Transfinite Arrays Formally

ARTJOMS ŠINKAROVS, Heriot-Watt University, UK

Streams are often thought of as infinite versions of lists. They have a very similar interface, and in some languages these two data structures are indistinguishable. What would be an infinite version for multi-dimensional arrays that preserve array-algebraic properties such as rank polymorphism and row-major reshaping? We answer this question with a novel data structure named transfinite arrays — arrays that are indexed with countable ordinals. With this data structure programmers can express generic algorithms on arrays without distinguishing whether the number of array elements is finite or inifnite, which avoids answering the ever lasting question "to stream or not to stream". We formalise finite rank-polymorphic arrays in Agda, show their main properties and extend them into transfinite domain, ensuring that rank-polymorphism and row-major reshaping is preserved. We explore alternative constructions of infinite arrays, their shortcomings and provide a number of examples.

Additional Key Words and Phrases: keyword1, keyword2, keyword3

1 INTRODUCTION

Numerical problems often model phenomena that occur in infinite domains. For example, dynamical systems bound by laws of physics usually evolve in infinite time and infinite space. As such domains cannot be represented in a computer memory directly, typically, either a model is restricted to finite subdomains, or a finite encoding is chosen for an infinite domain. For example, in dynamical systems, it is common to restrict space to some finite area/volume (so that it fits into memory). The infinite time is encoded as a function that returns the state of the chosen subspace at every given time.

While these restrictions are important implementational details, very often they are hard-coded into problem specifications. As a consequence, the algorithm gets significantly obfuscated by the encoding artifacts, suffering readability and maintainability. More importantly, an attempt to change data representation from finite to infinite or vice versa very often results in major program rewrites.

The main question of this paper is how to liberate numerical specifications from the necessity to distinguish between finite and infinite data and make them *infinity-agnostic*. If we restrict our collections to sequences, then list-stream pair provides a reasonable answer. Lists represent finite sequences, streams — infinite, and it is possible to define a common interface to both data structures. For example, combinators such as map, zip or take behave in the same way on both data structures.

However, the key data structure in numerical applications is the multi-dimensional array. On the one hand it is a natural abstraction for spaces with rectangular structure; on the other hand, computations on arrays can be efficiently implemented on conventional architectures. While arrays sometimes are identified with nested list, such a view misses on the very essential array programming feature *rank polymorphism* — the ability to define operations on arrays of any number of dimensions. This happens because in order to figure out whether the list is nested or not, one has to pattern-match on the type, which in turn breaks parametricity (see in-depth discussion in Section 2). Consequently, a joint interface between arrays and streams would miss on rank polymorphism as well.

Author's address: Artjoms Šinkarovs, School of Mathematical and Computer Sciences, Heriot-Watt University, Heriot-Watt University, Edinburgh, Scotland, EH14 4AS, UK, first1.last1@inst1.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Another essential part of rank-polymorphic arrays is the associated array algebras. After the success of APL [Iverson 1962] in early 60s, several attempts were taken to design an array theory [Jenkins and Glasgow 1989; More 1973; Mullin 1988]. Similarly to set theories, array theories capture the main assumptions on the structure of arrays, basic operations, and the axioms that explain behaviour of these operations. These axioms are often given as universal equalities that can be also thought of as rewrite rules. Therefore, we find it essential to preserve as many of such equalities as possible in the infinite case.

While the previously mentioned array theories differ in details, we observe at least one common axiom about the very basic array operation called *reshape* — an element-preserving change of the array shape. The axiom says that reshape happens by first row-major flattening an array (into a 1-dimensional array) and then "unflattening" it into the desired shape.

Unfortunately, row-major flattening does not work in case of streams or colists. The essence of the row-major flattening lies in the ability to take apart "things" that were concatenated. In the array/list terminology, an essential equality of the row-major reshape is:

$$\mathit{drop}\ (\mathit{shape}\ a)\ (a +\!\!+ b) = b$$

where a and b are 1-dimensional arrays, in which case *shape* is just their lengths. While this equality holds for lists, it fails in case a is an infinite stream/colist. Intuitively, we can concatenate infinite lists with "stuff", but we cannot access this stuff ever again.

In order to salvage row-major flattening, we introduce a novel data structure called *transfinite arrays* — arrays that are indexed with countable ordinals[1]. This appears to be a natural extension of finite arrays, and its definition is inductive. We make sure that ordinal equality and comparison is decidable, which means that infinity-agnostic specification can take local decisions based on the finiteness of inputs.

We use Martin-Löf dependent type theory [Martin-Löf 1985] (and Agda [Norell 2009] as the implementation of it) to define rank-polymorphic transfinite arrays. The resulting framework can be seen as a minimalist array theory that is consistent (none of the mentioned array theories formally showed this) and that offers immediate computational interpretation (all the mentioned array theories are axiomatic). We believe that the proposed theory exposes the basic principles that can be used to build an actual programming language.

The individual contributions of the paper are as follows:

- Define finite multi-dimensional rank-polymorphic arrays so as to provide a static guarantee that data-access is within bounds.
- Encode ordinal numbers in Cantor Normal Form, arithmetic operations, comparisons and a number of accompanying theorems.
- Generalise finite arrays to transfinite ones using ordinal indexing, demonstrate preservation of rank-polymorphism and row-major flattening.

This paper is an Agda script.

2 FINITE RANK-POLYMORPHIC ARRAYS

In this section we explore the consequences of not distinguishing arrays and lists; review three existing array theories and define the encoding for multi-dimensional arrays in Agda.

We define arrays as finite hyper-rectangular tables that can be indexed by tuples of natural numbers. These can be immediately thought of as nested lists[2] where the nesting depth is the number of array dimensions (also referred as *rank*). As noticed by [More 1979], such an encoding

[1]By ordinals in this text we mean an initial segment of countable ordinals up to ϵ_0.
[2]We mean here homogeneous nesting where all the sublists are of the same length.

looses the information about array nesting levels. That is, a list [[1,2,3],[4,5,6]] can be both: an array of shape 2×3 of numbers, and a two-element array of three-element arrays of numbers. While these two interpretations are isomorphic, conversion from arrays to lists is not injective.

Even if we assume that arrays do not nest, the list-based encoding does not admit rank-polymorphic operations. Assume defining a function that transposes any n-dimensional array. Computing rank from the list representation would require pattern-matching on the list element type. Even though this could be done in principle, most of the existing strongly-typed functional languages, *e.g.* Haskell, Agda, or Idris choose not to do so as it breaks parametricity. As a consequence, within a parametric type system arrays and lists shall be distinct.

Rank polymorphism. This concept was pioneered by APL and later picked-up by a plethora of array languages: J [Stokes 2015], K [Whitney 2001], SaC [Scholz 2003], Remora [Slepak et al. 2014], Qube [Trojahner and Grelck 2009] and many more. In essence, rank polymorphism is at the very core of the APL success story: incredible expressiveness of array computations written in an index-free combinator style. As an example, consider the following (valid) APL expression:

```
2 ÷⍨ 1 ⌽ a + ¯1 ⌽ a
```

that computes a two-point convolution of the array `a`. This is done by first rotating vectors along the last axis of `a` one element to the left (`¯1 ⌽ a`), then one element to the right (`1 ⌽ a`), then adding these results element-wise (`+`), and then dividing each element by two (`2 ÷⍨`). The important point is that this expression is applicable to `a` of *any* rank, including zero-dimensional arrays (often called *scalars*). Not only the initial set of APL combinators found very useful in practice, but it also gives rise to the number of universal equalities such as:

```
(-x) ⌽ x ⌽ a ≡ a
```

It says: if we first rotate vectors in the last axis of `a` `x` elements in one direction and then rotate by `x` elements in the opposite direction, we will always get back the same array. These universal equalities are not surprising, as they are based on simple arithmetic facts, yet they give a powerful reasoning technique and they can be used as rewrite rules when programs are automatically transformed.

2.1 Existing Theories

In order to study universal array equalities formally, a number of array theories have been proposed. Most noticeably by Jenkins, More and Mullin. All of these are given in an axiomatic style and do not have direct computational interpretations. Consistency of these theories, to our knowledge, has never been formally shown as well. We make a brief overview of the basic notions.

Jenkinks. The first section of [Jenkins and Glasgow 1989] studies whether it would be possible to use lists as a representation for arrays an uses the following instructional example that can be restated as follows. It is well-known that List and Vec[3] are isomorphic: going from Vec to List requires forgetting the length, and going from List to Vec is always possible by recomputing the length of the list.

[3] As a quick reminder, List of X-es has two constructors: [] to build an empty list and _::_ to append an element at the front of the already constructed list. Vec of X-es is an indexed version of the List, where the index is a natural number that represents the length of the list.

```
data List (X : Set) : Set where            data Vec (X : Set) : ℕ → Set where
  [] : List X                                [] : Vec X 0
  _::_ : X → List X → List X                 _::_ : ∀ {n} → X → Vec X n → Vec X (1 + n)
```

Note that for readability purposes, in the paper we avoid universe polymorphism. In the actual encoding, all the types are universe-polymorphic.

```
l-v : ∀ {X} (x : List X) → Vec X (length x)
v-l : ∀ {X}{n} (v : Vec X n) → List X
```

We can easily show[4] that these conversion functions are inverses of each other. Now, if we have a 2-d table of elements, and we have two functions: rows (that turns a table into the list of rows of the table), and mix that turns a list of rows back into the table; can we show that mix (rows *x*) is *x*?

```
data Table (X : Set) : ℕ → ℕ → Set where
  tab : (m n : ℕ) → Vec X (m * n) → Table X m n

#rows : ∀ {X} → List (List X) → ℕ
#cols : ∀ {X} → List (List X) → ℕ
rows : ∀ {X m n} → Table X m n → List (List X)
mix  : ∀ {X} (x : List (List X)) → Table X (#rows x) (#cols x)
```

First observation is that a straight-forward encoding representing tables with no rows with the empty list, leads to a contradiction. This happens because for the mix/rows isomorphism to work, rows have to preserve the size of the table:

```
#rows-pres : ∀ {X m n}{x : Table X m n} → #rows (rows x) ≡ m
#cols-pres : ∀ {X m n}{x : Table X m n} → #cols (rows x) ≡ n
```

which leads to the following contradiction:

```
contra : ∀ {X} → (thm : ∀ {n} {x : Table X 0 n} → rows x ≡ []) → 5 ≡ 6
```

The proof of the contradiction uses the fact that the empty table of 5-element rows is encoded in the same way as the empty table with 6-element rows. Therefore, if we ever can decode this back to the original table, #cols must (magically) produce the length of the original table, which is impossible.

However, does this mean that while List is isomorphic to Vec, List∘List and Vec∘Vec are not? It is clear that List (List *X*) encodes more objects than Vec (Vec X n) m, as inner lists may be of different lengths. Let us restrict to equilateral inner lists, *i.e.* each list *l* comes with the proof of type ll-coh *l*:

```
ll-coh : (x : List (List X)) → Set
ll-coh [] = ⊤
ll-coh (x :: xs) = All (λ t → length t ≡ length x) xs
```

Do we have an isomorphism then? It turns out that we do, and it is witnessed by (surprisingly long!) proof that can be found in supplementary materials when an instance of the Table≅ListList type is defined.

To work around the loss of information we use the following diagonalisation argument. As conversion functions of the empty table is parametric in *X*, the encoding must preserve the number of the elements, *i.e.* tables of shape $m \times 0$ or $0 \times n$ must be encoded with a sequence of empty lists. This means that we have to find an invertible encoding from pairs $(m, 0)$ and $(0, n)$ to natural numbers, and generate that many empty lists when converting an empty table. In the proof we use the following scheme:

$$(0,0) \mapsto 0 \qquad (m,0) \mapsto 2m-1 \qquad (0,n) \mapsto 2n$$

This encoding is obviously (not to Agda though) invertible, therefore we can recover previously lost information. Non-empty tables of shape $m \times n$ can be encoded in the usual row-major way: by "cutting" the data vector of the table into m lists of length n.

[4] This proof can be found in the supplementary materials when defining an instance of the List≅Vec type.

Jenkins Arrays. The array object in Jenkins theory can be translated to Agda as follows.

```
data JArKind : Set where
  Scal Lst Arr : JArKind

data JAr : JArKind → Set

conform : JAr Lst → JAr Lst → Set

data JAr where
  zero : JAr Scal
  succ : JAr Scal → JAr Scal
  void : ∀ {k} → JAr k → JAr Lst
  hitch : ∀ {k} → JAr k → JAr Lst → JAr Lst
  reshape : (s : JAr Lst) → (l : JAr Lst) → conform s l → JAr Arr
```

Three kinds of arrays are being distinguished: natural numbers, lists and multi-dimensional arrays. The natural numbers are given by the usual two constructors zero and succ. The lists are given by constructors void and hitch which are almost nil and cons with the following important difference. As a solution to the problem that nested lists cannot properly represent tables, it is concluded that there must be an infinite number of empty lists. This is achieved by parametrising the empty list constructor with *the value*. This is quite different from the polymorphic lists and vectors — it is possible to create a value of type List ⊥ in Agda, but not in the above system. Finally, and most importantly, reshape is a constructor for multi-dimensional arrays. It is given by the shape which is a list and the flattened list representation of array elements. The conform relation ensures that the shape is a list of numbers, and that the length of the element list is the same as the product of the shape elements. Note that this system allows for arbitrarily nested arrays: hitch accepts any array as its first argument. This treatment of arrays being a "view" for lists can be found in many array languages, *e.g.* SaC and Qube.

Mullin. While Jenkins' theory is given in a constructive style, Mathematics of Arrays [Mullin 1988] is set-up in an observational style. The theory introduces a generalised array indexing function called Ψ, and then the meaning of array operations is given by the way the behaviour of the Ψ function changes when it is applied to the result of the operation. For example, an element-wise addition is defined as: $\vec{\imath}\Psi(a+b) \equiv (\vec{\imath}\Psi a) + (\vec{\imath}\Psi b)$. Essentially saying that for any arrays a and b of the same shape, an element at index $\vec{\imath}$ in the sum of a and b is the same as sum of elements found at index $\vec{\imath}$ in the arrays a and b (for any valid $\vec{\imath}$). The theory only deals with the homogeneous (non-nested) arrays and defines Ψ-based observational equalities for a large subset of the APL operators.

More. The idea of [More 1973] is to combine APL operations and set theory in order to obtain a mathematical theory of arrays. The theory is given in the axiomatic style that closely resembles ZFC, and it axiomatises a large subset of APL (it has 32 axioms and about a 100 theorems). Arrays in this theory can nest similarly to Jenkins' theory. The important difference is given at [More 1973, page 137], where we read:

> A restriction of indices to the finite ordinal numbers is a needless limitation that obscures the essential process of counting and indexing.

This exactly in line with our experience, and we will make this point precise in Section 5.

2.2 Encoding

We intend to use our Agda encoding as a minimalist array theory in which we could restate as many as possible results from the previously mentioned array theories. This means that our arrays have to be rank-polymorphic, and the shape of the array has to become an invariant. If we chose to keep this invariant at the level of types, (which makes sense) then a number of key operations immediately become dependently typed. For example, in operations such as reshape, take and drop, the value that carries shape impacts the type of the result. We have chosen to encode arrays as follows:

```
data Ix : (d : ℕ) → (s : Vec ℕ d) → Set where
  []   : Ix 0 []
  _::_ : ∀ {d s x} → Fin x → (ix : Ix d s) → Ix (suc d) (x :: s)

data Ar (X : Set) (d : ℕ) (s : Vec ℕ d) : Set where
  imap : (Ix d s → X) → Ar X d s
```

The Ar data type is indexed by the shape s which is represented as a Vector of natural numbers. The Ix type is a type of valid indices within the index-space generated by the shape s. The valid index in such an index-space is a tuple of natural numbers that is component-wise less than the shape s. Finally, the array with elements of type X is given by a function from valid indices to X.

In some sense Ar and Ix are second-order versions of Vec and Fin[5]. This could be also thought of as a computational interpretation of the Mathematics of Arrays (where Ψ becomes an array constructor), or generalisation of pull arrays [Svensson and Svenningsson 2014].

This encoding intrinsically guarantees that all the array accesses are within bounds. It has a satisfying property that any array operation composition normalises to an array constructed by a function composition. This lies at the essence of the SaC programming language and its with-loop construct. Finally, imap can be thought of as an abstract tag that indicates that a function at runtime has to be tabulated. However, the decision on how exactly this is done is completely opaque.

Finally, the Ar type can be represented without using index-value functions to store the elements as follows.

```
Tensor : ∀ {d} → Set → Vec ℕ d → ⊤ → Set
Tensor X [] = λ _ → X
Tensor X (x :: v) = (flip Vec x) ∘ Tensor X v

a-t : ∀ {X d s} → Ar X d s → Tensor X s tt
a-t {s = []} (imap a) = a []
a-t {s = x :: s} (imap a) = tabulate λ i → a-t $ imap λ iv → a (i :: iv)

sel : ∀ {X d s} → Ar X d s → Ix d s → X
sel (imap a) iv = a iv

t-a : ∀ {X d s} → Tensor X s tt → Ar X d s
t-a {s = []}   x = imap λ _ → x
t-a {s = x₁ :: s} x = imap λ { (i :: iv) → sel (t-a $ lookup x i) iv }
```

We used the name Tensor for the following reason: in mathematics, a contravariant tensor of rank $(m, 0)$ is defined as a tensor product of m vectors: $V \otimes V \otimes \cdots$. As it can be seen, if we replace tensor product with composition, Tensor is a d-fold composition of Vec x types (where x-s are

[5]The type Fin x describes the set of natural numbers that are less than x

corresponding shape elements). The resulting tensor is contravariant because in order to transform an array of shape *sm* into the one of shape *sn*, we need to provide a function that transforms indices in the opposite direction:

```
transf : ∀ {X m n sm sn} → (it : Ix n sn → Ix m sm) → Ar X m sm → Ar X n sn
transf it (imap f) = imap $ f ∘ it
```

2.3 Extensions

While the above encoding is very minimal, it can be used as a basic building block for more complicated array expressions. We demonstrate it this by giving types to two important concepts found in array programming languages: partitioning and nesting.

Partitioning. In [Svensson and Svenningsson 2014] push arrays are motivated with an observation that concatenation of two (one-dimensional) pull arrays requires executing a conditional check at every array index:

```
conc : ∀ {X m n} → Ar X 1 (m :: []) → Ar X 1 (n :: []) → Ar X 1 (m + n :: [])
conc {m = m} (imap a) (imap b) = imap body where body : _
                                                 body (i :: []) with splitAt m i
                                                 ... | inj₁ i' = a $ i' :: []
                                                 ... | inj₂ i' = b $ i' :: []
```

This is indeed the case, and the splitAt function in the above specification hides that very conditional. Executing such a specification directly on a conventional (sequential) hardware may be inefficient. While push arrays help here, there is an alternative solution, for example used in SaC, which is to introduce the ability to partition the index-space when defining an imap. In other words, to internalise the conditional on the index. Here is the type that makes this idea precise:

```
data Part (d : ℕ) : Vec ℕ d → Set where
  vec : (v : Vec ℕ d) → Part d v
  cut : ∀ {s} → (i : Fin d) → (k : ℕ)
      → Part d s
      → Part d (s [ i ]:= k)
      → Part d (s [ i ]%= (_+ k))

data ArP (X : Set) : (d : ℕ) → (s : _) → (Part d s) → Set where
  imap-vec : ∀ {d s} → (Ix d s → X) → ArP X d s (vec s)
  imap-cut : ∀ {d s i k pl pr} → (ArP X d s pl) → (ArP X d _ pr) → ArP X d _ (cut i k pl pr)
```

The ArP type says that an array either consists of a single partition (in which case it is just an Ar), or it is a concatenation of two arrays. Then Part ensures that the shapes of these two arrays are valid for the concatenation, *i.e.* the shapes differ at only one element at position *i*. The intuition here is that in order to concatenate two matrices, either the number of columns should match (in which case we concatenate them horizontally), or the number of rows should match (in which case we concatenate them vertically). The same holds for *n*-dimensional case, except we have *n* choices and not two.

Nesting. In APL, arrays of type Ar (Ar X n s_2) m s_1 and Ar X $(m + n)$ $(s_1 \mathbin{++} s_2)$ are indistinguishable. In our system these arrays are isomorphic, but we cannot abstract over the nesting. However, we can achieve this with yet another wrapper type that internalises nesting:

```
data Nest : (d : ℕ) → Set where
  done : ∀ {d s} → Part d s → Nest d
  nest : ∀ {dl dr} → Nest dl → Nest dr → Nest (dl + dr)

data ArNest (X : Set) : (d : ℕ) → Nest d → Set where
  imap-done : ∀ {d s pv} → ArP X d s pv → ArNest X d (done pv)
  imap-nest : ∀ {r rr l ll} → ArNest (ArNest X r rr) l ll → ArNest X (l + r) (nest ll rr)
```

The ArNest type says that nested array is either an ArP (a partitioned array), or a nested array of nested arrays.

Overall, the latter two constructions uses the standard dependently-typed technique of defining a structure that guides the induction, and then defining an indexed type over this structure. Similarly to Vec, where ℕ is an index, Vec is an index to Ar, the latter in an index to ArP, and so on. For example, we can envision defining a grid structure, or any other inductively definable traversal of array index-spaces.

However, it is important to notice, that all of thees constructions are really wrappers around the Ar type, which can be indeed treated as a basic building block of an array language/theory.

3 PROPERTIES OF FINITE ARRAYS

In this section we demonstrate how the proposed encoding can be used to specify typical array problems and reason about their behaviour. The imap-based encoding is especially useful when elements of the resulting array can be computed independently of each other. At the same time, dependent computation can be always recovered by means of recursion. Consider the specification of the matrix multiplication problem.

```
sum-ix₁ : ∀ {k} → (Ix 1 (k :: []) → ℕ) → ℕ
sum-ix₁ {zero} f = 0
sum-ix₁ {suc k} f = (f $ zero :: []) + sum-ix₁ λ where (i :: []) → f (suc i :: [])

matmul : ∀ {m k n} → Ar ℕ 2 (m :: k :: []) → Ar ℕ 2 (k :: n :: []) → Ar ℕ 2 (m :: n :: [])
matmul (imap a) (imap b) =
  imap λ where (i :: j :: []) → sum-ix₁
                     λ where (k :: []) → (a $ i :: k :: []) * (b $ k :: j :: [])
```

First we define how to compute a sum of the 1-dimensional array (its inner function, to be precise) of natural numbers. We do this recursively in the same way as one would define it for Vec. Note that the where after λ defines a pattern-matching lambda, so that we access components of the index. The matrix multiplication is defined on two-dimensional arrays and by binding the shape components to the variables m, n, and k we encode the expected shape of matrix multiplication. Observe that this specification guarantees well-behaved indexing of all arrays. Also note, that in this particular case there was no need in providing any proofs about compatibility of the indices.

As a next example let us consider a rank-polymorphic operation that rotates the vectors on the inner-most axis (similar the one we used in introduction). This can be defined in many different ways, but at the core of it we would have to traverse to the last axis and map 1-d rotation to all the elements. As we have noticed previously, we can cyrry/uncurry selection into arrays with respect to index components. Let us spice-up our arsenal of array operations with index curry.

```
ix-curry : ∀ {X : Set}{d s ss} → (f : Ar X (suc d) (s :: ss)) → (Fin s) → (Ar X d ss)
ix-curry (imap f) i = imap λ iv → f (i :: iv)
```

```
ix-uncurry : ∀ {X d s ss} → (Fin s → Ar X d ss) → Ar X (suc d) (s :: ss)
ix-uncurry f = imap λ where (i :: iv) → sel (f i) iv
```

The reason this works so seamlessly has to do with the Ix type being indexed by Vec and mimicing its structure. Therefore when Vec splits, the Ix must split in exactly the same way. The definitions of rotation follow.

```
rotatev : ∀ {X k} → Ar X 1 (k :: []) → ℕ → Ar X 1 (k :: [])
rotatev {k = zero} (imap a) r = imap λ where (() :: [])
rotatev {k = suc k} (imap a) r = imap λ where (i :: []) → a $ (toℕ i + r) mod (suc k) :: []

rotate : ∀ {X d s} → Ar X d s → ℕ → Ar X d s
rotate {s = []}      (imap a) r = imap a
rotate {s = x :: []} (imap a) r = rotatev (imap a) r
rotate {s = x :: y :: s} (imap a) r = ix-uncurry λ i → rotate (imap λ iv → a (i :: iv)) r
```

The rotatev function falls into two cases: when the array is empty (is of shape 0 :: []) and when it contains some elements. In the first case (empty 1-d array), when constructing an imap, we have to produce a function from the index where with the first component of type Fin 0 (this type is uninhabited). However, there always exists a function (exactly one) from the empty type to any given type. In order to construct such a function we need to use an eliminator for the empty type. This can be done explicitly, or as in the code above, by pattern matching on the index, in which case Agda is able to figure out that the first component could not possibly exist, in which case the absurd pattern '()' completes the definition.

In the second case rotatev uses modular arithmetic to rotate the array to the left. Notice that the definition of mod requires the second argument to be non-zero. As this is trivially true, the proof obligation is discharged automatically.

Finally, rotate splits into three cases. For 0-dimensional arras the value stays untouched — no matter by how much we rotate it, there is only one position in this array, therefore we can only use identity map. For 1-dimensional arrays we use rotatev. For arrays with two an more dimensions we recursively apply rotate to all subarrays.

Let us now consider generalisation of the above pattern used in rotate that is found extremely often in array programming: mapping a function over the last k axes of the array. In APL this is commonly referred as rank operator [Bernecky 1987], and in our framework this can be easily achieved by means of nesting. Let us first define a map operator.

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

This is straight-forward to do using the imap constructor. Furthermore, we can define an operation that turns an array of shape $(l \mathbin{++} r)$ into a nested array of shape l, where elements are arrays of shape r. We start with a few helper functions that are counterparts for ++, take and drop for vectors.

```
_ix++_ : ∀ {l r ll rr} → (iv : Ix l ll) → (jv : Ix r rr) → Ix (l + r) (ll ++ rr)
take-ix : ∀ {l r ll rr} → Ix (l + r) (ll ++ rr) → Ix l ll
drop-ix : ∀ {l r ll rr} → Ix (l + r) (ll ++ rr) → Ix r rr
```

The ix++ concatenates two indices within the shape ll and rr into an index within the index-space $(ll \mathbin{++} rr)$. The other two functions are left and right inverses to the concatenation: take-ix takes the left part of the index in the index-space $(ll \mathbin{++} rr)$ and drop-ix takes the right part of the index in the same space.

Using these functions we can define general nesting and flattening operations as follows.

```
nest : ∀ {X l r ll rr} → Ar X (l + r) (ll ++ rr) → Ar (Ar X r rr) l ll
nest (imap a) = imap λ iv → imap λ jv → a $ iv ix++ jv

unnest : ∀ {X l r ll rr} → Ar (Ar X r rr) l ll → Ar X (l + r) (ll ++ rr)
unnest (imap a) = imap λ iv → sel (a $ take-ix iv) (drop-ix iv)
```

The nest operation is similar to currying: we force to supply indices to the array in two parts, and the lengths of the corresponding indices are given by *ll* and *rr*. As currying comes with uncurry operation, unnest is the inverse operation to nest.

Finally, here is an alternative definition of the rotate function that nests an array, maps the rotatev function and unnests it back.

```
rotate-map : ∀ {X d s} → Ar X d s → ℕ → Ar X d s
rotate-map {d = zero} a r = a
rotate-map {d = suc d} {s = s} a r rewrite (+-comm 1 d) with splitAt d s
... | ll , (k :: []) , refl = unnest $ map (flip rotatev r) $ nest a
```

It is defined by cases on the rank *d*: for zero-dimensional arrays we return the array itself, and for (1 + *d*)-dimensional ones we map the function over the last dimension. Let us decipher what exactly is happening. As nest operates on arrays of type Ar *X* (*l* + *r*) (*ll* ++ *rr*), we need to transform the type of our input array (Ar *X* (1 + *d*) *ss*) into the right form. We do this by using the theorem +-comm that states that addition is commutative. The rewrite of this theorem says that we replace all the occurrences of (1 + *d*) with (*d* + 1). As a result the type of the input (and the output!) arrays become (Ar *X* (*d* + 1) *s*). Recall from the *Ar* construction that the type of *s* also became Vec ℕ (*d* + 1). The splitAt *d s* function splits the vector (*d* + *k*) into two vectors *ll* and *rr* of the types Vec ℕ *d* and Vec ℕ 1, and returns a proof that ll ++ *rr* is equal to *s*. When pattern-matching on this proof the types of the input/output arrays are refined to Ar *X* (*d* + 1) (*ll* ++ *rr*). Therefore we can apply nest to *a*.

Note that we did not need to define a special case for 1-dimensional arrays, as 1 = 0 + 1 and consequently *s* = [] ++ *s*. That is a one-dimensional array can be turned into a zero-dimensional array of 1-dimensional arrays. More generally, we can always enclose an element of some type into a zero-dimensional array of that type:

```
enc : ∀ {X : Set} → X → Ar X 0 []
enc a = imap λ _ → a
```

Note that we do not lose or duplicate any information here: a zero-dimensional array has exactly one position and the element can be retrieved by indexing with the index [].

```
enc-thm : ∀ {X : Set} (x : X) → sel (enc x) [] ≡ x
enc-thm x = refl
```

At this moment we hope that the reader is convinced that the proposed array types are strong enough to allow for most of the rank-polymorphic APL-like expressions. If in doubts, we invite a reader to inspect the APL module in the supplementary materials where we define a large enough set of APL operators so that we can encode a convolutional neural network (found in supplementary materials as well).

3.1 Properties

Let us now represent a few array-theoretical properties that can be observed in the proposed framework. We start with the definition of array equality. In all the three mentioned array theories array equality is defined extensionally.

```
_=a_ : ∀ {X : Set}{d s} → Ar X d s → Ar X d s → Set
imap f =a imap g = ∀ iv → f iv ≡ g iv
```

We can straight-forwardly show that this relation is reflexive, symmetric and transitive: simply because the propositional equality ≡ is.

```
refl-=a  : ∀ {X : Set}{d s}{x  : Ar X d s} → x =a x
sym-=a : ∀ {X : Set}{d s}{l r : Ar X d s} → l =a r → r =a l
trans-=a : ∀ {X : Set}{d s}{x y z : Ar X d s} → x =a y → y =a z → x =a z
```

Note that as standard Agda uses intensional type theory, =a will not be substitutitive, *i.e.* there will be no way to prove that:

```
subst-a : ∀ {X d s Y d' s'} → (f : Ar X d s → Ar Y d' s') → ∀ a b → a =a b → (f a) =a (f b)
```

This happens because the intensional type theory does not admit functional extensionality. This is a well-known problem, and as a simple way to overcome this one might either postulate functional extensionality (locally or globally). An alternative solution would be switching to cubical Agda [Vezzosi et al. 2019], but this development is left as a future work.

Lifting relations. Unsurprisingly, any element-wise relation can be generalised for the entire arrays.

```
ARel : ∀ {X : Set} → (P : X → X → Set) → ∀ {d s} → Ar X d s → Ar X d s → Set
ARel p (imap x) (imap y) = ∀ iv → p (x iv) (y iv)
```

As before, we define the new array relation as a pointwise relation given by P on all the corresponding elements.

A more interesting fact about array relations is that we can generally show that if P is decidable, then ARel P is also decidable. Here is a function that builds a decision procedure for ARel P from the decision procedure on P.

```
mk-dec-arel : ∀ {X : Set} → (p : X → X → Set) → Decidable p
            → ∀ {d s} → Decidable (ARel p {d = d} {s = s})
```

The essence of this construction lies in defining an inductive traversal through the index-space of both arrays, using the previously defined ix-curry function. In order to state that all the elements in the entire array are related, we check that all the subarrays (formed by currying on the first axis) are related. The base case of the induction is an empty array, for which the decision procedure is trivial. The step of the induction applies the procedure to all the sub-arrays and then uses the following function to build either an evidence or a refutation.

```
check-all-subarrays : ∀ {d s ss}{X : Set}{P : X → X → Set}
  → let _~a_ = ARel P in
      (a b : Ar X (suc d) (suc s :: ss))
  → (∀ i → Dec (ix-curry a i ~a ix-curry b i))
  → (Σ (Fin (suc s)) λ i → ¬ (ix-curry a i ~a ix-curry b i))
  ⊎ (∀ i → (ix-curry a i ~a ix-curry b i))
```

This says: if we decided the relation on all the sub-arrays, then we can either give an index at which the relation does not hold or it holds for all of them. With an extra bit of plumbing we turn this disjoint union into yes or no answer.

Reshape. In all the three array theories the reshape operator is amongst the core primitives, or as in case of Jenkins' theory, an array constructor. This operation is indeed at the core of array computations — it is very useful to have a universal way to change the shape and possibly the rank of an array.

All three theories agree, that reshape has to be constrained with the axiom:

$$a \equiv \text{reshape } (\text{shape } a))$$

which is pretty much common sense. If we assume that arrays of shapes s and s' have the same number of elements, Axiom 19 in Moore's theory states that reshape is contractible:

$$\text{reshape } s \text{ } (\text{reshape } s' \text{ } a) \equiv \text{reshape } s \text{ } a$$

We could not help noticing that these two axioms very much remind a structure of an indexed monad:

```
Sh = Σ ℕ λ n → Vec ℕ n
Arr (d , s) X = Ar X d s

compat : Sh → Sh → Set
compat (d₁ , s₁) (d₂ , s₂) = prod s₁ ≡ prod s₂

record IM {X : Set} (F : (a b : Sh) → compat a b → Arr b X → Arr a X) : Set where
  field
    return : ∀ {i a} → a =a F i i refl a
    join : ∀ {i j k ij jk a} → F i j ij (F j k jk a) =a F i k (trans ij jk) a
```

Indeed, an element-preserving reshape operation can be thought of as an indexed functor, within indices being shapes, on a category of arrays. This functor is bound by the "monadic" laws. Notice the contravariant nature of F, exactly the same contravariancy is observed in morphisms on the category of containers [Abbott et al. 2005], as well as in our definition of transf in Section 2.2.

The second observation is that in case we accept the "join" axiom, reshaping can be defined in terms of flattening. Flattening is an *element-preserving* reshape to the array of rank one. In this case reshape can be defined as an inverse of the flattening:

$$\text{reshape } s \text{ } (\text{flatten } a) \equiv \text{reshape } s \text{ } a \qquad \text{Ax 19}$$
$$\text{unflatten } s \text{ } (\text{flatten } a) \equiv \text{reshape } s \text{ } a$$

Once again, element-preserving reshape can be defined in many ways that would not satisfy the join axiom, but it does if we define it via flattening.

While all three array theories agree on reshaping from a canonical flattening, they disagree on the compatibility constraint. The main difficulty is to decide what is the behaviour if the number of elements in the array and the number of elements in the argument shape do not match. In case both shapes are non-empty, all three theories agree that the flattened list is adjusted to match the argument shape. In case there is too much elements, the list is truncated, in case there is to little elements, the list is extended by cycling the original list. The difficult question is what to do when empty arrays are reshaped into a non-empty ones. Mullin's theory defines reshape by case analysis and rules out this possibility: if the argument shape is non-empty, the array shape must be non-empty as well. In Jenkins' theory empty lists carry "default element", therefore this element is replicated as many times as needed. More says that reshaping (into a rank-1 array) forms a list by cycling through the flattened input array so many times. It is not guaranteed that the shape of the reshaped result will be of the shape supplied as the argument, therefore for empty lists, reshape is vacuously defined.

In the proposed formalism, reshape is not a primitive, therefore all the three behaviours (except the builtin default element) can be defined. It is worth mentioning that the unusual behaviour of array operators may result from the fact that all the three theories are untyped. Therefore, there is a natural desire to make most of the functions total, even if one has to "patch" some of the corner cases. With dependent types there is no real need to do this, as all the assumptions can be encoded in types.

Finally, if the reshaping operation is defined via flattening, we have to choose a canonical one. All three array theories agree that the row-major ordering is the canonical one. This is indeed a good choice, as lexicographical ordering of indices gives a rise to the following inductive definition:

```
flatv : ∀ {X d s} → Ar X d s → Vec X (prod s)
flatv {d = zero} {s = []} (imap a) = (a []) ∷ []
flatv {d = suc d} {s = s ∷ ss} a = foldr (λ i → Vec _ (i * _)) _++_ [] (tabulate (flatv ∘ ix-curry a))
```

This homomorphism is saying that flattening is a concatenation of flattened subarrays.

We define flattening/unflattening operation using the following two functions:

```
off→idx : ∀ {d} s → Ix 1 (prod s ∷ []) → Ix d s
idx→off : ∀ {d} s → Ix d s → Ix 1 (prod s ∷ [])

flatten : ∀ {X d} s → Ar X d s → Ar X 1 (prod s ∷ [])
flatten s (imap a) = imap $ a ∘ off→idx s
unflatten : ∀ {X d} s → Ar X 1 (prod s ∷ []) → Ar X d s
unflatten s (imap a) = imap $ a ∘ idx→off s
```

The following two theorems state that we can switch between the flattened view and back.

```
io-oi : ∀ {d}{s : Vec _ d}{iv : Ix 1 (prod s ∷ [])} → idx→off s (off→idx s iv) ≡ iv
oi-io : ∀ {d}{s : Vec _ d}{iv : Ix d s} → off→idx s (idx→off s iv) ≡ iv
```

The definition of reshape and the theorems ensuring that reshape behaves according to the IM rules follow.

```
reshape : ∀ {X d d' s} → (s' : Vec ℕ d') → Ar X d s → (prod s' ≡ prod s) → Ar X d' s'
reshape {s = s} s' (imap a) pf = imap λ iv → a (off→idx s (subst (λ x → Ix 1 (x ∷ [])) pf (idx→off s' iv)))

reshape-thm : ∀ {X d s}{a : Ar X d s} → reshape s a refl =a a
reshape-thm {a = imap a} iv rewrite oi-io {iv = iv} = refl

reshape-join : ∀ {X d₁ d₂ d₃ s₃}{s₁ : Vec _ d₁}{s₂ : Vec _ d₂}{s23 : prod s₂ ≡ prod s₃}{s12 : prod s₁ ≡ prod s₂}
             → (a : Ar X d₃ s₃) → reshape s₁ (reshape s₂ a s23) s12 =a reshape s₁ a (trans s12 s23)
reshape-join {s₃ = s₃}{s₁ = s₁}{s₂ = s₂}{s23 = s23}{s12 = s12} (imap a) iv
  rewrite (io-oi {s = s₂} {iv = subst (λ x → Ix 1 (x ∷ [])) s12 (idx→off s₁ iv)})
    = cong a (cong (off→idx s₃) (subst-subst s12))
```

4 INFINTE COUNTERPARTS

We have seen that rank-polymorphic arrays are "views" of the Vec type. The structure of this view is tightly coupled with the idea of nesting, and the row-major flattening is a way to convert from arrays to Vecs.

As we are interested in extending arrays to support infinite collections of ordered data, we investigate whether it would be possible to replicate this "view" approach to the infinite counterparts of Vec, preserving typical array properties.

We leave the term infinite counterpart a bit loose, as our main goal here is not a formal duality between data structures, but rather a common interface between finite and infinite arrays. In the rest of the section we consider three coinductive data structures: Streams, Colists and Covectors.

Coinductive structures. The main mechanism for defining coinductive data structures in Agda is via records that are tagged with coinductive. This instructs Agda to use greatest fixed point for recursive records, and check for productivity rather than for termination. In order to facilitate productivity checks, Agda introduces the notion of sized types [Abel 2012] which gives a way to attach a measure to the size of the term, and possibly refer to it in definitions. In the standard library coinductive data structures are typically defined with the help of the record called Thunk:

```
record Thunk {a} (F : Size → Set a) (i : Size) : Set a where
  coinductive
  field force : {j : Size< i} → F j
```

It is a 1-element record that ensures that its content is size-wise smaller than the entire record. To match the standard library definitions we use thunks in the presentation as well.

Streams. The first infinite data type we consider is Stream that is defined as follows:

```
data Stream (A : Set) (i : Size) : Set where
  _::_ : A → Thunk (Stream A) i → Stream A i
```

In some sense, it is a List that does not have the [] case, thus representing *only* infinite sequences. Unsurprisingly, there exists a number of List operations that can be defined on Streams:

```
maps : ∀ {i}{X Y} → (X → Y) → Stream X i → Stream Y i
maps f (x :: xs) = f x :: λ where .force → maps f (xs .force)

zips : ∀ {i}{X Y} → Stream X i → Stream Y i → Stream (X × Y) i
zips (x :: xs) (y :: ys) = (x , y) :: λ where .force → zips (xs .force) (ys .force)
```

As Stream X is isomorphic to $\mathbb{N} \to X$ (do not have upper bound), array shapes are determined only by the number of dimensions. Therefore we can define rank-polymorphic infinite arrays as:

```
data Ar-s (X : Set) (d : ℕ) : Set where
  imap : (Vec ℕ d → X) → Ar-s X d
```

Note that indices into such an array are vectors of natural numbers (and not bound vectors Ix d s as in Ar) — any d-element tuple of $\mathbb{N}$ is a valid index. Similarly to Ar, we can represent the above infinite array as d-fold nested stream and convert between the two notations:

```
Tensor-s : Set → ℕ → Set
Tensor-s X zero = X
Tensor-s X (suc d) = Stream (Tensor-s X d) ∞

froma : ∀ {X i} → (ℕ → X) → Stream X i
froma f = f 0 :: λ where .force → froma (f ∘ suc)

as-ts : ∀ {X d} → Ar-s X d → Tensor-s X d
as-ts {d = zero} (imap a) = a []
as-ts {d = suc d} (imap a) = froma λ i → as-ts (imap λ iv → a $ i :: iv)
```

Note the built-in size ∞ used in the definition of Tensor-s — it may be interpreted as: "the size of the largest term that one could possibly build in Agda".

While the obtained array type is rank-polymorphic, it does not serve as a good unifying array, as it cannot represent finite arrays. Let us now consider Colists and Covec types.

4.1 Colist and Covec

Colist can be thought of a Stream with a missing [] case:

```
data Colist (A : Set) (i : Size) : Set where
  [] : Colist A i
  _::_ : A → Thunk (Colist A) i → Colist A i
```

As with Lists and Vecs, we can define an indexed version of the Colist called Covec, where the index is the length. Similarly to the List/Vec pair, the structures are isomorphic (bisimilar). This may be verified by looking at fromColist and toColist functions in the standard library. In the remaining of the section we only show constructions on Covecs. As the length of Covec may be infinite, conatural numbers are used as indices:

```
data Conat (i : Size) : Set where
  zero : Conat i
  suc : Thunk Conat i → Conat i
```

As the element of suc is a Thunk, we are able to define a conatural number infinity as follows:

```
infinity : ∀ {i} → Conat i
infinity = suc λ where .force → infinity
```

This says that infinity is a number whose successor is the number itself. Then the definition of the Covec is:

```
data Covec (A : Set) (i : Size) : Conat ∞ → Set where
  [] : Covec A i zero
  _::_ : ∀ {n} → A → Thunk (flip (Covec A) (n .force)) i → Covec A i (suc n)
```

The flip in the Thunk swaps the length and the size arguments of the Covec so that the Thunk type is satisfied. As a demonstration that Covec admits finite and infinite data, consider the following two definitions: a two element vector (42, 43) and infinite vector of 42s.

```
v42-43 : Covec ℕ ∞ (fromℕ 2)
v42-43 = 42 :: (λ where .force → 43 :: (λ where .force → []))

inf42 : ∀ {i} → Covec ℕ i infinity
inf42 = 42 :: λ where .force → inf42
```

Similarly we can define a function that converts from Vec to Covec as well as from Stream to Covec and back, given that the length is finite or infinite:

```
v-cov : ∀ {X : Set}{n} → Vec X n → Covec X ∞ (fromℕ n)
v-cov [] = []
v-cov (x :: a) = x :: λ where .force → v-cov a

s-cov : ∀ {X : Set}{i} → Stream X i → Covec X i infinity
s-cov (x :: xs) = x :: λ where .force → s-cov (xs .force)

cov-v : ∀ {a}{n}{X : Set a} → (pf : Finite n) → Covec X ∞ n → Vec X (toℕ pf)
cov-v zero [] = []
cov-v (suc pf) (x :: xs) = x :: cov-v pf (xs .force)
```

```
cov-s : ∀ {a}{X : Set a}{i} → Covec X i infinity → Stream X i
cov-s (x :: xs) = x :: λ where .force → cov-s (xs .force)
```

Note the Finite predicate in the cov-v function which ensures that the conatural number (the length of the vector) is finite. Let us now show how Covec can be used to define rank-polymorphic arrays. We define Cofin type which is a version of Fin for conatural numbers. We leave this definition as an exercise (the answer can be found in the standard library). In this case:

```
data CoIx : (d : ℕ) → (s : Vec (Conat ∞) d) → Set where
  [] : CoIx 0 []
  _::_ : ∀ {d s x} → Cofin x → (ix : CoIx d s) → CoIx (suc d) (x :: s)

data CoAr {a} (X : Set a) (d : ℕ) (s : Vec (Conat ∞) d) : Set a where
  imap : (CoIx d s → X) → CoAr X d s
```

The same using nested Covec:

```
Tensor-co : ∀ {n} → Set → Vec (Conat ∞) n → Set
Tensor-co X []      = X
Tensor-co X (x :: v) = Covec (Tensor-co X v) ∞ x

cotabulate : ∀ {X : Set}{n i} → (Cofin n → X) → Covec X i n
cotabulate {n = zero} f = []
cotabulate {n = suc x} f = (f zero) :: (λ where .force → cotabulate (f ∘ suc))

nest-co : ∀ {X d s} → CoAr X d s → Tensor-co X s
nest-co {d = zero} {[]} (imap a) = a []
nest-co {d = suc d} {x :: s} (imap a) = cotabulate λ i → nest-co $ imap λ iv → a (i :: iv)
```

4.1.1 *Restrictions.*

While CoAr indeed extends Ar, it is worth noticing a number of restrictions that the more general data structure brings. First, no matter how we encode infinite arrays, we would have to give up operations like sum of all elements, reverse, *etc.*. The good thing is that Agda would not allow us to define such operations as there is no proof that such a computation terminates.

Secondly, relations on conatural numbers are *undecidable*. For example, there is no constructive algorithm that checks for equality of two conatural numbers as one (or both) of them might be infinite. Moreover, all the reasoning on coinductive data structures has to happen via bisimulation. That is, we have to build an argument that two objects have the same observable behaviour. For example consider a proof that adding a number to infinity is bisimilar to the infinity:

```
∞+m : ∀ {i m} → i ⊢ (infinity + m) ≈ infinity
∞+m {m = m} = suc λ where .force → ∞+m {m = m}

m+∞ : ∀ {i m} → i ⊢ (m + infinity) ≈ infinity
m+∞ {m = zero} = refl
m+∞ {m = suc m} = suc λ where .force → m+∞ {m = m .force}
```

Similarly to functional extensionality, bisimilarity is an equivalence relation, but we cannot substitute bisimilar things for the bisimilar ones in the intuitionistic type theory.

The above fact about adding to infinity tells us that array reshaping defined via row-major flattening would not work for the following reason:

```
∞-++ : ∀ {a}{i m}{X : Set a}
       → (l : Covec X ∞ infinity)
       → (r : Covec X ∞ m) → i , infinity ⊢ (cast ∞+m (l ++ r)) ≈ l
∞-++ (x :: xs) r = refl :: λ where .force → ∞-++ (xs .force) r
```

This theorem says that concatenation of l and r, where l is infinite, is bisimilar to l. This means that elements of r will be "lost forever". Recall that reshaping axioms say that we can flatten the array and reshape it back into the same shape and get back the same array. Now consider what happens if the shape of the array is (2 :: infinity :: []). We may flatten the array (in row-major order) into a 1-d array of length (2 * infinity = infinity). In order to unflatten we would need to take infinity many elements from the flattening *twice*. But there will be no elements left after the first take.

Disclaimer. It is worth noting that we are not claiming that such a lack of invertible flattening makes it impossible to develop an array theory. We can envision that flattening could be defined by means of diagonalisation, and probably it can be shown that it is invertible. However, there is no array theory known to us that permits this. Also, if we scale our experience of diagonalising empty tables (Section 2.1), the infinite case may be infinitely harder to handle. By no means we suggest that it should not be done, but before engaging with this research we explore row-major-preserving alternatives first.

5 TRANSFINITE ARRAYS

This chapter describes the key contribution of the paper: observing that the use of ordinals as indices naturally extends finite rank-polymorphic arrays with row-major flattening into an infinite domain. Here we make this idea precise.

Formal Definition. We start with a very brief formal introduction of ordinals, but due to the lack of space we are not providing formal definitions of ordinal operations. For references on ordinals consider [Ciesielski 1997; Manolios and Vroon 2005].

A totally ordered set $\langle A, < \rangle$ is said to be well ordered if and only if every nonempty subset of A has a least element [Ciesielski 1997]. Given a well-ordered set $\langle X, < \rangle$ and $a \in X$, $X_a \overset{\text{def}}{=} \{x \in X | x < a\}$. An ordinal is a well-ordered set $\langle X, < \rangle$, such that: $\forall a \in X : a = X_a$. As a consequence, if $\langle X, < \rangle$ is an ordinal then $<$ is equivalent to $\in$. Given a well-ordered set $A = \langle X, < \rangle$ we define an ordinal that this set is isomorphic to as $Ord(A, <)$. Given an ordinal α, its successor is defined to be $\alpha \cup \{\alpha\}$. The minimal ordinal is $\emptyset$ which is denoted with 0. The next few ordinals are:

$$\begin{array}{lllll} 1 & = & \{0\} & = & \{\emptyset\} \\ 2 & = & \{0, 1\} & = & \{\emptyset, \{\emptyset\}\} \\ 3 & = & \{0, 1, 2\} & = & \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ & & \dots & & \end{array}$$

A limit ordinal is an ordinal that is greater than zero that is not a successor. The set of natural numbers $\{0, 1, 2, 3, \dots\}$ is the smallest limit ordinal that is denoted ω.

5.1 Church Encoding

An alternative way to understand ordinals is via their Church encodings. Similarly to the familiar church-encoded natural numbers, the same can be done with ordinals, and it is instructional to look at the two encodings side-by-side.

```
N = {X : Set} → (X → X) → X → X

nadd nmul nexp : N → N → N
nadd a b s z = b s (a s z)
nmul a b s = b (a s)
nexp a b {X} = b {X → X} a

nzer : N; nsuc : N → N
nzer s z = z
nsuc a s z = s (a s z)

(f ** zero) x = x
(f ** suc n) x = f ((f ** n) x)
```

```
O = {X : Set} → ((ℕ → X) → X) → (X → X) → X → X

oadd omul oexp : O → O → O
oadd a b l s z = b l s (a l s z)
omul a b l s = b l (a l s)
oexp a b {X} l = b {X → X} (λ f x → l λ n → f n x) (a l)

ozer : O; osuc : O → O; olim : (ℕ → O) → O
ozer l s z = z
osuc x l s z = s (x l s z)
olim f l s z = l λ n → f n l s z

oω = olim λ n → (osuc ** n) ozer
```

Both encodings are given in the impredicative style — natural numbers on the left and ordinals on the right. As it can be seen, the ordinal type has an additional limit "constructor" that makes it possible to form infinite sequences. The arithmetic operations are defined very similarly, except ordinals have to deal with the limit case. In case of addition and multiplication it is just passed by, and in case of exponentiation, it is turned into a pointwise limit. At the end of right column we define the first limit ordinal ω as a limit of iterated successors (the iteration of successors is given at the end of the left column).

5.2 Ordinals, CNF

While Church ordinals gives a convenient understanding of ordinals as iterators, the presence of the (ℕ → X) function that denotes sequences makes operations like comparison undecidable. A typical workaround is to use some well-behaved ordinal notation system for a chosen initial segment of ordinals. One of the commonly used ordinal notation systems is called Cantor Normal Form (CNF). This can be thought of as a hereditary ω-based positional numeral system. More formally, every ordinal α can be *uniquely* represented as:

$$\alpha = \omega^{\beta_1} c_1 + \omega^{\beta_2} c_2 + \cdots + \omega^{\beta_k} c_k$$

where $\beta_1 > \beta_2 > \cdots > \beta_k \geq 0$ and $c_i > 0$. The ordinal 0 has a cantor normal form with $k = 0$. The highest exponent (also called the degree of α) satisfies $\beta_1 \leq \alpha$. However, for the cases when $\alpha < \epsilon_0$, this inequality becomes strict, *i.e.* $\beta_1 < \alpha$. The ordinal ϵ_0 is the least solution to the equation $\alpha = \omega^\alpha$, or a sequence $\{\omega, \omega^\omega, \omega^{\omega^\omega}, \omega^{\omega^{\omega^\omega}}, \ldots\}$.

Now we demonstrate the encoding of the CNF in Agda. The interesting thing about this encoding is that we have to simultaneously define the structure and the relation on this structure.

```
data Ordinal : Set

record OrdTerm : Set where
  inductive; constructor ω^_·_⟨_⟩
  field
    exp : Ordinal
    k : ℕ
    .k>0 : k > 0
```

The dots like the one in front of k>0 indicate irrelevance of the argument at runtime and simplifies some of the proofs. In the context of this paper the dots can be ignored — we keep them here so that the code matches the supplementary materials. We start with the definition of the OrdTerm which represents the term $\omega^{\exp} k$ with the additional condition that k is positive. Then the Ordinal later will happen to be a list of OrdTerms. Before that we declare existence two relations: $>_e$ and $<_o$. The former compares the exponent of the term (on the left) and the degree of the ordinal on the right. The $<_o$ is a regular ordinal comparison.

```
data _>ₑ_ : OrdTerm → Ordinal → Set
data _<ₒ_ : Ordinal → Ordinal → Set
_>ₒ_ : Ordinal → Ordinal → Set
a >ₒ b = b <ₒ a
```

The Ordinal is a list of terms, where the head of the list must be greater ($>_e$) than the degree of the tail. This implements the exponent ordering required by the CNF.

```
data Ordinal where
  [] : Ordinal
  _::_⟨_⟩ : (x : OrdTerm) → (xs : Ordinal) → .(x >ₑ xs) → Ordinal
```

The exponent comparison is a wrapper around the ordinal comparison: it says that any ordinal term has a greater exponent than zero; or the term exponent is greater than the degree of the ordinal when the exponent of the term is greater than the exponent of the first element in the list.

```
data _>ₑ_ where
  zz : ∀ {t} → t >ₑ []
  ss : ∀ {t o l}.{pf} → exp t >ₒ exp o → t >ₑ (o :: l ⟨ pf ⟩)
```

Comparison of two ordinals falls into four cases: zero is less than any ordinal term; degree of the left ordinal is less than the degree of the right one; degree is the same but the left coefficient is less than the right one; and finally, degrees and coefficients are equal, but the left tail is less than the right one.

```
data _<ₒ_ where
  z< : ∀ {t l}.{pf} → [] <ₒ t :: l ⟨ pf ⟩
  e< : ∀ {t t₁ l l₁}.{pf pf₁} → exp t <ₒ exp t₁ → t :: l ⟨ pf ⟩ <ₒ t₁ :: l₁ ⟨ pf₁ ⟩
  k< : ∀ {t t₁ l l₁}.{pf pf₁} → exp t ≡ exp t₁ → k t < k t₁ → t :: l ⟨ pf ⟩ <ₒ t₁ :: l₁ ⟨ pf₁ ⟩
  t< : ∀ {t t₁ l l₁}.{pf pf₁} → exp t ≡ exp t₁ → k t ≡ k t₁ → l <ₒ l₁ → t :: l ⟨ pf ⟩ <ₒ t₁ :: l₁ ⟨ pf₁ ⟩
```

Here we assign readable names to the ordinals zero, one and ω. In the proofs for ordinals 1_o and ω it says that 1 is positive and the second proof at the end says that the term is greater ($>_e$) than the tail, which is trivially true.

```
pattern 0ₒ = []
pattern 1ₒ = ω^ 0ₒ · 1 ⟨ s≤s z≤n ⟩ :: [] ⟨ zz ⟩
pattern ωₒ = ω^ 1ₒ · 1 ⟨ s≤s z≤n ⟩ :: [] ⟨ zz ⟩
```

5.3 Operations

We make a brief overview of the operations and their properties that we define in our formalisation.

As CNF ordinals are essentially trees, comparison of the elements is decidable. This is very valuable for the upcoming array formalisms, as array operations may chose to behave differently on finite/infinite shape values, limit values, *etc.*

```
_≟ₒ_ : Decidable (_≡_ {A = Ordinal})
_<ₒ?_ : Decidable (_<ₒ_); _>ₒ?_ : Decidable (_>ₒ_)
```

```
_≥ₒ_ : Ordinal → Ordinal → Set
a ≥ₒ b = a >ₒ b ⊎ a ≡ b
```

We define usual arithmetic operations on ordinals of the following types:

```
_+ₒ_ : Ordinal → Ordinal → Ordinal
_*ₒ_ : Ordinal → Ordinal → Ordinal
_-ₒ_ : (a b : Ordinal) → .{≥ : a ≥ₒ b} → Ordinal
_divmodₒ_ : (a b : Ordinal) → .{≠0 : b ≢ 0ₒ} → Ordinal × Ordinal
```

Note that subtraction and division/modulo require an extra argument: a proof that the arguments are compatible. Our mechanisation is standard and is similar to the one found in [Manolios and Vroon 2005], except division/modulo, which we have never seen begin mechanised before. However, as division is the key to the row-major reshapes, we have to provide it.

Addition and multiplication on ordinals are famously non-commutative. If we recall the formal definition, then addition is a concatenation of two orders. Now consider $2 + \omega$ and $\omega + 2$. In the first case we concatenate $0' < 1' < 0 < 1 < \dots$, in the second we have $0 < 1 < 2 < \cdots < 0' < 1'$. In the first case only $0'$ does not have immediate predecessor, so if we relabel the elements we get ω; however in the second case, both 0 and $0'$ have no immediate predecessors. As multiplication is defined via addition, it is non-commutative either. This means that subtraction and division can be defined in two different ways (on the left and on the right). In our formalisation we only define left subtraction and left division; its correctness is verified by the following two theorems:

```
b+a-b≡b : ∀ {a b} → (≥ : a ≥ₒ b) → b +ₒ (a -ₒ b) {≥} ≡ a
divmod-thm : (x y : Ordinal) → (≠0 : y ≢ 0ₒ) → let p , q = (x divmodₒ y) {≠0} in (y *ₒ p) +ₒ q ≡ x
x%y<y : (x y : Ordinal) → (≠0 : y ≢ 0ₒ) → let p , q = (x divmodₒ y) {≠0} in q <ₒ y
```

In the supplementary materials we define much more (about 20) theorems about properties of ordinal arithmetics and comparisons. Facts like associativity of addition, continuity of addition on the right, trichotomy of comparison, distributivity of multiplication on the left, and many more. The basic goal is to define enough facts so that we could state row-major flattening. While ti is a non-trivial job to convince Agda that these facts hold, most of them can be found in math books about set theory. To save some space, we omit the full details on these arithmetic facts.

5.4 Transfinite Arrays

By now we have enough equipment to define transfinite arrays and consider a few basic examples. First, we define the OFin type, which is a version of Fin that is indexed by ordinals:

```
record OFin (u : Ordinal) : Set where
  constructor _bounded_
  field
    v : Ordinal
    .v<u : v <ₒ u
```

We do this in a refinement type style: instead of new inductive definition, we pair-up the ordinal and the proof that its value is less than the upper bound. As before, ignore the irrelevance annotation at v<u. After that, transfinite arrays are defined by literally replacing natural numbers with ordinals and Fin with OFin:

```
data Ix : (d : ℕ) → (s : Vec Ordinal d) → Set where
  []   : Ix 0 []
  _::_ : ∀ {d s x} → OFin x → (ix : Ix d s) → Ix (suc d) (x :: s)
```

```
data Ar (X : Set) (d : ℕ) (s : Vec Ordinal d) : Set where
  imap : (Ix d s → X) → Ar X d s
```

Consider now a few simple examples that we can write in this system. We start with a list operations head and tail that we reimplement for 1-dimensional arrays in infinity-agnostic style:

```
hd : ∀ {X : Set}{n} → .(n≥1 : n ≥ₒ 1ₒ) → Ar X 1 (n :: []) → X
hd n≥1 (imap a) = a (0ₒ bounded n≥1⟹0<n n≥1 :: [])

tl : ∀ {X : Set}{n} → .(n≥1 : n ≥ₒ 1ₒ) → Ar X 1 (n :: []) → Ar X 1 ((n -ₒ 1ₒ) {n≥1} :: [])
tl n≥1 (imap a) = imap λ where
  (o bounded o<n-1 :: []) → a (((1ₒ +ₒ o) bounded subst (_>ₒ 1ₒ +ₒ o) (b+a-b≡b n≥1)
                                                        (a+b>a+c {a = 1ₒ} o<n-1) ) :: [])
```

We use a few theorems that we did not present, but we hope that the name of the identifier suggests its meaning. Both hd and tl abstract over array length, which is bound to n of type Ordinal. Both functions require a proof that the length is at least one. In hd we simply select the element at position zero, and we have to construct a proof that zero is smaller than n (given that $n \geq 0$). The type signature of tl says that the result will be one element "shorter" than the input. We construct a proof that the given the index o that is bound by $n -_o 1_o$, the index $1_o +_o o$ is valid in the index-space of a.

While these definitions feel very much like the ones define on lists/streams, there are at least two important subtleties. First, the choice on the length relation is crucial here. If we were to specify input/output lengths in tl as $n + 1_o \to n$ we would not be able to define the tail in the usual sense. The reason is that $(\omega_o +_o 1_o) -_o 1_o = (\omega_o +_o 1_o)$, therefore in the limit case, we would have to take an element from the right. Secondly, the head and tail from above are insufficient to traverse arbitrary 1-dimensional (transfinite) arrays. Head and tails would only operate on the initial segment of length ω. In order to go "beyond" we will have to switch from the regular induction to the transfinite one, *i.e.* explain what happens at every limit ordinal.

As another example consider a concat, take and drop, also defined in infinity-agnostic style:

```
conc : ∀ {X : Set}{m n} → Ar X 1 (m :: []) → Ar X 1 (n :: []) → Ar X 1 (m +ₒ n :: [])
conc {m = m}{n}(imap a) (imap b) = imap body
  where
    body : _
    body (x bounded x<m+n :: []) with x <ₒ? m
    ... | yes p = a ((x bounded p) :: [])
    ... | no ¬p = b (((x -ₒ m) bounded (a+b<a+c⟹b<c {a = m}
                                    $ subst (_<ₒ m +ₒ n) (sym $ b+a-b≡b (o-≮⟹≥ ¬p)) (x<m+n))) :: [])

atake : ∀ {X : Set}{m n} → Ar X 1 (m +ₒ n :: []) → Ar X 1 (m :: [])
adrop : ∀ {X : Set}{m n} → Ar X 1 (m +ₒ n :: []) → Ar X 1 (n :: [])
```

And two theorems that define their correctness.

```
conc-thm-l : ∀ {X : Set}{m n}{a : Ar X 1 (m :: [])}{b : Ar X 1 (n :: [])}
           → ∀ ix → sel (adrop {m = m} (conc a b)) ix ≡ sel b ix
conc-thm-r : ∀ {X : Set}{m n}{a : Ar X 1 (m :: [])}{b : Ar X 1 (n :: [])}
           → ∀ ix → sel (atake {n = n} (conc a b)) ix ≡ sel a ix
```

Finally, the row-major flattening for transfinite arrays, as previously, is given by the following two functions:

```
off→idx : ∀ {n} → (sh : Vec Ordinal n) → Ix 1 (prod sh :: []) → Ix n sh
idx→off : ∀ {n} → (sh : Vec Ordinal n) → Ix n sh → Ix 1 (prod sh :: [])
```

Two important things to notice. First, the flattened shape is computed as a product of the *reversed* shape vector:

```
prod [] = 1ₒ
prod (x :: xs) = prod xs *ₒ x
```

This happens because the array of shape $[2, \omega]$ represents two rows each of which is of length ω resulting in $\omega 2$ flattening; whereas the shape $[\omega, 2]$ represents infinitely many two-element arrays, resulting in the flattening of length $2\omega = \omega$. That gives the following definition of the flattening: for an array of shape $[m, n]$, translate each index $[i, j]$ into $n * i + j$, which is guaranteed (with the help of the rm-thm theorem) to be less than $n * m$. Then this scheme is applied inductively over the structure of the shape vector. Unflattening successively applies modulo and division, following the structure of the prod, and using the divmod-thm and x%y<y to ensure that the indices are within bounds.

5.5 Relation to Streams

Let us now relate transfinite arrays wits streams. The following intuition helps: a 1-d array of length ω is a Stream. Arrays of length ωk can be represented with k streams. A 1-d array of length ω^2 can be represented as a stream of streams. The length $\omega^k + p$ can be thought of as two objects: a k-nested stream and whatever p represents. This means that as long as the length of our 1-d array is less than ω^ω, we can use the product of nested streams to represent 1-d transfinite arrays.

To facilitate presentation, we will redefine the ordinal structure to the list of ordterms, where both exponents and coefficient are natural numbres (and we avoid encoding that exponents decrease):

```
data OrdTerm : Set where
  ω^_·_ : ℕ → ℕ → OrdTerm
Ord = List OrdTerm
```

First, let us represent ordinal term $\omega^e \cdot n$ as n e-fold nested streams.

```
Fot : Set → OrdTerm → Set
Fot X (ω^ e · n) = Vec (nest-stream X e) n
  where
    nest-stream : _ → _ → _
    nest-stream X zero = X
    nest-stream X (suc x) = Stream (nest-stream X x) ∞
```

Recall that ordinal in CNF is the sum of terms in the decreasing sequence. We can literally translate this idea as follows:

```
OVec : Set → Ord → Set
OVec X [] = ⊤
OVec X (x :: xs) = Fot X x × OVec X xs
```

A vector of ordinal length is a sequence of vectors of nested strems where the nesting could be zero. For example consider an OVec X of length $\omega + 3$, this expands to: Vec (Stream X ∞) 1 × Vec

X 3 × ⊤. With a bit of effort we can translate ordinal-length 1-d arrays (up to ω^ω) into the OVec form and back. The nesting of such OVecs would give us the transfinite Tensor representation.

This construction provides an interface between transfinite arrays and streams. For example, an application may obtaining data as a sequence of streams, arrange them in a rectangular structure, perform array-like operations using ordinal indices, and convert results back to streams. Note that this is clearly not the only way to convert between stream and transfinite arrays. However, one of the nice properties of these data structures is that they preserve the order of elements within the streams.

One may notice that ω^ω is a relatively small ordinal, and does not cover the entire space of the presented transfinite arrays. Can we represent larger transfinite arrays with streams? Intuitively, ω^ω is an infinitely-dimensional space, and the indices into such a space are ω-sequences of natural numbers. A natural move here is to consider a function Stream ℕ → X, but Ar already gives us a function based representation. The question is whether there is an alternative.

An important observation here is that it is possible to obtain a point of ω^ω space without providing all the index components. We can give an initial prefix of the index and assume that the (infinite) postfix contains zeros at all positions. Similarly as if we fix a hyperplane in a 3-d space, we can index its points with two coordinates. This is one of the possible interpretations of the limit $\omega^\omega = \omega^0\ \omega^1\ \omega^2 \ldots$: for every point in the infinitely-dimensional space, we choose the number of dimensions k, and then we use the k-th member of the sequence to find the actual value of the point (in the k-dimensional space) we are looking for. If we translate it back to streams, we need a data structure that represents an infinite sequence, but where elements "grow" at every step. In other words, if streams representat functions $\mathbb{N} \to X$, in order to represent ordinals ω^ω and beyond, we need a representation for the function $(x : \mathbb{N}) \to X(n)$. While such a data structure can be easily defined:

```
record St (X : ℕ → Set) : Set where
  coinductive
  field
    hd : X 0
    tl : St (X ∘ suc)

ω^ω : Set → Set
ω^ω X = St (λ n → OVec X (ω^ n · 1 :: []))
```

its practical use is yet to be determined.

6 RELATED WORK

The idea to extend array indexing domain for better expressibility of a language is not new. For example in [McDonnell and Shallit 1980], the extended indices are treated as cardinal numbers with one extra point called ∞. Such a construction is sometimes called Rieman sphere. While the arithmetic rules in this system are straight-forward, the row-major flattening is lost. In J [Jsoftware 2016] infinity is added in a similar style as a value, but infinite arrays are not allowed.

In [Taylor 1982] the authors propose to extend the domain of array indices with real numbers. More specifically, a real-valued function gives rise to an array in which valid indices are those that belong to the domain of that function. The authors investigate expressibility of such arrays and they identify classes of problems where this could be useful, but neither provide a full theory nor discuss any implementation-related details.

Besides the related work that stems from APL and descendent array languages, there is an even larger body of work that has its origins in lists and streams. One of the best-known fundamental works on the theory of lists using ordered pairs can be found in [McCarthy 1960, sec. 3], where a class of S-expressions is defined. The concepts of *nil* and *cons* are introduced, as well as *car* and *cdr*, for accessing the constituents of *cons*.

The Theory of Lists [Bird 1987] defines lists abstractly as linearly ordered collections of data. The empty list and operations like length of the list, concatenation, filter, map and reduce are introduced axiomatically. Lists are assumed to be finite. The questions of representation of this data structure in memory, or strictness of evaluation, are not discussed.

Concrete Stream Calculus [Hinze 2010] introduces streams as codata. Streams are similar to McCarthy's definition of lists, in that they have functions *head* and *tail*, but they lack *nil*. This requires streams to be infinite structures only. The calculus is presented within Haskell.

Streams are also related to dataflow models, such as [Estrin and Turn 1963; Kahn 1974; Petri 1962]. The computational graphs in the latter can be seen as recursive expressions on potentially infinite streams. As demonstrated in [Beck et al. 2015], there is a demand to consider multidimensional infinite streams that cache their parts for better efficiency.

Two array representations, called *push arrays* and *pull arrays*, are described in [Svensson and Svenningsson 2014]. The framework presented in this paper can be seen as an extensions of the concept of pull arrays. While our arrays are still index-value functions, we make sure that they are rank-polymorphic, and dependent types make it possible to move a number of checks such as range check into the type signatures. Push arrays turn an array into a stateful object that can be updated at the given index. The main motivation for such a data structure is efficient code generation. There is no conceptual difficulty in introducing push arrays in Agda. However, as we are not yet concerned with code generation, passing around stateful objects that are encapsulated in monads complicates the specification and the reasoning.

All arrays in the described framework use finite representation of the shape: a Vec of ordinals/natural numbers. In the finite cases, this fact in combination with updates in place, which can be achieved by means of monads [Wadler 1995], uniqueness typing [Barendsen and Smetsers 1996] or reference counting [Grelck and Scholz 2006] make efficient code generation possible. It would be interesting to explore whether we can reuse the same approach in the infinite cases.

This work is largely based on the idea of containers [Abbott et al. 2003a, 2004, 2005, 2003b], which give a uniform way to represent "collections of things" *e.g.* lists, trees *etc.* Containers are given by the type of shapes, and the shape-indexed type family of positions. In case of our presentation of arrays, the shapes are Vectors of natural numbers, and the positions are given by the Ix type. The type of the presented arrays is a restricted form of containers.

A similar idea in the context of Haskell is described in [Gibbons 2017]. The formalism is also largely based on the notion of containers, and while providing a construction for rank-polymorphic arrays, it does not support rank-polymorphic *dependent* operations.

The work on dependent type systems for array languages include [Slepak et al. 2014; Trojahner and Grelck 2009; Xi and Pfenning 1998]. All three frameworks are practically oriented and use a *restricted* form of dependent types. For example, Qube uses SMT solver for type inference; in Dependent ML not all the types can act as type indices; and Remora restricts the type system in a such a way so that generated type constraints fall into decidable domain. While these restrictions are indeed very useful when writing programs in a language, we believe that our approach (that has none of the above restrictions) is more applicable as an exploration vehicle.

An in-depth investigation on ways to extend a type theory to include the notion of ordinals can be found in [Hancock 2000].

7 CONCLUDING REMARKS

In this paper we have formalised the notion of rank-polymorphic transfinite arrays — multi-dimensional arrays indexed with countable ordinals. We have demonstrated that, as it is prescribed by many array theories for finite arrays, transfinite arrays admit row-major flattening and reshaping. This makes it possible to use transfinite arrays as a vehicle for infinity-agnostic specifications of numerical problems, *i.e.* specifications that work equally well on finite and infinite data.

Rank polymorphism makes it possible to define specifications using index-free APL-style array combinators. In contrast to APL, the combinators are fully typed, and they are defined within the language, meaning that the resulting specifications have a predictable behaviour and opens up opportunities for cross-operator optimisations.

As we have seen in the previous section, transfinite arrays offer an interface to nested streams, and make it possible to input (or output) data as streams while making computations in the array style. We have also seen that transfinite arrays offer a convenient interface to work with very large objects such as arrays of size ω^ω, for example representing Hilbert spaces. As our ordinals are defined in Cantor Normal From, their definition is inductive and comparison operations are decidable. This allows one to take algorithmic decisions based on the finiteness of the input within the same data type.

While there is no yet efficient implementation for the proposed framework, we briefly report on our two experimental prototypes. First we have implemented a small language with transfinite arrays called Heh. This is an untyped interpreter written in Ocaml. As opposed to the imap presented in this paper, Heh implements support for partitions within the definitions of the index spaces and makes sure that infinite arrays are updated in a lazy fashion. Programs that use finite arrays can be compiled to SaC, leveraging efficient code generation for parallel architectures. Our second prototype translates finite Ar-based Agda specifications (as we presented them in this paper to SaC). The prototype is written in Agda and it uses reflection capabilities of the language to perform the translation. Both prototypes can be found in the supplementary materials.

The main difficulty with compiling transfinite arrays lies in figuring out efficient memory management strategy. On the one hand, lack of garbage collection in finite array languages make them efficient in high-performance domains (mainly due to their ability to do efficient in-place updates). The best memory management strategy in case of infinite arrays is yet to be determined. While we would like to keep in-place updates, infinite arrays cannot be strict, so we will have to find a strategy on how to update thunks and make sure that no space leaks are introduced.

The most disturbing part of ordinal-based indexing is non-commutativity of arithmetic operations. While working within a proof-assistant a large number of mistakes is caught, yet it is still possible to write unintended specifications (recall an example when $(\omega + 1) - 1 = \omega + 1$). There exists a concept of natural sum and product on ordinals: both are commutative and associative, but they loose the continuity in the right argument. Whether these can be used in the array theory is an open question.

Our reasoning would become simpler if we were to treat extensionally equal arrays as substitutable. For that we need functional extensionality which we could obtain by switching to Cubical Agda. This investigation is future work.

As we have stated in Section 4, it is unclear whether it would be possible to maintain well-behaved reshaping when using colist based implementation of infinite arrays. Our assumption is that there should be a way to diagonalise any n-dimensional so that reshaping still works. However, specific details and consequences are yet to be investigated.

Finally, it is not clear what happens with transfinite arrays if we go beyond ϵ_0.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

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

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

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theoretical Computer Science* 342, 1 (2005), 3 – 27. https://doi.org/10.1016/j.tcs.2005.06.002 Applied Semantics: Selected Topics.

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

Andreas Abel. 2012. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. *Electronic Proceedings in Theoretical Computer Science* 77 (Feb 2012), 1–11. https://doi.org/10.4204/eptcs.77.1

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6, 6 (1996), 579–612.

Jarryd P. Beck, John Plaice, and William W. Wadge. 2015. Multidimensional infinite data in the language Lucid. *Mathematical Structures in Computer Science* 25, 7 (2015), 1546–1568. https://doi.org/10.1017/S0960129513000388

Robert Bernecky. 1987. An Introduction to Function Rank. *ACM SIGAPL Quote Quad* 18, 2 (Dec. 1987), 39–43.

R. S. Bird. 1987. An Introduction to the Theory of Lists. In *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*. Springer-Verlag New York, Inc., New York, NY, USA, 5–42. http://dl.acm.org/citation.cfm?id=42675.42676

K. Ciesielski. 1997. *Set Theory for the Working Mathematician*. Cambridge University Press.

G. Estrin and R. Turn. 1963. Automatic Assignment of Computations in a Variable Structure Computer System. *IEEE Transactions on Electronic Computers* EC-12, 6 (Dec 1963), 755–773. https://doi.org/10.1109/PGEC.1963.263559

Jeremy Gibbons. 2017. APLicative Programming with Naperian Functors. In *European Symposium on Programming (LNCS)*, Hongseok Yang (Ed.), Vol. 10201. 568–583. https://doi.org/10.1007/978-3-662-54434-1_21

Clemens Grelck and Sven-Bodo Scholz. 2006. SAC - A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427. https://doi.org/10.1007/s10766-006-0018-x

Peter Hancock. 2000. *Ordinals and interactive programs*. Ph.D. Dissertation.

Ralf Hinze. 2010. Concrete Stream Calculus: An Extended Study. *J. Funct. Program.* 20, 5-6 (Nov. 2010), 463–535. https://doi.org/10.1017/S0956796810000213

Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA.

Michael A. Jenkins and Janice I. Glasgow. 1989. A logical basis for nested array data structures. *Computer Languages* 14, 1 (1989), 35 – 51. https://doi.org/10.1016/0096-0551(89)90029-5

Inc. Jsoftware. 2016. Jsoftware: High performance development platform. http://www.jsoftware.com/.

Gilles Kahn. 1974. The Semantics of Simple Language for Parallel Programming.. In *IFIP Congress*. 471–475.

Panagiotis Manolios and Daron Vroon. 2005. Ordinal Arithmetic: Algorithms and Mechanization. *Journal of Automated Reasoning* 34, 4 (2005), 387–423. https://doi.org/10.1007/s10817-005-9023-9

P. Martin-Löf. 1985. Constructive Mathematics and Computer Programming. In *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*. Prentice-Hall, Inc., USA, 167–184.

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

Eugene E. McDonnell and Jeffrey O. Shallit. 1980. Extending APL to Infinity. In *APL 80 : International Conference on APL*, Gijsbert van der Linden (Ed.). Amsterdam ; New York : North-Holland Pub. Co. : sole distributors for the USA and Canada, Elsevier North-Holland, 123–132.

Trenchard More. 1973. Axioms and Theorems for a Theory of Arrays. *IBM J. Res. Dev.* 17, 2 (March 1973), 135–175. https://doi.org/10.1147/rd.172.0135

Trenchard More. 1979. The Nested Rectangular Array as a Model of Data. In *Proceedings of the International Conference on APL: Part 1 (APL '79)*. Association for Computing Machinery, New York, NY, USA, 55–73. https://doi.org/10.1145/800136.804440

Lenore M. Restifo Mullin. 1988. *A Mathematics of Arrays.* Ph.D. Dissertation. Syracuse University.

Ulf Norell. 2009. *Dependently Typed Programming in Agda.* Springer Berlin Heidelberg, Berlin, Heidelberg, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

Carl Adam Petri. 1962. *Kommunikation mit Automaten.* Ph.D. Dissertation. Universität Hamburg.

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

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

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

Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing Push Arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '14)*. ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/2636228.2636231

R. W.W. Taylor. 1982. Indexing Infinite Arrays: Non-finite Mathematics in APL. *SIGAPL APL Quote Quad* 13, 1 (July 1982), 351–355. https://doi.org/10.1145/390006.802264

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

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 87 (July 2019), 29 pages. https://doi.org/10.1145/3341691

Philip Wadler. 1995. *Monads for functional programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 24–52. https://doi.org/10.1007/3-540-59451-5_2

Arthur Whitney. 2001. K. http://archive.vector.org.uk/art10010830. [Accessed: June 2020].

Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732