$$avgp\begin{pmatrix}1 & 2 & 5 & 6\\ 3 & 4 & 7 & 8\end{pmatrix} = \begin{pmatrix}(1+2+3+4)/4 & (5+6+7+8)/4\end{pmatrix}$$

A 2×4 matrix is turned into a 1×2 one where all the 2×2 subarrays are averaged.
Even though it is straight-forward to implement this function directly:

```
avgp-direct : ∀ {m n} → Ar 2 ℕ ((tt , 2 :: []) , m * 2 :: n * 2 :: [])
                       → Ar 2 ℕ ((tt , 2 :: []) , m :: n :: [])
avgp-direct {m}{n} (imap a) = imap array-content
  where array-content : _
        array-content (ix (i bounded _ :: j bounded _ :: [])) = let
          t : Ar 2 ℕ ((tt , 2 :: []) , 2 :: 2 :: [])
          t = imap λ { (ix (i' bounded _ :: j' bounded _ :: [])) →
                       a (ix $ (i + i') bounded i+i'<m*2 :: (j + j') bounded j+j'<n*2 :: []) }
          in sum t / 4
```

we had to manually perform index manipulations and needed to prove two theorems. What if we try to implement the same algorithm using aggregate array operations in the style of APL to avoid index manipulations?

If we were to transform the array of shape $(2m)\times(2n)$ into an array of shape $m\times n\times 2\times 2$, we could use the concept of a ranked operator [5] to apply average operation on the last two axes. A ranked operator can be thought of as a facility to turn a level-2 array (`Ar 2 X` $m\times n\times 2\times 2$) into a nested array `Ar 2 (Ar 2 X` 2×2`)` $m\times n$. The argument to the ranked operator typically indicates where we "cut" the shape vector.

We can define a reshape operation that allows us to change the shape of the array, given that the new shape has the same number of elements and that the order of elements under some chosen linearisation is preserved. Here is the type for the reshape operation for arrays with levels:

```
reshape : ∀ {X l l₁ s s₁} → Ar l X s → prod s ≡ prod s₁ → Ar l₁ X s₁
```

Implementation of this operation can be found in [12]. We choose a row-major order as our linearisation. The reshape operation first computes an offset into the linearised array and then turns this offset into the index of the new shape.

Note that reshaping an array of shape $(2m) \times (2n)$ into shape $m \times n \times 2 \times 2$ would deliver an incorrect tiling.

$$\mathit{reshape} \begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{pmatrix} = \left(\begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} \quad \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix} \right) \quad \text{and not} \quad \left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \right)$$

This is happening because under row-major order, the elements of the row are kept together. In order to fix this problem we need to reshape our array into shape $m \times 2 \times n \times 2$. However, if we do so, we cannot apply ranked operator anymore — there is no way to "cut" the $m \times 2 \times n \times 2$ shape vector so that 2×2 become "neighbours". It is here the concept of higher-level shapes becomes useful. If we consider the shape $m \times 2 \times n \times 2$ as a linearised 2×2 array $\begin{pmatrix} m & 2 \\ n & 2 \end{pmatrix}$ then 2×2 are neighbours in the second column of the shape. Now we can define a smarter version of the ranked operator, that "cuts" the shape across the column, and produce a level-3 nested array where the shape of the inner array is $\begin{pmatrix} 2 \\ 2 \end{pmatrix}$.

We make this idea precise by defining a ranked operator that "cuts" shapes of any levels into two parts. Later these parts can be used to create an element-preserving array nesting.

Let us first define the type that would capture all the allowed "cuts" of the given shape:

```
RankedT : ∀ {l : ℕ} → ShType l → Set
RankedT {0} _ = ⊤
RankedT {1} (s , v) = BFin (1 + prod s)
RankedT {suc (suc l)} ((s , v) , _) = Σ (BFin (prod s)) λ i → BFin (1 + blookup v i)
```

where `blookup` has the type `Vec X n → BFin n → X` and it selects an element from a vector at a given index.

Level-0 arrays can be cut only in a single way, producing two unit shapes. Then the array of type `Ar 0 X tt` can be straight-forwardly nested into the array of type `Ar 0 (Ar 0 X tt) tt)`.

Level-1 arrays can be nested in two different ways: with a singleton shape on the outside or on the inside. That is, $[1,2,3]$ can be turned into a nested vector as $[[1,2,3]]$ or as $[[1],[2],[3]]$.

For arrays of levels greater than one we first pick an index into the shape of the shape (vector `v`), and then we pick a number that is less or equal than the element of `v` at that index. Consider an example. For a level-2 array of shape `(tt , 3 :: []) , m :: n :: k :: []` we first have to pick an element from the one-element vector, and then we pick a number that is less or equal than 3. Assume, we picked 1, in this case all the elements of `m :: n :: k :: []` with index that is smaller to 1 will form a left shape and the rest of the index will go into the right shape, resulting in shapes `(tt , 1 :: []) , m :: []` and `(tt , 2 :: []) , n :: k :: []`. Note that if we pick 0, we end-up with shapes `(tt , 0 :: []) , []` and `(tt , 3 :: []) , m :: n :: k :: []`. The left shape in this case is a singleton which means that we will get a valid nesting.

We define a function that "cuts" a shape into two given the shape and the argument of `RankedT` type.

```
ranked-cut : ∀ {l : ℕ} → (s : ShType l) → RankedT s → ShType l × ShType l
```

And we use this function to define the nesting operation.

```
nest : ∀ {l X s} → Ar l X s → (ri : RankedT s) →
       let s₁ , s₂ = ranked-cut s ri in
       Ar l (Ar l X s₂) s₁
```

The implementation details can be found in [12]. Unfortunately things get rather non-trivial rather quickly, as we have to implicitly prove that the shapes obtained as a result of `ranked-cut` can be merged together into the same shape.

Finally, consider an index-free formulation of the average pooling using level-3 arrays.

```
s≡s' : ∀ m n → m * 2 * (n * 2 * 1) ≡ m * (n * 2 + (n * 2 + 0))

map : ∀ {X Y l s} → (X → Y) → Ar l X s → Ar l Y s
map f (imap a) = imap λ iv → f $ a iv

avgp : ∀ {m n}
     → Ar 2 ℕ ((tt , 2 :: []) , m * 2 :: n * 2 :: [])
     → Ar 2 ℕ ((tt , 2 :: []) , m :: n :: [])
avgp {m}{n} a = let
  s₁ : ShType 3
  s₁ = ((_ , 2 :: []) , 2 :: 2 :: [])
       , m :: 2 :: n :: 2 :: []
  a₁ = reshape {s₁ = s₁} a (s≡s' m n)
  aₙ = nest a₁ ((1 bounded auto≥) , (1 bounded auto≥))
  r₃ = map ((_/ 4) ∘ sum) aₙ
  in reshape r₃ refl
```

First, we reshape the input array of shape `m*2 :: n*2 :: []` into the level-3 array of shape $s_1 = \begin{pmatrix} m & 2 \\ n & 2 \end{pmatrix}$. Then we cut s_1 using `ranked-cut` with the argument (1 , 1) of type `RankedT`. The first '1' says that from axes `2 :: 2 :: []` (the shape of s_1) we pick the second. The second '1' says that all the s_1 elements which have index where the second component is smaller than one will form the left shape of the `ranked-cut`. In other words we are saying that we are cutting s_1 vertically, taking the first column as the left shape. Then we apply the average function to $\begin{pmatrix} 2 \\ 2 \end{pmatrix}$ subarrays. The `auto`$\geq$ is an instance function that automatically proves trivial inequalities on natural numbers. The `map` is defined right before the `avgp`. Note that this version of the map does not prescribe the order in which array elements are traversed. We say only that all the elements are modified with some function `f` of the appropriate type. In the last statement, the equality between products of $\begin{pmatrix} 2 \\ 2 \end{pmatrix}$ level-3 shape and 2×2 level-2 shape is obvious to Agda.

Discussion The definition of the `nest` operator uses some quite heavy machinery, even though we eliminated direct index manipulations in the average function. One can ask whether this is justified. Surely, for the purposes of a single function it may be not. However, the pattern exemplified above, in which we reshape an array in a such a way that inner dimensions contain the elements of interest and then we map a function over all these elements is extremely common in array programming. In an array library or a DSL embedded into Agda-like framework, the *ability to define* the extended ranked operator is very valuable — we only have to define it once.

Finally, while defining `nest` we had to first prove the fact that index to offset and offset to index functions are inverses of each other. The core of this idea is captured in the theorem `io-oi` in [12].

6 Conclusions

We have presented a novel data structure that generalises multi-dimensional arrays. The key to our construction is a strong symmetry or analogy between the type that describes the shape of the data structure and the data structure itself. Such a symmetry gives rise to the hierarchy of types — in our case, an array of natural numbers can be used as a shape descriptor of the next array type in the hierarchy. We start with unit shapes and corresponding one-element level-0 arrays. After that, level-1 arrays have a shape that is described by a single natural number and the array itself has n elements. At level 2 we the shape is given by an n-element array of natural numbers, and so on.

This hierarchy appeared naturally after we encoded multi-dimensional arrays using containers. We discovered a new container operation, and the iteration of this operation led to the array type hierarchy.

While only the first three levels of the hierarchy have been used in practice so far, we have demonstrated that higher levels are also of practical useful. They naturally fit the tradition of rank-polymorphic aggregated operations in the style of APL, suggesting that array operations can be expressed in an index-free combinator style. The availability of higher-level arrays makes it possible to enrich the functionality of existing combinators, as we have demonstrated at the example of the ranked operator.

In order to make this idea precise, we encoded the proposed array types and operations in Agda and observed a number of standard array properties.

This work opens up a number of interesting research directions. For example: using other containers to form similar type hierarchies. It would be interesting to explore alternative ways to work around the container extensionality problem, so that less encoding machinery needs to be exposed.

Acknowledgements

I am very grateful to Peter Hancock. He was the first one to suggest to treat arrays as containers, and came up with the initial version of the $\diamond$ operation. Two days of intensive discussions with Sven-Bodo Scholz on the meaning of arrays with levels were of a great help as well.

References

[1] Michael Abbott, Thorsten Altenkirch & Neil Ghani (2003): *Categories of Containers*. In Andrew D. Gordon, editor: *Foundations of Software Science and Computation Structures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 23–38, doi:10.1007/3-540-36576-1_2.

[2] Michael Abbott, Thorsten Altenkirch & Neil Ghani (2004): *Representing Nested Inductive Types Using W-Types*. In Josep Díaz, Juhani Karhumäki, Arto Lepistö & Donald Sannella, editors: *Automata, Languages and Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 59–71, doi:10.1007/978-3-540-27836-8_8.

[3] Michael Abbott, Thorsten Altenkirch & Neil Ghani (2005): *Containers: Constructing Strictly Positive Types*. *Theor. Comput. Sci.* 342(1), p. 3–27, doi:10.1016/j.tcs.2005.06.002.

[4] Michael Abbott, Thorsten Altenkirch, Neil Ghani & Conor McBride (2003): *Derivatives of Containers*. In Martin Hofmann, editor: *Typed Lambda Calculi and Applications*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 16–30, doi:10.1007/3-540-44904-3_2.

[5] R. Bernecky (1987): *An Introduction to Function Rank*. *SIGAPL APL Quote Quad* 18(2), p. 39–43, doi:10.1145/377719.55632.

[6] Y-Lan Boureau, Jean Ponce & Yann LeCun (2010): *A Theoretical Analysis of Feature Pooling in Visual Recognition*. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, Omnipress, Madison, WI, USA, p. 111–118.

[7] Tim Freeman & Frank Pfenning (1991): *Refinement Types for ML*. *SIGPLAN Not.* 26(6), p. 268–277, doi:10.1145/113446.113468.

[8] Susumu Hayashi (1994): *Logic of refinement types*. In Henk Barendregt & Tobias Nipkow, editors: *Types for Proofs and Programs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 108–126, doi:10.1007/3-540-58085-9_74.

[9] Kenneth E. Iverson (1962): *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, doi:10.1145/1460833.1460872.

[10] C Barry Jay & Paul A Steckler (1998): *The functional imperative: shape!* In: *European Symposium on Programming*, Springer, pp. 139–153, doi:10.1007/BFb0053568.

[11] Sven-Bodo Scholz (2003): *Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting*. *J. Funct. Program.* 13(6), pp. 1005–1059, doi:10.1017/S0956796802004458.

[12] Artjoms Šinkarovs (2020): *Arrays with Levels in Agda*. https://github.com/ashinkarov/agda-arrays-with-levels. [Accessed: March 2020].

[13] Justin Slepak, Olin Shivers & Panagiotis Manolios (2014): *An Array-Oriented Language with Static Rank Polymorphism*. In Zhong Shao, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 27–46, doi:10.1007/978-3-642-54833-8_3.

[14] Roger Stokes (15 June 2015): *Learning J. An Introduction to the J Programming Language*. http://www.jsoftware.com/help/learning/contents.htm. [Accessed: March 2020].

[15] Kai Trojahner & Clemens Grelck (2009): *Dependently typed array programs don't go wrong*. *The Journal of Logic and Algebraic Programming* 78(7), pp. 643 – 664, doi:10.1016/j.jlap.2009.03.002. The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[16] Arthur Whitney (2001): *K*. http://archive.vector.org.uk/art10010830. [Accessed: March 2020].