A Lambda Calculus for Transfinite Arrays

Unifying Arrays and Streams

ARTJOMS ŠINKAROVS, Heriot-Watt University
SVEN-BODO SCHOLZ, Heriot-Watt University

We propose a design for a functional language that natively supports infinite arrays. We use ordinal numbers to introduce the notion of infinity in shapes and indices. By doing so, we obtain a calculus that naturally extends existing array calculi and, at the same time, allows for recursive specifications as they are found in stream- and list-based settings. Furthermore, the main language construct that can be thought of as an n-fold *cons* operator gives rise to expressing transfinite recursion in data, something that lists or streams usually do not support. This makes it possible to treat the proposed calculus as a unifying theory of arrays, lists and streams. We give an operational semantics of the proposed language, discuss design choices that we have made, and demonstrate its expressibility with several examples. We also demonstrate that the proposed formalism preserves a number of well-known universal equalities from array/list/stream theories, and discuss implementation-related challenges.

CCS Concepts: • **Theory of computation** → **Operational semantics**;

Additional Key Words and Phrases: ordinals, arrays, semantics, functional languages

ACM Reference format:
Artjoms Šinkarovs and Sven-Bodo Scholz. 2017. A Lambda Calculus for Transfinite Arrays. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 30 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Conceptually, lists and streams are different objects. Lists are finite inductive objects that can be characterised as the smallest fixpoint: Lst $A = \mu X.1 + A \times X$, and streams are infinite co-inductive objects that are characterised as the greatest fixpoint: Str $A = \nu X.A \times X$.

Despite these conceptual differences between lists and streams, it has been proven useful to enable programmers to specify functions that can operate on both forms equally well. In particular languages that allow for the construction of cyclic structures can support a list type $[A]$ as the greatest fix point $\nu X.1{+}A{\times}X$ without requiring extra implementation effort. With this construction, any function that operates on lists inherently is applicable to streams as well.

A similar unification of streams and arrays is less straight-forward. The main obstacle to such a unification lies in the fact that array computations usually make heavy use of random access selections, while stream computations are expressed in a step-wise fashion on a temporarily available window of elements. This difference has led to two distinct programming styles: stream processing [Hinze 2010; Stephens 1997; Thies et al. 2002] and array programming [Grelck and Scholz 2006; IBM 1994; Svensson and Svenningsson 2014]. If we want to apply some array-based program to a stream, it typically requires the given program to be massively rewritten.

The key towards a unification of arrays and streams, at least on a conceptual level, becomes evident when looking at arrays as index-value mappings. We can model arrays of element type A as a family of types:

$$[A]_n = \text{Fin}(n) \rightarrow A \qquad n : \text{Nat}$$

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

where $\text{Fin}(n)$ denotes the set $\{0, \ldots n{-}1\}$.

With this in mind, we can observe the following correspondence for streams:

$$\text{Str } A \simeq A^{\omega} \simeq \text{Nat} \to A \simeq [A]_{\omega}$$

Streams are isomorphic to infinite sequences, and A^{ω} is an exponential object that can be seen as a mapping of positions in that sequence to its values. Such an object is nothing but an array of infinite length. Consequently, a unification of arrays and streams can be achieved by extension of our type family for arrays to:

$$[A]_{\alpha} = \text{Fin}(\alpha) \to A \quad \alpha : \text{Nat} + 1$$

where the right injection of the sum type contains ω and the definition of Fin is extended by $\text{Fin}(\omega) = \text{Nat}$. While this conceptually unites arrays and streams in the same way as the type $[A]$ unites lists and streams, we identify two main challenges that we address in this paper.

The first challenge arises from the fact that algebraic properties on finite structures often are lost when switching to the infinite setting. As an example consider some classical list properties: value-related properties such as $\text{map } f \circ \text{map } g = \text{map } (f \circ g)$ hold for lists and streams alike but properties that relate to the structure of lists such as $\text{drop } (\text{len } a)\ (a \mathbin{++} b) = b$ typically only hold for (finite) lists; for streams, they break. While this loss of properties might be deemed acceptable in the context of list programming, in the context of array programming such structural properties play a very important role. Sophisticated array calculi have evolved around such properties such as Mullin's ψ-calculus [Mullin and Thibault 1994; Mullin 1988], Nial [Glasgow and Jenkins 1988] and the many APL-inspired array languages [Bernecky and Berry 1993; Breed et al. 1972; Hui and Iverson 1998]. Losing the generality of such properties for the sake of including streams would constitute an unacceptable loss. We tackle this issue by extending our type families for arrays further. We introduce the notion of *Transfinite Arrays* as we expand our type indices to countable ordinals:

$$[A]_{\alpha} = \text{Fin}(\alpha) \to A \quad \alpha : \text{Ord}$$

With this extension, we can resurrect most algebraic array properties for the infinite case.

The second challenge arises from the observation that transfinite arrays imply the existence of transfinite streaming, a concept that rarely considered in stream processing. We discuss what implications this extension has on classical streaming problems such as filtering and we propose solutions on how to deal with it.

The individual contributions of this paper are as follows:

(1) We define an applied λ-calculus on finite arrays, its operational semantics and a type system for array operations. The calculus is a generic core language that implicitly supports several array calculi as well as compilation to highly efficient parallel code.
(2) We expand the λ-calculus to support infinite arrays and show that the use of ordinals as indices enables a wide range of array-algebraic laws to carry over from the finite case to the infinite case.
(3) We show that the proposed calculus also maintains many streaming properties even in the context of transfinite streaming.
(4) We show that the proposed calculus inherently supports transfinite recursion. Several examples are contrasted to traditional list-based solutions.
(5) We provide and describe a prototypical implementation[1]. It demonstrates the viability of our semantics and it shows how the strict and finite fragment of the language can be mapped

[1]The implementation is provided in the anonymous supplementary materials.

into high-performance code. We also provide a brief discussion on the opportunities and challenges involved when compiling the full language capabilities into efficient code.

We start with a description of the finite array calculus and naive extensions for infinite arrays in Section 2, before presenting the ordinal-based approach and its potential in Sections 3–5. Section 6 presents our prototypical implementation. Related work is discussed in Section 7; we conclude in Section 8.

2 EXTENDING ARRAYS TO INFINITY

We define an idealised, data-parallel array language, based on an applied λ-calculus that we call λ_α. The key aspect of λ_α is built-in support for shape- and rank-polymorphic array operations, similar to what is available in APL [Iverson 1962], J [Jsoftware 2016], or SaC [Grelck and Scholz 2006].

In the array programming community, it is well-known [Falster and Jenkins 1999; Jenkins and Mullin 1991] that basic design choices made in a language have an impact on the array algebras to which the language adheres. While we believe that our proposed approach is applicable within various array algebras, we chose one concrete setting for the context of this paper. We follow the *design decisions* of the functional array language SaC, which are compatible with many array languages, and which were taken directly from K.E. Iverson's design of APL.

DD 1 *All expressions in λ_α are arrays.* Each array has a shape which defines how components within arrays can be selected.

DD 2 *Scalar expressions, such as constants or functions, are 0-dimensional objects with empty shape.* Note that this maintains the property that all arrays consist of as many elements as the product of their shape, since the product of an empty shape is defined through the neutral element of multiplication, *i.e.* the number 1.

DD 3 *Arrays are rectangular — the index space of every array forms a hyper-rectangle.* This allows the shape of an array to be defined by a single vector containing the element count for each axis of the given array.

DD 4 *Nested arrays that cater for inhomogeneous nesting are not supported. Homogeneously nested array expressions are considered isomorphic with non-nested higher-dimensional arrays.* Inhomogeneous nesting, in principle, can be supported by adding dual constructs for enclosing and disclosing an entire array into a singleton, and vice versa. DD 2 implies that functions and function application can be used for this purpose.

DD 5 *λ_α supports infinitely many distinct empty arrays that differ only in their shapes.* In the definition of array calculi, the choice whether there is only one empty array or several has consequences on the universal equalities that hold. While a single empty array benefits value-focussed equalities, structural equalities require knowledge of array shapes, even when those arrays are empty. In this work, we assume an infinite number of empty arrays; any array with at least one shape element being 0 is empty. Empty arrays with different shape are considered distinct. For example, the empty arrays of shape $[3, 0]$ and $[0]$ are different arrays.

Further we describe the syntax and informal semantics of the language in Section 2.1 and we present types for the main array constructs in Section 2.2. Readers who feel more comfortable when explanation of the language starts with types can immediately refer to Section 2.2.

2.1 Syntax Definition and Informal Semantics of λ_α

We define the syntax of λ_α in Fig. 1. Its core is an untyped, applied λ-calculus. Besides scalar constants, variables, abstractions and applications, we introduce conditionals, a recursive let operator and some basic functions on the constants, including arithmetic operations such as `+`, `-`, `*`, `/`, a

$$
\begin{array}{lcll}
c & ::= & 0, 1, \ldots, & \text{(numbers)} \\
  & \mid & \mathit{true}, \mathit{false} & \text{(booleans)} \\
  & & & \\
e & ::= & c & \text{(constants)} \\
  & \mid & x & \text{(variables)} \\
  & \mid & \lambda x.e & \text{(abstractions)} \\
  & \mid & e\ e & \text{(applications)} \\
  & \mid & \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e & \text{(conditionals)} \\
  & \mid & \mathit{letrec}\ x = e\ \mathit{in}\ e & \text{(recursive let)} \\
  & \mid & e + e, \ldots & \text{(built-in binary)} \\
  & \mid & [e, \ldots, e] & \text{(array constructor)} \\
  & \mid & e.e & \text{(selections)} \\
  & \mid & |e| & \text{(shape operation)} \\
  & \sim & &
\end{array}
$$

$$
\begin{array}{lcll}
  & \sim & & \\
  & \mid & \mathit{reduce}\ e\ e\ e & \text{(reduction)} \\
  & \mid & \mathit{imap}\ s \begin{cases} g_1 : & e_1, \\ \quad \ldots & \\ g_n : & e_n \end{cases} & \text{(index map)} \\
  & & & \\
s & ::= & e & \text{(scalar imap)} \\
  & \mid & e|e & \text{(generic imap)} \\
g & ::= & (e \mathrel{<=} x < e) & \text{(index set)} \\
  & \mid & \_(x) & \text{(full index set)}
\end{array}
$$

Fig. 1. The syntax of λ_α

remainder operation denoted as %, and comparisons <, <=, =, *etc*. The actual support for arrays as envisioned by the aforementioned design principles is provided through five further constructs: array construction, selection, shape operation, *reduce* and *imap* combinators.

All arrays in λ_α are immutable. Arrays can be constructed by using potentially nested sequences of scalars in square brackets. For example, $[1, 2, 3, 4]$ denotes a four-element vector, while $[[1, 2], [3, 4]]$ denotes a two-by-two-element matrix. We require any such nesting to be homogeneous, for adherence to DD 4. For example, the term $[[1, 2], [3]]$ is irreducible, so does not constitute a value.

The dual of array construction is a built-in operation for element selection, denoted by a dot symbol, used as an infix binary operator between an array to select from, and a valid index into that array. A valid index is a vector containing as many elements as the array has dimensions; otherwise it is undefined.

$$
[1, 2, 3, 4].[0] = 1 \qquad [[1, 2], [3, 4]].[1, 1] = 4 \qquad [[1, 2], [3, 4]].[1] = \bot
$$

The third array-specific addition to λ_α is the primitive *shape* operation, denoted by enclosing vertical bars. It is applicable to arbitrary expressions, as demanded by DD 1, and it returns the shape of its argument as a vector, leveraging DD 3. For our running examples, we obtain: $\big|[1, 2, 3, 4]\big| = [4]$ and $\big|[[1, 2], [3, 4]]\big| = [2, 2]$. DD 5 and DD 2 imply that we have:

$$
\big|[]\big| = [0] \qquad \big|[[]]\big| = [1, 0] \qquad \big|\mathit{true}\big| = [] \qquad \big|42\big| = [] \qquad \big|\lambda x.x\big| = []
$$

λ_α includes a *reduce* combinator which in essence, it is a variant of *foldl*, extended to allow for multi-dimensional arrays instead of lists. *reduce* takes three arguments: the binary function, the neutral element and the array to reduce. For example, we have:

$$
\mathit{reduce}\ (+)\ 0\ [[1, 2], [3, 4]] = ((((0 + 1) + 2) + 3) + 4)
$$

assuming row-major traversal order. This allows for shape-polymorphic reductions such as:

```
sum ≡ λa.reduce (λx.λy.x+y) 0 a ; also works for scalars and empty arrays
```

The final, and most elaborate, language construct is the *imap* (index map) construct. It bears some similarity to the classical map operation, but instead of mapping a function over the elements of an array, it constructs an array by mapping a function over all legal indices into the index space denoted by a given shape expression[2]. Added flexibility is obtained by supporting a piecewise definition of the function to be mapped. Syntactically, the *imap*-construct starts out with the keyword `imap`, followed by a description of the result shape (rule s in Fig. 1). The shape description is followed by a curly bracket that precedes the definition of the mapping function. This function can be defined piecewise by providing a set of index-range expression pairs. We demand that the set of index ranges constitutes a partitioning of the overall index space defined through the result shape expression, *i.e.* their union covers the entire index space and the index ranges are mutually disjoint. We refer to such index ranges as *generators* (rule g in Fig. 1), and we call a pair of a generator and its subsequent expression a *partition*. Each generator defines an index set and a variable (denoted by x in rule g in Fig. 1) which serves as the formal parameter of the function to be mapped over the index set. Generators can be defined in two ways: by means of two expressions which must evaluate to vectors of the same shape, constituting the lower and upper bounds of the index set, or by using the underscore notation which is syntactic sugar for the following expansion rule:

$$(\textbf{imap}\ s\ \{\ \_(iv)\ \ \ldots) \equiv (\textbf{imap}\ s\ \{\ \underbrace{[0,\ldots,0]}_{n}\ <=\ iv\ <\ s:\ \ \ldots)$$

assuming that $|s| = [n]$. The variable name of a generator can be referred to in the expression of the corresponding partition.

The <= and < operators in the generators can be seen as element-by-element array counterparts of the corresponding scalar operators which, jointly, specify sets of constraints on the indices described by the generators. As the index-bounds are vectors, we have:

$$v_1 <= v_2 \implies |v_1|.[0] = |v_2|.[0] \wedge \forall 0 <= i < |v_1|.[0] : v_1.[i] <= v_2.[i]$$

In the rest of the paper, we use the same element-wise extensions for scalar operators, denoting the non-scalar versions with dot on top: $c = a \dot{+} b \implies c.i = a.i + b.i$. This often helps to simplify the notation[3].

As an example of an *imap*, consider an element-wise increment of an array a of shape $[n]$. While a classical *map*-based definition can be expressed as $map\ (\lambda x.x + 1)\ a$, using *imap*, the same operation can be defined as:

```
imap [n] { [0] <= iv < [n]: a.iv + 1
```

Having mapping functions from indices to values rather than values to values adds to the flexibility of the construct. Arrays can be constructed from shape expressions without requiring an array of the same shape available:

```
imap [3,3] { [0,0] <= iv < [3,3]: iv.[0]*3 + iv.[1]
```

defines a 2-dimensional array $[[0, 1, 2], [3, 4, 5], [6, 7, 8]]$. Structural manipulations can be defined conveniently as well. Consider a *reverse* function, defined as follows:

```
reverse ≡ λa.imap |a| { [0] <= iv < |a|: a.(|a|∸iv∸[1])
```

In order to express this with *map*, one needs to construct an intermediate array, where indices of a appear as values. Note also that the explicit shape of the *imap* construct makes it possible to define shape-polymorphic functions in a way similar to our definition of *reverse*. An element-wise increment for arbitrarily shaped arrays can be defined as:

```
increment ≡ λa.imap |a| { _(iv): a.iv + 1 ; also works for scalars & empty arrays
```

[2]For readers familiar with Haskell: the *imap* defined here derives the index space from a shape expression. It does not require an argument array of that shape.

[3]A formal definition of the extended operator is: $(\dot{\oplus}) \equiv \lambda a.\lambda b.imap\ |a|\ \{\_(iv) : a.iv \oplus b.iv$ where $\oplus \in \{+, -, \cdots\}$.

DD 4 allows *imap* to be used for expressing operations in terms of n-dimensional sub-structures. All that is required for this is that the expressions on the right hand side of all partitions evaluate to non-scalar values. For example, matrices can be constructed from vectors. Consider the following expression:

```
imap [n] { [0] <= iv < [n]: [1,2,3,4] ; non-scalar partitions (incorrect attempt)
```

Its shape is $[n, 4]$; however, this shape no longer can be computed without knowing the shape of at least one element. If the overall result array is empty, its shape determination is a non-trivial problem. To avoid this situation, we require the programmer to specify the result shape by means of two shape expressions separated by a vertical bar: see the rule (generic imap) in Fig. 1. We refer to these two shape expressions as the *frame shape* which specifies the overall index range of the *imap* construct as well as the *cell shape* which defines the shape of all expressions at any given index. The concatenation of those two shapes is the overall shape of the resulting array. For more discussions related to the concepts of frame and cell shapes, see [Bernecky 1987, 1993; Bernecky and Iverson 1980]. The above *imap* expression therefore needs to be written as:

```
imap [n]|[4] { [0] <= iv < [n]: [1,2,3,4] ; non-scalar partitions (correct)
```

to be a legitimate expression of λ_α. The (scalar imap) case in Fig. 1, which we use predominantly in the paper, can be seen as syntactic sugar for the generic version, with the second expression being an empty vector.

2.2 Towards a Type System for λ_α

We will present an outline of a type system here so that a reader could develop a better understanding of the essence of the array calculus that λ_α provides. For the sake of readability, we have taken some small liberties, like omitting definitions of standard arithmetic operations as well as standard non-array constructs.

We use dependent types to specify array operations. First we define the types we will use as well as well-formedness criteria for array types.

$$\textsc{Nat}\quad \frac{}{\text{Nat} : \text{Type}} \qquad \textsc{Bool}\quad \frac{}{\text{Bool} : \text{Type}} \qquad \textsc{Fin}\quad \frac{n : \text{Nat}}{\text{Fin}(n) : \text{Type}} \qquad \textsc{Fun}\quad \frac{A : \text{Type} \qquad B : \text{Type}}{A \to B : \text{Type}}$$

$$\textsc{Array}\quad \frac{T : \text{Type} \qquad T \notin \{\text{Array}\} \qquad d : \text{Nat} \qquad s : \text{Fin}(d) \to \text{Nat} \qquad v : \left(\prod i : \text{Fin}(d).\text{Fin}(s\ i)\right) \to T}{\text{Array}(T, d, s, v) : \text{Type}}$$

Nat is a type for natural numbers, Bool is a type for booleans, $\text{Fin}(n)$ is a type for numbers from 0 to $n - 1$. Function types are standard. An array type is a quadruple, where the first element is a type of the base element. We prohibit T to be of array types, as according to DD 4, nested arrays are not supported. The second element of the tuple is the dimensionality of an array. We do not support nested arrays, but we support multi-dimensional arrays, so instead of having a type $[[A]_m]_n$ we have a type $[A]_{\langle n,m\rangle}$. Such a shape vector $\langle n, m\rangle$ is a third component of the tuple and it is modeled as a function from positions into vector components, *e.g.* $\{0 \to n, 1 \to m\}$ in our example. The last component of the tuple is a function type that maps an index vector type to a value type T. For each dimension i : $\text{Fin}(d)$ the corresponding index component has to be within the given shape, *i.e.* it has to be of type $\text{Fin}(s\ i)$.

The definitions of Nat and Fin are standard:

$$\textsc{Nat}_0\ \frac{}{0 : \mathrm{Nat}} \qquad \textsc{Nat}_S\ \frac{n : \mathrm{Nat}}{S\ n : \mathrm{Nat}} \qquad \textsc{Fin-0}\ \frac{n : \mathrm{Nat}}{\overline{0} : \mathrm{Fin}(S\ n)} \qquad \textsc{Fin-S}\ \frac{n : \mathrm{Nat} \qquad k : \mathrm{Fin}(n)}{\overline{S}\ k : \mathrm{Fin}(S\ n)}$$

We use $\overline{x}$ notation to denote conversion from Nat to $\mathrm{Fin}(x+1)$:

$$x : \mathrm{Nat} \implies \overline{x} : \mathrm{Fin}(x+1)$$

We use standard context $\Gamma ::= \cdot \mid \Gamma, x : A$, where A : Type. All the numbers in the language are natural numbers and the shape operation for any array of shape s returns a one-dimensional vector of Nats, with the content s:

$$\textsc{Const}\ \frac{}{\Gamma \vdash c : \mathrm{Nat}} \qquad \textsc{Shape}\ \frac{\Gamma \vdash a : \mathrm{Array}(T, d, s, v)}{\Gamma \vdash |a| : \mathrm{Array}(\mathrm{Nat}, 1, \lambda\_.d, \lambda\phi.s\ (\phi\ \overline{0}))}$$

To construct a one-dimensional array using the bracket notation $[e_0, \ldots, e_{n-1}]$ we ensure that all the elements have the same type, the shape vector of such an array is $\langle n \rangle$ — a single-element vector containing n. The value function of such an array is $\{\langle 0 \rangle \mapsto e_0, \langle 1 \rangle \mapsto e_1, \ldots\}$ and we use a meta operator packvec to construct it.

$$\textsc{1d-Arr}\ \frac{\forall 0 \le i < n.\ \Gamma \vdash e_i : T \qquad T \notin \{\mathrm{Array}\}}{\Gamma \vdash [e_0, \ldots, e_{n-1}] : \mathrm{Array}(T, 1, \lambda\_.n, \mathrm{packvec}\ e_0\ \ldots\ e_{n-1})}$$

$$\begin{array}{l} \mathtt{packvec}\ e_0\ \ldots\ e_{n-1} = \\ \quad \lambda\phi\,.\ \mathbf{if}\ \phi\ \overline{0} = \overline{0}\ \mathbf{then}\ e_0 \\ \qquad\quad \mathbf{else\ if}\ \phi\ \overline{0} = \overline{1}\ \mathbf{then}\ e_1 \\ \qquad\quad \ldots \end{array}$$

To construct a $(d+1)$-dimensional array using n d-dimensional arrays we expect all the arrays to have the same dimensionality d and the same shape. Therefore we require d to be the same and we require s to be the same. By the latter we mean extensional equality. As s will be of a type $\mathrm{Fin}(d) \longrightarrow \mathrm{Nat}$, such a check is decidable. Finally, we use the pack meta operator to create a value function for the resulting array.

$$\textsc{nd-Arr}\ \frac{\begin{array}{c}\forall 0 \le i < n.\ \Gamma \vdash e_i : \mathrm{Array}(T, d, s, v_i) \\ s_a \equiv \lambda i.\mathrm{if}\ i = \overline{0}\ \mathrm{then}\ n\ \mathrm{else}\ s\ (i - \overline{1})\end{array}}{\Gamma \vdash [e_0, \ldots, e_{n-1}] : \mathrm{Array}(T, d+1, s_a, \mathrm{pack}\ v_0\ \ldots\ v_{n-1})}$$

$$\begin{array}{l} \mathtt{pack}\ v_0\ \ldots\ v_{n-1} = \\ \quad \lambda\phi\,.\ \mathbf{if}\ \phi\ \overline{0} = \overline{0}\ \mathbf{then} \\ \qquad\qquad v_0\ (\lambda \mathrm{i}\,.\,\phi\ (\mathrm{i}+\overline{1})) \\ \qquad\quad \mathbf{else\ if}\ \phi\ \overline{0} = \overline{1}\ \mathbf{then} \\ \qquad\qquad v_1\ (\lambda \mathrm{i}\,.\,\phi\ (\mathrm{i}+\overline{1})) \\ \qquad\qquad \ldots \end{array}$$

When selecting an element from a d-dimensional array, we have to provide an index which shall be a 1-dimensional array of Nats of d elements, where each element is bound by the shape of the array we are selecting from.

$$\textsc{Sel}\ \frac{\begin{array}{c}\Gamma \vdash a : \mathrm{Array}(T, d, s_a, v_a) \\ \Gamma \vdash i : \mathrm{Array}(\mathrm{Nat}, 1, s_i, v_i) \qquad \Gamma \vdash s_i\ \overline{0} = d \qquad \forall 0 \le j < d.\ \Gamma \vdash (v_i\ (\lambda\_.\overline{j})) < (s_a\ \overline{j})\end{array}}{\Gamma \vdash a.i : T}$$

The *imap* construct can be seen as a generalisation of the $[e_0, \ldots]$ construct, a higher-order function that takes the shape of an array and a set of functions that generate elements for a given range of indices. We demonstrate the typing rule for the scalar *imap*, and we avoid the construction of the value function of the resulting array, as such a construction is reflected in our semantics.

$$\textsc{imap-scal}\quad \frac{\begin{array}{c}\Gamma \vdash s : \mathrm{Array}(\mathrm{Nat}, 1, s_s, v_s) \\ \forall 1 \leq i \leq n.\ \Gamma \vdash l_i : \mathrm{Array}(\mathrm{Nat}, 1, s_s, \_) \\ \forall 1 \leq i \leq n.\ \Gamma \vdash u_i : \mathrm{Array}(\mathrm{Nat}, 1, s_s, \_) \\ \forall 1 \leq i \leq n.\ \Gamma, iv_i : \mathrm{Array}(\mathrm{Nat}, 1, s_s, \_) \vdash e_i : T \qquad T \notin \{\mathrm{Array}\}\end{array}}{\Gamma \vdash \mathit{imap}\ s \begin{cases} l_1 \leq iv_1 < u_1 : & e_1, \\ \dots \\ l_n \leq iv_n < u_n : & e_n, \end{cases} : \mathrm{Array}(T, (s_s\ \overline{0}), \lambda i.v_s\ (\lambda\_.i), \_)}$$

The rest of the typing rules for applications, abstractions, letrec and conditionals are standard, therefore we omit them here.

The type system presented here imposes a distinction between natural numbers and arrays of natural numbers of an empty shape. While this helps keeping the presentation reasonably compact this distinction is undesirable for λ_α from a pragmatical perspective. As most array calculi do, we want to consider scalars to be 0-dimensional arrays with empty shape. Amongst other benefits, this allows the function $\lambda a.\mathit{imap}\ |a|\ \{\_(iv) : a.iv + 1$ to be applied to regular arrays and scalars alike.

In the above type system, we can create an array of an empty shape: $\mathrm{Array}(\mathrm{Nat}, 0, \mathrm{efq}, \lambda\phi.5)$, where $\mathrm{efq} : \mathrm{Fin}(0) \longrightarrow \mathrm{Nat}$ (a function from empty type to Nat). The object of such a type will be isomorphic to $5 : \mathrm{Nat}$, but not identical. This means that we will have to introduce explicit coercions not only between numbers of type Fin and Nat, but also between any non-array type T and an empty array of type T.

For the price of further type constructions, some of these equalities can be regained as shown in [Elsman and Dybdal 2014; Slepak et al. 2014; Trojahner and Grelck 2009]. Since this paper is mainly concerned with the calculus itself and its properties, we omit such elaboration. Instead, we assume that $T \equiv \mathrm{Array}(T, 0, \_, \_)$, for non-array types T, and numbers of type Nat and Fin can be used interchangeably.

2.3 Formal Semantics of λ_α

In this section, we offer a brief overview of the semantics. A complete semantics can be found in [Anonymous-1 2018].

In λ_α, evaluated arrays are pairs of shape and element tuples. A shape tuple consists of numbers, and an element tuple consists of numbers, booleans or functions closures. We denote pairs and tuples, as well as element selection and concatenation on them, using the following notation:

$$\vec{a} = \langle a_1, \dots, a_n \rangle \implies \vec{a}_i = a_i \qquad \langle a_1, \dots, a_n \rangle +\!\!+ \langle b_1, \dots, b_m \rangle = \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle$$

To denote the product of a tuple of numbers, we use the following notation:

$$\vec{s} = \langle s_1, \dots, s_n \rangle \implies \otimes\vec{s} = s_n \cdot \dots \cdot s_1 \cdot 1$$

When a tuple is empty, its product is one. An array is rectangular, so its shape vector specifies the extent of each axis. The number of elements of each array is finite. Element vectors contain all the elements in a linearised form. While the reader can assume row-major order, formally, it suffices that a fixed linearisation function $F_{\vec{s}}$ exists which, given a shape vector $\vec{s} = \langle s_1, \dots, s_n \rangle$, is a bijection between indices $\{\langle 0, \dots, 0 \rangle, \dots, \langle s_1 - 1, \dots, s_n - 1 \rangle\}$ and offsets of the element vector: $\{1, \dots, \otimes\vec{s}\}$. Consider, as an example, the array $[[1, 2], [3, 4]]$, with F being row-major order. This array is evaluated into the shape-tuple element-tuple pair $\langle\langle 2, 2 \rangle, \langle 1, 2, 3, 4 \rangle\rangle$. Scalar constants are arrays with empty shapes. We have 5 evaluating to $\langle\langle\rangle, \langle 5 \rangle\rangle$. The same holds for booleans and function closures: *true* evaluates to $\langle\langle\rangle, \langle \mathit{true} \rangle\rangle$ and $\lambda x.e$ evaluates to $\langle\langle\rangle, \langle [\![\lambda x.e, \rho]\!] \rangle\rangle$.

F is an invariant to the presented semantics. In finite cases, the usual choices of F are row-major order or column-major order. In infinite cases, this might be not the best option, and one could consider space-filling curves instead. F is only relevant for two operations; the creation of array values and the selection of elements from it. Selections relate the indices of the index vectors to the axes of the arrays following the order of nesting and starting with the index 0 on each level. We have: $[[1,2],[3,4]]\ [1,0] = 3$, Assuming F is row-major, $F_{\langle 2,2\rangle}(\langle 1,0\rangle)$ equals 2 which, when used as index into $\langle\langle 2,2\rangle, \langle 1,2,3,4\rangle\rangle$ returns the intended result 3.

The inverse of F is denoted as $F^{-1}_{\vec{s}}$ and for every legal offset $\{1,\ldots,\otimes\vec{s}\}$ it returns an index vector for that offset.

Deduction rules. To define the operational semantics of λ_α, we use a *natural semantics*, similar to the one described in [Kahn 1987]. To make sharing more visible, instead of a single environment ρ that maps names to values, we introduce a concept of storage; environments map names to pointers and storage maps pointers to values. Environments are denoted by ρ and are ordered lists of name-pointer pairs. Storage is denoted by S and consists of an ordered list of pointer-value pairs.

Formally, we construct storage and environments as lists of pointer-value and variable-pointer bindings, respectively, using comma to denote extensions:

$$S ::= \emptyset \mid S, p \mapsto v \qquad \rho ::= \emptyset \mid \rho, x \mapsto p$$

A look-up of a storage or an environment is performed *right to left* and is denoted as $S(p)$ and $\rho(x)$, respectively. Extensions are denoted with comma. Semantic judgements can take two forms:

$$S; \rho \vdash e \Downarrow S';\ p \qquad S; \rho \vdash e \Downarrow S';\ p \Rightarrow v$$

where S and ρ are initial storage and environment and e is a λ_α expression to be evaluated. The result of this evaluation ends up in the storage S' and the pointer p points to it. The latter form of a judgement is a shortcut for: $S; \rho \vdash e \Downarrow S';\ p \wedge S'(p) = v$.

Values. The values in this semantics are constants (including arrays) and λ-closures which contain the λ term and the environment where this term shall be evaluated:

$$\langle\langle\ldots\rangle, \langle\ldots\rangle\rangle \qquad \langle\langle\rangle, \langle[\![\lambda x.e, \rho]\!]\rangle\rangle$$

Meta-operators. Further in this section we use the following meta-operators:

$\mathbf{E}(v)$ Lift the internal representation of a vector or a number into a valid λ_α expression. For example: $\mathbf{E}(5) = 5$, $\mathbf{E}(\langle 1,2,3\rangle) = [1,2,3]$, *etc.*

$\langle\vec{s}, \_\rangle$ We use underscore to omit the part of a data structure, when binding names. For example: $S;\ p \Rightarrow \langle\vec{s}, \_\rangle$ refers to binding the variable $\vec{s}$ to the shape of $S(p)$ which must be a constant.

2.4 Core Rules

In λ_α, the rules for the λ-calculus core, *i.e.* constants, variables, abstractions and applications are straightforward adaptations of the standard rules for strict functional languages to our notation with storage and pointers:

$$\textsc{Const-Scal}\ \frac{c \textit{ is scalar}}{S;\rho \vdash c \Downarrow S_1, p \mapsto \langle\langle\rangle, \langle c\rangle\rangle;\ p} \qquad \textsc{Var}\ \frac{x \in \rho \qquad \rho(x) \in S}{S;\rho \vdash x \Downarrow S;\ \rho(x)}$$

$$\textsc{Abs}\ \frac{}{S;\rho \vdash \lambda x.e \Downarrow S, p \mapsto \langle\langle\rangle, \langle[\![\lambda x.e, \rho]\!]\rangle\rangle;\ p} \qquad \textsc{App}\ \frac{\begin{array}{c} S;\rho \vdash e_1 \Downarrow S_1;\ p_1 \Rightarrow \langle\langle\rangle, [\![\lambda x.e, \rho_1]\!]\rangle \\ S_1;\rho \vdash e_2 \Downarrow S_2;\ p_2 \qquad S_2;\rho_1, x \mapsto p_2 \vdash e \Downarrow S_3;\ p_3 \end{array}}{S;\rho \vdash e_1\ e_2 \Downarrow S_3;\ p_3}$$

As an illustration, consider the evaluation of $(\lambda x.x)\ 42$:

$$\begin{array}{lrl}
\emptyset;\ \emptyset & (\lambda x.x)\ 42 & \textsc{Abs} \\
S_1 = p_1 \mapsto \langle\langle\rangle, [\![\lambda x.x, \emptyset]\!]\rangle;\ \emptyset & p_1\ 42 & \textsc{Const-Scal} \\
S_2 = S_1, p_2 \mapsto \langle\langle\rangle, \langle 42\rangle\rangle;\ \emptyset & p_1\ p_2 & \textsc{App} \\
S_2;\ x \mapsto p_2 & x & \textsc{Var} \\
S_2;\ \emptyset & p_2 & \square
\end{array}$$

We start with an empty storage and an empty environment. The outer application demands that the App-rule be used. It enforces three computations: the evaluation of the function, the evaluation of the argument and the evaluation of the function body with an appropriately expanded environment. The function is evaluated by the Abs-rule which adds a closure $p_1 \mapsto \langle\langle\rangle, [\![\lambda x.x, \emptyset]\!]\rangle$ to the storage and returns the pointer p_1 to it. The argument is evaluated by the Const-Scal-rule which adds $p_2 \mapsto \langle\langle\rangle, \langle 42\rangle\rangle$ to the storage and returns p_2. Finally, the App-rule demands the evaluation of the body of the function with an environment $\rho_1 = x \mapsto p_2$. The body being just the variable x, the Var-rule gives us $S_2;\ p_2$ as final result.

The rules for array constructors and array selections are rather straightforward as well. Both these constructs are strict:

$$\textsc{Imm-Array}\ \frac{\begin{array}{c} n \geq 1 \qquad \overset{n}{\underset{i=1}{\forall}} S_i;\rho \vdash c_i \Downarrow S_{i+1};\ p_i \\ P = \langle p_1, \dots, p_n\rangle \qquad \text{AllSameShape}(S_{n+1}, P) \qquad S' = S_{n+1}, p_o \mapsto \langle\langle 1\rangle, \langle n\rangle\rangle, p_i \mapsto S_{n+1}(p_1) \\ S', \rho \vdash \mathit{imap}_1\ p_o|p_i\ \{\langle i{-}1\rangle \mapsto p_i \mid i \in \{1, \dots, n\}\} \Downarrow S'';\ p \end{array}}{S_1;\rho \vdash [c_1, \dots, c_n] \Downarrow S'';\ p}$$

$$\textsc{Imm-Array-empty}\ \frac{}{S;\rho \vdash [\,] \Downarrow S, p \mapsto \langle\langle 0\rangle, \langle\rangle\rangle;\ p}$$

$$\textsc{Sel-strict}\ \frac{S;\rho \vdash i \Downarrow S_1;\ p_i \Rightarrow \langle\langle d\rangle, \vec{\imath}\rangle \qquad S_1;\rho \vdash a \Downarrow S_2;\ p_a \Rightarrow \langle\vec{s}, \vec{a}\rangle \qquad k = F_{\vec{s}}(\vec{\imath})}{S;\rho \vdash a.i \Downarrow S_3, p \mapsto \langle\langle\rangle, \langle\vec{a}_k\rangle\rangle;\ p}$$

Empty arrays are put into the storage with shape [0] (Imm-Array-empty-rule). Non-empty arrays (Imm-Array-rule) evaluate all the components and ensure that they are all of the same finite shape. Subsequently, we assemble evaluated components into the resulting array value ensuring that the flattening adheres to F. This is achieved by using an auxiliary term imap_1. It takes the form $\mathit{imap}_1\ p_o|p_i\ \{\vec{\imath}^1 \mapsto p_{\vec{\imath}^1}, \dots, \vec{\imath}^n \mapsto p_{\vec{\imath}^n}\}$ where p_o and p_i are pointers to frame and cell shapes, and the set $\{\vec{\imath}^1 \mapsto p_{\vec{\imath}^1}, \dots, \vec{\imath}^n \mapsto p_{\vec{\imath}^n}\}$ contains pairs of frame-shape indices and value pointers for all

$$\frac{\begin{array}{c}\textsc{IMap-Strict}\\ S;\rho\vdash e_{\text{out}}\Downarrow S_1;\ p_{\text{out}}\Rightarrow\langle\langle d_o\rangle,\vec{s_{\text{out}}}\rangle \qquad S_1;\rho\vdash e_{\text{in}}\Downarrow S_2;\ p_{\text{in}}\Rightarrow\langle\langle d_i\rangle,\vec{s_{\text{in}}}\rangle\\ \hat{S}_1=S_2 \qquad \overset{n}{\underset{i=1}{\forall}}\ \hat{S}_i;\rho\vdash g_i\Downarrow\hat{S}_{i+1};\ p_{g_i}\Rightarrow\bar{g}_i \qquad \text{FormsPartition}(\vec{s_{\text{out}}},\{\bar{g}_1,\ldots,\bar{g}_n\})\\ \bar{S}_1=\hat{S}_{n+1} \qquad \forall(i,\vec{\imath})\in\text{Enumerate}(\vec{s_{\text{out}}})\exists k:\left|\begin{array}{l}\vec{\imath}\in\bar{g}_k\ \wedge\ \bar{g}_k=Gen(x_k,\_,\_)\\ \bar{S}_i,p\mapsto\langle\langle d_o\rangle,\vec{\imath}\rangle;\rho,x_k\mapsto p\vdash e_k\Downarrow\bar{S}'_i;\ p_{\vec{\imath}}\\ \bar{S}'_i;\rho,x\mapsto p_{\vec{\imath}}\vdash|x|\Downarrow\bar{S}_{i+1};\ p'_{\vec{\imath}}\Rightarrow\langle\langle d_i\rangle,\vec{s_{\text{in}}}\rangle\end{array}\right.\\ \bar{S}_{\otimes\vec{s_{\text{out}}}+1},\rho\vdash imap_1\ p_{\text{out}}|p_{\text{in}}\ \{\vec{\imath}\mapsto p_{\vec{\imath}}\mid(\_,\vec{\imath})\in\text{Enumerate}(\vec{s_{\text{out}}})\}\Downarrow S';\ p\end{array}}{S;\rho\vdash imap\ e_{\text{out}}|e_{\text{in}}\begin{cases}g_1: & e_1,\\ \ldots & \\ g_n: & e_n\end{cases}\Downarrow S';\ p}$$

$$\frac{\begin{array}{c}\textsc{Gen}\\ S;\rho\vdash e_1\Downarrow S_1;\ p_1\Rightarrow\langle\langle n\rangle,\vec{l}\rangle \qquad S_1;\rho\vdash e_2\Downarrow S_2;\ p_2\Rightarrow\langle\langle n\rangle,\vec{u}\rangle\end{array}}{S;\rho\vdash(e_1\leq x<e_2)\Downarrow S,p\mapsto Gen(x,\vec{l},\vec{u});\ p}$$

legal indices into the frame shape. The formal definition of the deduction rule for $imap_1$ is provided in [Anonymous-1 2018, Sec 2.1.1].

The rule for selection (Sel-strict-rule) first evaluates the array we are selecting from, and the index vector specifying the array index we wish to select. Then, we compute the offset into the data vector by applying F to the index vector. Finally, we get the scalar value at the corresponding index. When applying F, we implicitly check that:

- the index is within bounds $1 \leq k \leq \otimes\vec{s}$, as $F_{\vec{s}}$ is undefined outside the index space bounded by $\vec{s}$; and
- the index vector and the shape vector are of the same length, which means that selections evaluate scalars and not array sub-regions.

IMap. In order to keep the *imap* rule reasonably concise, we introduce two separate rules, a rule Gen for evaluating the generator bounds, and the main rule for *imap*, the Imap-Strict-Rule. The Gen-rule introduces auxiliary values $Gen(x, \vec{l}, \vec{u})$ which are triplets that keep a variable name, lower bound and upper bound of a generator together. These auxiliary values are references only by the rule for *imap*.

Evaluation of an *imap* happens in three steps. First, we compute shapes and generators, making sure that generators form a partition of $\vec{s_{\text{out}}}$ (FormsPartition is responsible for this). Secondly, for every valid index defined by the frame shape (Enumerate generates a set of offset-index-vector pairs), we find a generator that includes the given index (denoted $\vec{\imath} \in \bar{g}_k$). We evaluate the generator expression e_k, binding the generator variable x_k to the corresponding index value and check that the result has the same shape as p_{in}. Finally, we combine evaluated expressions for every index of the frame shape into $imap_1$ for further extraction of scalar values.

All missing rules, including built-in operations, conditionals and recursion through the *letrec*-construct are straightforward adaptations of the standard rules. They can be found in [Anonymous-1 2018]. Formal definitions of helper functions, such as AllSameShape, will also be found there.

2.5 Infinite Arrays

In order to support infinite arrays, we introduce the notion of infinity in λ_α, and we allow infinities to appear in shape components. Syntactically, this can be achieved by adding a symbol for infinity, as shown in Fig. 2. For disambiguation, we refer to the extended version of λ_α as λ_α^∞. Adding ∞ has

λ_α with cardinal infinity. extends λ_α

$$
\begin{array}{rcll}
c & ::= & \cdots & \\
  & | & \infty & \text{(infinity constant)}
\end{array}
$$

Fig. 2. The syntax of λ_α^∞

several implications. First of all, our built-in arithmetic needs to be extended. We treat infinity in the usual way, applying the model commonly known as a Riemann sphere. That is:

$$
z + \infty = \infty \qquad z \times \infty = \infty \qquad \frac{z}{\infty} = 0 \qquad \frac{z}{0} = \infty
$$

The following operations are undefined:

$$
\infty + \infty \qquad \infty - \infty \qquad \infty \times 0 \qquad \frac{0}{0} \qquad \frac{\infty}{\infty}
$$

While these additions to the semantics are trivial, allowing infinity to appear in shapes has a more profound impact on our semantics. Our rule for *imap*-constructs (IMap-Strict) forces the evaluation of all elements. If our result shape contains infinity, this can no longer be done. As we want to maintain a strict evaluation regime for function applications in general, we turn our *imap*-construct into a lazy data-structure which does not immediately compute its elements, but only does so when individual elements are being inspected. For this purpose, we extend our set of allowed values of our semantics with an *imap*-closure:

$$
\left\llbracket \mathit{imap}\ p_{\text{out}} | p_{\text{in}} \begin{cases} \bar{g}_1 : & e_1, \\ \dots & , \rho \\ \bar{g}_n : & e_n \end{cases} \right\rrbracket
$$

The *imap* closure contains pointers to frame and element shapes (p_{out} and p_{in} correspondingly), the list of partitions, where generators have been evaluated and the environment in which the *imap* shall be evaluated. The overall idea is to update, in place, this closure whenever individual elements are computed. With this extension, we can now replace our strict *imap*-rule by a lazy variant:

$$
\text{IMap-Lazy}\quad
\frac{
\begin{array}{c}
S; \rho \vdash e_{\text{out}} \Downarrow S_1;\ p_{\text{out}} \Rightarrow \langle\langle \_ \rangle, \vec{s_{\text{out}}}\rangle \qquad S_1; \rho \vdash e_{\text{in}} \Downarrow S_2;\ p_{\text{in}} \Rightarrow \langle\langle \_ \rangle, \_ \rangle \\
\hat{S}_1 = S_2 \qquad \overset{n}{\underset{i=1}{\forall}}\ \hat{S}_i; \rho \vdash g_i \Downarrow \hat{S}_{i+1};\ p_{g_i} \Rightarrow \bar{g}_i \qquad \text{FormsPartition}(\vec{s_{\text{out}}}, \{\bar{g}_1, \dots, \bar{g}_n\},)
\end{array}
}{
S; \rho \vdash \mathit{imap}\ e_{\text{out}} | e_{\text{in}} \begin{cases} g_1 : & e_1, \\ \dots & \\ g_n : & e_n \end{cases} \Downarrow \hat{S}_{n+1}, p \mapsto \left\llbracket \mathit{imap}\ p_{\text{out}} | p_{\text{in}} \begin{cases} \bar{g}_1 : & e_1, \\ \dots & ; \rho \\ \bar{g}_n : & e_n \end{cases} \right\rrbracket ;\ p
}
$$

We can see that the new rule for *imap*-constructs, in essence, performs a subset of what the strict rule from the previous section does. It still forces the result shapes, it still computes the boundaries of the generators, and it checks the validity of the overall generator set. Once these computations have been done, further element computation is delayed and an imap-closure is created instead.

The actual computation of elements is triggered upon element selection. Consequently, we need a second selection rule which can deal with *imap* closures in the array argument position:

Sel-lazy-imap

$$\frac{\begin{array}{c} S;\rho \vdash i \Downarrow S_1;\ p_i \Rightarrow \langle\langle \_ \rangle, \vec{v}\,\rangle \qquad S_1;\rho \vdash a \Downarrow S_2;\ p_a \Rightarrow \left[\!\!\left[imap\ p_{\text{out}}|p_{\text{in}} \begin{cases} \bar{g}_1 & e_1 \\ \dots & \\ \bar{g}_n & e_n \end{cases}\!\!, \rho' \right]\!\!\right] \\ S_2(p_{\text{out}}) = \langle\langle m\rangle, \_\rangle \qquad (\vec{\imath}, \vec{\jmath}) = \text{Split}(m, \vec{v}\,) \\ \exists k : \vec{\imath} \in \bar{g}_k \qquad \bar{g}_k = Gen(x_k, \_, \_) \qquad S_2, p \mapsto \mathbf{E}(\vec{\imath}); \rho', x_k \mapsto p \vdash e_k \Downarrow S_3;\ p_{\vec{\imath}} \\ S_3; \rho', x \mapsto p_{\vec{\imath}} \vdash x.\mathbf{E}(\vec{\jmath}) \Downarrow S_4;\ p \qquad S_5 = \text{UpdateIMap}(S_4, p_a, \vec{\imath}, p_{\vec{\imath}}) \end{array}}{S;\rho \vdash a.i \Downarrow S_5;\ p}$$

Selections into *imap*-closures happen at indices that are of the same length as the concatenation of the *imap* frame and cell shapes. This means that the index the *imap*-closure is being selected from has to be split into frame and cell sub-indices: $\vec{\imath}$ and $\vec{\jmath}$ correspondingly. Given that $\bar{g}_k$ contains $\vec{\imath}$, we evaluate e_k with x_k being bound to $\vec{\imath}$. As this value may be non-scalar, we evaluate a selection into it at $\vec{\jmath}$. Finally, the evaluated generator expression is saved within the *imap* closure. This step is performed by the helper function UpdateIMap, which splits the k-th partition into a single-element partition containing $\vec{\imath}$ with the computed value $p_{\vec{\imath}}$, and further partitions covering the remaining indices of $\bar{g}_k$ with the expression e_k. For more details see [Anonymous-1 2018, Sec. 2.1.1].

With this, we can define and use infinite arrays in an overall strict setting. Let us consider the definitions of the infinite array of natural numbers in λ_α^∞ on the left and Haskell-like definition on the right:

```
nats ≡ imap [∞] { _(iv): iv.[0]            nats = 0: map (+1) nats
```

Both versions define an object that delivers the value n when being selected at any index n. Both definitions provide a data structure whose computation unfolds in a lazy fashion. The main difference is that the Haskell-like specification introduces dependencies between the elements of the list. Arguably, for a large number of practical implementations, whenever an element n is selected, the entire spine of the list, up to the n-th element, has to be in place. In the λ_α^∞ case, the specification explicitly states how to compute the element at any position: the undersore in the *imap* is similar to the λ-binder. Therefore, we encode less dependencies, which means that space-efficient implementation of *imap* closures can be derived with less analysis. For example, we can envision representing *imap* closures as a hashmap.

The above comparison demonstrates important difference between a data-parallel programming style and a list-based, inherently recursive programming style. This observation leads us to the question whether similar recursive definitions are possible in λ_α^∞ at all?

2.6 Recursive Definitions

It turns out that the lazy *imap*, together with the *letrec* construct, allows for recursive definitions of arrays. A recursive definition of the natural numbers, including 0, can be defined in λ_α^∞ by:

```
letrec nats = imap [∞] { [0] <= iv < [1]: 0,
                         [1] <= iv < [∞]: nats.(iv∸[1]) + 1 in nats
```

The interesting question here is whether the semantics defined thus far ensures that all elements of the array nats are actually being inserted into one and the same *imap*-closure. For this to happen, we need the environment of the *imap*-closure to map nats to itself, and we need the selection within the body of the imap to modify the closure from which it is selecting. While the latter is given through the Sel-Lazy-Imap-rule, the former is achieved through the rule for letrec-constructs. For λ_α, we have:

Letrec

$$\frac{\rho_1 = \rho, x \mapsto p \qquad S_1 = S, p \mapsto \bot \quad S_1; \rho_1 \vdash e_1 \Downarrow S_2;\ p_2 \qquad S_3 = S_2[p_2/p] \qquad S_3; \rho, x \mapsto p_2 \vdash e_2 \Downarrow S_4;\ p_r}{S; \rho \vdash \mathit{letrec}\ x = e_1\ \mathit{in}\ e_2 \Downarrow S_4;\ p_r}$$

where $S[p_2/p]$ denotes substitution of the $x \mapsto p$ bindings inside of the enclosed environments with $x \mapsto p_2$, where x is any legal variable name. This substitution is key for creating the circular reference in the *imap*-closure from the example above.

In conclusion, the above recursive specification denotes an array with the same elements as the data-parallel specification from the previous section. In contrast to data-parallel version, this specification behaves much more like the recursive, Haskell-like version; the computation of individual elements can no longer happen directly. Since there is an encoded dependency between an element and its predecessor, the first access to an element at index n, in this variant, will trigger the computation of all elements from 0 up to n. The implementation of the UpdateIMap operation on *imap*-closures determines how these numbers are stored in memory and, consequently, how efficiently they can be accessed.

The availability of direct indexes makes it possible to encode an arbitrary order for the recursion. Consider the following example:

```
letrec a = imap [10] { [9] <= iv < [10]: 9,
                       [0] <= iv < [9]:  a.(iv∔[1])-1 in a
```

Selection of the 9th element can be evaluated in one step. In case of lists, the selection request always starts at the beginning of the list. Hence, to obtain the same performance, some optimisation of the list case is required.

2.7 List Primitives in the Array Setting

We have enabled two features that are inherent with lists, but that are usually not supported in an array setting: recursively defined data-structures and infinite arrays. All that is required to achieve this is a recursion-aware, lazy semantics of the *imap*-construct and the inclusion of an explicit notion of infinity. With these extensions, the key primitives for lists, *head*, *tail*, and *cons* can be defined as

```
head ≡ λa.a.[0]
tail ≡ λa.imap |a|∸[1] { _(iv): a.([1]∔iv)
cons ≡ λa.λb.imap [1]∔|b| { [0] <= iv < [1]: a,
                            [1] <= iv < [1]∔|b|: b.(iv∸[1])
```

More complex list-like functions can be defined on top of these. An example is concatenation:

```
letrec (++) = λa.λb.if |a|.[0] = 0 then b
                    else cons (head a) ((tail a) ++ b) in (++)
```

In case a is infinite, however, the above definition of concatenation is unsatisfying. The strict nature of λ_α will force *tail a* forever as $|a|.[0] = 0$ never yields *true*. The way to avoid this is to shift the case distinction into the lazy *imap* construct:

```
(++) ≡ λa.λb.imap |a|∔|b| { [0] <= iv < |a|: a.iv,
                            |a| <= iv < |a|∔|b|: b.(iv∸|a|)
```

As we have seen earlier, λ_α enables the typical constructions of recursive definitions of infinite vectors well-known from the realm of lists such as list of ones, natural numbers or fibonacci sequence.

Having a unified interface for arrays and lists enables programmers to switch the algorithmic definitions of individual arrays from recursive to data-parallel styles without modifying any of the code that operates on them.

However, such a unification comes at a price: we have to support a lazy version of the *imap*-construct. As a consequence, we conceptually lose the advantage of $O(1)$ access. Despite λ_α offering many opportunities for compiler optimisations like pre-allocating arrays and potentially enforcing strictness on finite, non-recursive *imap*s, one may wonder at this point how much λ_α differs from a lazy array interface in a lazy, list-based language such as Haskell?

3 TRANSFINITE ARRAYS

We now investigate to what extent λ_α^∞ adheres to the key properties of array programming — array algebras and array equalities.

3.1 Algebraic Properties

Array-based operations offer a number of beneficial algebraic properties. Typically, these properties manifest themselves as universally valid equalities which, once established, improve our thinking about algorithms and their implementations, and give rise to high-level program transformations. We define equality between two non-scalar arrays a and b as

$$a == b \iff |a| = |b| \land \forall\, iv < |a| : a.iv = b.iv$$

that is, we demand equality of the shapes and equality of all elements. The demand for equality of shapes recursively implies equality in dimensionality and the extensional character of this definition through the use of array selections ensures that we can reason about equality on infinite arrays as well.

Arrays give rise to many algebras such as Theory of Arrays [More 1973], Mathematics of Arrays [Mullin 1988], and Array Algebras [Glasgow and Jenkins 1988]. Most of the developed algebras differ only slightly, and the set of equalities that are ultimately valid depends on some fundamental choices, such as the ones we made in the beginning of the previous section. At the core of these equalities is the ability to change the shape of arrays in a systematic way without losing any of their data.

An equality from [Falster and Jenkins 1999] that plays a key role in consistent shape manipulations is:

$$\mathit{reshape}\ |a|\ (\mathit{flatten}\ a)\ \ ==\ \ a \tag{1}$$

where *flatten* maps an array recursively into a vector by *concatenating* its sub-arrays in a row-major fashion and *reshape* performs the dual operation of bringing a row-major linearisation back into multi-dimensional form. These operations can be defined in λ_α^∞ as

```
flatten ≡ λa.imap [count a] { _(iv): a.(o2i iv.[0] |a|)
reshape ≡ λshp.λa.imap shp { _(iv): (flatten a).[i2o iv shp]
```

where *count* returns the product of all shape components and *o2i* and *i2o* translate offsets into indices and vice versa, respectively. These operations effectively implement conversions from mixed-radix systems into natural numbers using multiplications and additions and back using division and remainder operations.

The above equality states that any array a can be brought into flattened form and, subsequently be brought back to its original shape. For arrays of finite shape s, this follows directly from the fact that $\mathit{o2i}\ (\mathit{i2o}\ iv\ s)\ s = iv$ for all legitimate index vectors iv into the shape s.

If we want Eq. 1 to hold for all arrays in λ_α^∞, we need to show that the above equality also holds for arrays with infinite axes. Consider an array of shape $s = [2, \infty]$. For any legal index vector $[1, n]$

into the shape s, we obtain:

$$\begin{aligned} o2i\ (i2o\ [1,n]\ [2,\infty])\ [2,\infty]) &= o2i\ (\infty \cdot 1 + n)\ [2,\infty] \\ &= o2i\ \infty\ [2,\infty] \\ &= [\infty\ /\ \infty,\ \infty\ \%\ \infty] \end{aligned}$$

which is not defined. We can also observe that all indices $[1,n]$ are effectively mapped into the same offset: ∞ which is not a legitimate index into any array in λ_α^∞. This reflects the intuition that the concatenation of two infinite vectors effectively looses access to the second vector.

The inability to concatenate infinite arrays also makes the following equality fail:

$$drop\ |a|\ (a \mathbin{++} b) == b \tag{2}$$

where a and b are vectors and $drop\ s\ x$ removes first s elements from the left. The reason is exactly the same: given that $|a| = [\infty]$ and b is of finite shape $[n]$, the shape of the concatenation is $[\infty + n] = [\infty]$, and drop of $|a|$ results in an empty vector.

Clearly, λ_α^∞ as presented so far is not strong enough to maintain universal equalities such as Eq. 1 or 2. Instead, we have to find a way that enables us to represent sequences of infinite sequences that can be distinguished from each other.

3.2 Ordinals

When numbers are treated in terms of cardinality, they describe the number of elements in a set. Addition of two cardinal numbers a and b is defined as a size of a union of sets of a and b elements. This notion also makes it possible to operate with infinite numbers, where the number of elements in an infinite set is defined via bijections. As a result, differently constructed infinite sets may end up having the same number of elements. For example, if there exists a bijection from $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$, the cardinality of both sets is the same.

When studying arrays, treating their shapes and indices using cardinal numbers is an over-simplification, because arrays have richer structure. Arrays are collections of ordered elements, where the order is established by the indices. Ordinal numbers, as introduced by G. Cantor in 1883, serve exactly this purpose — to “label” positions of objects within an ordered collection. When collections are finite, cardinals and ordinals can be used interchangeably, as we can always count the labels. Infinite collections are quite different in that regard: despite being of the same size, there can be many non-isomorphic well-orderings of an infinite collection. For example, consider two infinite arrays of shapes $[\infty, 2]$ and $[2, \infty]$. Both of these have infinitely many elements, but they differ in their structure. From a row major perspective, the former is an infinite sequence of pairs, whereas the latter are two infinite sequences of scalars. Ordinals give a formal way of describing such different well-orderings.

First let us try to develop an intuition for the concept of ordinal numbers and then we give a formal definition. Consider an ordered sequence of natural numbers: $0 < 1 < 2 < \cdots$. These are the first ordinals. Then, we introduce a number called ω that represents the limit of the above sequence: $0 < 1 < 2 < \cdots < \omega$. Further, we can construct numbers beyond ω by putting a “copy” of natural numbers “beyond” ω:

$$0 < 1 < 2 < \cdots \omega < \omega + 1 < \omega + 2 < \cdots < \omega + \omega$$

For the time being, we treat operations such as $\omega + n$ symbolically. The number $\omega + \omega$ which can be also denoted as $\omega \cdot 2$ is the second limit ordinal that limits any number of the form $\omega + n, n \in \mathbb{N}$. Such a procedure of constructing limit ordinals out of already constructed smaller ordinals can be applied recursively. Consider a sequence of $\omega \cdot n$ numbers and its limit:

$$0 < \omega < \omega \cdot 2 < \omega \cdot 3 < \cdots < (\omega \cdot \omega = \omega^2)$$

and we can carry on further to ω^n, ω^ω, *etc.* Note though that in the interval from ω^2 to ω^3 we have infinitely many limit ordinals of the form:

$$\omega^2 < \omega^2 + \omega < \omega^2 + \omega \cdot 2 < \cdots < \omega^3$$

and between any two of these we have a "copy" of the natural numbers:

$$\omega^2 + \omega < \omega^2 + \omega + 1 < \cdots < \omega^2 + \omega \cdot 2$$

3.2.1 *Formal definitions.* 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 $a \in X$, $X_a \stackrel{\text{def}}{=} \{x \in X | x < a\}$. 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. We use *islim* x to denote that x is a limit ordinal.

3.2.2 *Arithmetic on Ordinals.*

Addition. Ordinal addition is defined as $\alpha + \beta = Ord(A, <_A)$, where $A = \{0\} \times \alpha \cup \{1\} \times \beta$ and $<_A$ is the lexicographic ordering on A. Ordinal addition is associative but *not* commutative. As an example consider expressions $2 + \omega$ and $\omega + 2$. The former can be seen as follows: $0 < 1 < 0' < 1' < \cdots$, which after relabeling is isomorphic to ω. However, the latter can be seen as: $0 < 1 < \cdots < 0' < 1'$, which has the largest element $1'$, whereas ω does not. Therefore $2 + \omega = \omega < \omega + 2$. We have used $0'$, $1'$ to indicate the right hand side argument of the addition.

Subtraction. Ordinal subtraction can be defined in two ways, as partial inverse of the addition on the left and on the right. For left subtraction, which will be used by default throughout this paper unless otherwise specified, $\alpha - \beta$ is defined when $\beta \le \alpha$, as: $\exists \xi : \beta + \xi = \alpha$. As ordinal addition is left-cancelative ($\alpha + \beta = \alpha + \gamma \implies \beta = \gamma$), left subtraction always exists and it is unique.

Right subtraction is a bit harder to define as:

- it is not unique: $1 + \omega = 2 + \omega$ but $1 \ne 2$; therefore $\omega -_R \omega$ can be any number that is less than ω: $\{0, 1, 2, \dots\}$.
- even if $\beta < \alpha$, the difference $\alpha - \beta$ might not exist. For example: $42 < \omega$; however, $\omega -_R 42$ does not exist as $\nexists \xi : \xi + 42 = \omega$.

Despite those difficulties, right subtraction can be useful at times and can be defined for $\alpha -_R \beta$:

$$\min\{\xi : \xi + \beta = \alpha\}$$

Multiplication. Ordinal multiplication $\alpha \cdot \beta = Ord(A, <_A)$ where $A = \alpha \times \beta$ and $<_A$ is the lexicographic ordering on A. Multiplication is associative and left-distributive to addition:

$$\alpha \cdot (\beta + \gamma) = (\alpha \cdot \beta) + (\alpha \cdot \gamma)$$

However, multiplication is not commutative and is not distributive on the right: $2 \cdot \omega = \omega < \omega \cdot 2$ and $(\omega + 1) \cdot \omega = \omega \cdot \omega < \omega \cdot \omega + \omega$.

Exponentiation. Exponentiation can be defined using transfinite recursion: $\alpha^0 = 1, \alpha^{\beta+1} = \alpha^\beta \cdot \alpha$ and for limit ordinals λ: $\alpha^\lambda = \bigcup_{0<\xi<\lambda} \alpha^\xi$.

ϵ-ordinals. Using ordinal operations we can construct the following hierarchy of ordinals: $0, 1, \dots, \omega, \omega+1, \dots, \omega \cdot 2, \omega \cdot 2+1, \dots, \omega^2, \dots, \omega^3, \dots \omega^\omega, \dots$. The smallest ordinal for which $\alpha = \omega^\alpha$ is called ϵ_0. It can also be seen as a limit of the following $\omega^\omega, \omega^{\omega^\omega}, \dots, \omega^{\omega^{\cdots}}$.

3.2.3 Cantor Normal Form. For every ordinal $\alpha < \epsilon_0$ there are unique $n, p < \omega, \alpha_1 > \alpha_2 > \cdots > \alpha_n$ and $x_1, \dots, x_n \in \omega \setminus \{0\}$ such that $\alpha > \alpha_1$ and $\alpha = \omega^{\alpha_1} \cdot x_1 + \cdots + \omega^{\alpha_n} \cdot x_n + p$. Cantor Normal Form makes provides a standardized way of writing ordinals. It uniquely represents each ordinal as a finite sum of ordinal powers, and can be seen as an ω based polynomial. This can be used as a basis for an efficient implementation of ordinals and their operations.

3.3 λ_ω: Adding Ordinals to λ_α

The key contribution of this paper is the introduction of λ_ω, a variant of λ_α, which use ordinals as shapes and indices of arrays and which reestablishes global equalities in the context of infinite arrays.

Before revisiting the equalities, we look at the changes to λ_α that are required to support transfinite arrays. Syntactically, to introduce ordinals in the language, we make two minor additions to λ_α. Firstly, we add ordinals[4] as scalar constants. Secondly, we add a built-in operation, *islim*, which takes one argument and returns *true* if the argument is a limit ordinal and *false* otherwise. For example: *islim* ω reduces to *true* and *islim* $(\omega + 21)$ reduces to *false*.

λ_α with ordinals — extends λ_α

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
  & | & \mathit{islim} & \text{(limit ordinal predicate)} \\
c & ::= & \cdots & \\
  & | & \omega, \omega+1, \dots & \text{(ordinals)}
\end{array}
$$

Fig. 3. The syntax of λ_ω.

Semantically, it turns out that all core rules can be kept unmodified apart from the aspect that all helper functions, arithmetic, and relational operations now need to be able to deal with ordinals instead of natural numbers. In particular, the semantic for lazy *imap*s as developed for λ_α^∞ can be used unaltered, provided that all helper functions involved such as for splitting generators *etc.* are expanded to cope with ordinals.

3.4 Array Equalities Revisited

With the support of Ordinals in λ_ω, we can now revisit our equalities Eq. 1 and 2. Let us first look at the counter example for Eq. 1: from Section 3.1: With an array shape $s = [2, \omega]$ and a legal index vector into s $[1, n]$, we now obtain:

$$
\begin{aligned}
o2i\ (i2o\ [1, n]\ [2, \omega])\ [2, \omega]) &= o2i\ (\omega + n)\ [2, \omega] \\
&= [(\omega + n)\ /\ \omega,\ \ (\omega + n)\ \%\ \omega] \\
&= [1,\ n]
\end{aligned}
$$

[4]Technically, we support ordinal values only up to ω^ω, as ordinals are constructed using the constant ω and $+, -, *, /$ and % operations (no built-in ordinal exponentiation).

The crucial difference to the situation from λ_α^∞ in Section 3.1 here is the ability to divide $(\omega + n)$ by ω and to obtain a remainder, denoted by %, of that division as well. By means of induction over the length of the shape and index vectors this equality can be proven to hold for arbitrary shapes in λ_ω, and, based on this proof, Eq. 1 can be shown as well.

In the same way as the arithmetic on ordinals is key to the proof of Eq. 1, it also enables the proof of Eq. 2 for arbitrary ordinal-shaped vectors[5] a and b, with the definition of ++ from the previous section and *drop* being defined as:

```
drop ≡ λs.λa. imap |a|∸s { [0] <= iv < |a|∸s: a.(s∔iv)
```

After inlining ++ and *drop*, the left hand side of Eq. 2 can be rewritten as:

```
letrec ab = imap |a|∔|b| { [0] <= jv < |a|: a.jv,
                           |a| <= jv < |a|∔|b|: b.(jv∸|a|) in

imap |ab|∸|a| { [0] <= iv < |ab|∸|a|: ab.(|a|∔iv)
```

Consider the shape of the goal expression of the letrec. According to the semantics of the shape of an *imap*, we get: $|ab| \dot{-} |a|$. The shape of ab is $|a| \dot{+} |b|$. According to ordinal arithmetic: $(|a| \dot{+} |b|) \dot{-} |a|$ is $|b|$. Therefore the shapes of right-hand and left-hand sides of the goal expressions are the same.

Let us rewrite the last *imap* as:

```
imap |b| { [0] <= iv < |b|: ab.(|a|∔iv)
```

Consider now selections into *ab*. All the selections into *ab* will happen at indices that are greater than *a*. This is because all the legal *iv* in the *imap* are from the range $[0]$ to $|b|$.

According to the semantics of selections into *imap*s, $ab.(|a| \dot{+} iv)$ will select from the second partition of the *imap* that defines *ab*, and will evaluate to: $b.((|a| \dot{+} iv) \dot{-} |a|)$. According to ordinal arithmetic, $(|a| \dot{+} iv) \dot{-} |a|$ is identical to *iv*, therefore we can rewrite the previous *imap* as:

```
imap |b| { [0] <= iv < |b|: b.iv
```

As it can be seen, this is an identity *imap*, which is extensionally equivalent to b.

4 EXAMPLES

Transfinite tail. As explained in Section 3.3, the shift from natural numbers to ordinals as indices in λ_ω implies corresponding extensions of the built-in arithmetic operations. As these operations lose key properties, such as commutativity, once arguments exceed the range of natural numbers, we need to ensure that function definitions for finite arrays extend correctly to the transfinite case.

As an example, consider the definition of *tail* from the previous section:

```
tail ≡ λa.imap |a|∸[1] {_(iv): a.([1]∔iv)
```

For the case of finite vectors, we can see that a vector shortened by one element is returned, where the first element is missing and all elements have been shifted to the left by one element.

Let us assume we apply tail to an array a with $|a| = [\omega]$. The arithmetic on ordinals gives us a return shape of $[\omega] \dot{-} [1] = [\omega]$. That is, the tail of an infinite array is the same size as the array itself, which matches our common intuition when applying tail to infinite lists. The elements of that infinite list are those of a, shifted by one element to the right, which, again, matches our expected interpretation for lists.

Now, assume we have $|a| = [\omega + 42]$, which means that $(\textit{tail}\ a).[\omega]$ should be a valid expression. For the result shape of tail a, we obtain $[\omega + 42] \dot{-} [1] = [\omega + 42]$. A selection $(\text{tail}\ a).[\omega]$ evaluates to $a.([1] \dot{+} [\omega]) = a.[\omega]$. This means that the above definition of the tail shifts all the elements at indices smaller than $[\omega]$ one left, and leaves all the other unmodified. While this may seem

[5] Eq. 2 can be generalised and shown to hold in the multi-dimensional case, provided that ++ and *drop* operate over the same axis.

counter-intuitive at first, it actually only means that *tail* can be applied infinitely often but will never be able to reach "beyond" the first limit.

Finally, observe that the body of the *imap*-construct in the definition of *tail* uses $[1] \dotplus iv$ is an index expression, not $iv \dotplus [1]$. In the latter case, the tail function would behave differently beyond $[\omega]$: it would attempt to shift elements to the left. However, this would make the overall definition faulty. Consider again the case when $|a| = [\omega + 42]$: the shape of the result would be $|a|$, which would mean that it would be possible to index at position $[\omega + 41]$, triggering evaluation of $a.([\omega + 41] \dotplus [1])$ and consequently, producing an *index error*, or out-of-bounds access into a.

Zip. Let us now define zip of two vectors that produces a vector of tuples. Consider a Haskell definition of zip function first:

```
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []
```

The result is computed lazily, and the length of the resulting list is a minimum of the lengths of the arguments. Like concatenation, a literal translation into λ_ω is possible, but it has the same drawbacks, *i.e.* it is restricted to arrays whose shape has no components bigger than ω.

A better version of zip that can be applied to arbitrary transfinite arrays looks as follows:

```
zip ≡ λa.λb.imap (min |a| |b|)|[2] {_(iv): [a.iv, b.iv]
```

Here, we use a constant array in the body of the *imap*. This forces evaluation of both arguments, even if only one of them is selected. This can be changed by replacing the constant array with an *imap*:

```
zip ≡ λa.λb.imap (min |a| |b|)|[2] {_(iv): imap [2] { [0] <= jv < [1] a.iv,
                                                      [1] <= jv < [2] b.iv
```

which can be fused in a single *imap* as follows:

```
zip ≡ λa.λb.letrec s = (min |a| |b|).[0] in
            imap [s,2] { [0,0] <= iv < [s, 1]: a.[iv.[0]],
                         [0,1] <= iv < [s, 2]: b.[iv.[0]]
```

Data Layout and Transpose. A typical transformations in stream programming is changing the granularity of a stream and joining multiple streams. In λ_ω, these transformations can be expressed by manipulating the shape of an infinite array. Consider changing the granularity of a stream a of shape $[\omega]$ into a stream of pairs:

```
imap (|a|/̇[2])|[2] { _(iv): [a.[2*iv.[0]], a.[2*iv.[0]+1]]
```

or we can express the same code in a more generic fashion:

```
(λn.reshape ((|a|/̇[n])++[n]) a) 2
```

This code can operate on the streams of transfinite length, as well. If we envision compiling such a program into machine code, the infinite dimension of an array can be seen as a time-loop, and the operations at the inner dimension seen as a stream-transforming function. Such granularity changes are often essential for making good use of (parallel) hardware resources, *e.g.* FPGAs.

Transposing a stream makes it possible to introduce synchronisation. Consider transforming a stream a of shape $[2, \omega]$ into a stream of pairs (shape $[\omega, 2]$):

```
imap [ω]|[2] { _(iv): [a.[iv.[0],0], a.[iv.[0],1]]
```

Conceptually, an array of shape $[2, \omega]$ represents two infinite streams that reside in the same data structure. An operation on such a data structure can progress independently on each stream, unless some dependencies on the outer index are introduced. A transpose, as above, makes it possible to introduce such a dependency, ensuring that the operations on both streams are synchronized. A key to achieving this is the ability to re-enumerate infinite structures, and ordinal-based infinite arrays make this possible.

Ackermann function. The true power of multidimensional infinite arrays manifests itself in definitions of non-primitive-recursive sequences as data. Consider the Ackermann function, defined as a multi-dimensional stream:

```
letrec a = imap [ω, ω] {_(iv): letrec m = iv.[0] in
                               letrec n = iv.[1] in
                               if m = 0 then n + 1
                               else if m > 0 and n = 0 then a.[m-1, 1]
                               else a.[m-1, a.[m,n-1]] in a
```

Such a treatment of multi-dimensional infinite structures enables simple transliteration of recursive relations *as data*. Achieving similar recursive definitions when using cons-lists is possible, but they have a subtle difference. Consider a Haskell definition of the Ackermann function in data:

```
a = [[ if m == 0 then n+1
       else if m > 0 then a !! (m-1) !! 1
       else a !! (m-1) !! (a !! m !! (n-1))
     | n <- [0..]]
    | m <- [0..]]
```

We use two [0..] generators for explicit indexing, even though at runtime, all necessary elements of the list will be present. The lack of explicit indexes forces one to use extra objects to encode the correct dependencies, essentially implementing *imap* in Haskell. Conceptually, these generators constitute two further locally recursive data structures. Whether they can be always can be optimised away is not clear. Avoiding these structures in an algorithmic specification can be a major challenge.

Game of Life. As a final example, consider Conway's Game of Life which describes an evolution of cells on a plane. The most interesting aspect of this example is the fact that we can encode it in λ_ω in such a way that the shape of the plane is never specified. This means that the program can operate with infinite planes, *e.g.* of shape $[\omega, \omega]$, as well as finite 2d planes with no changes to source code.

First we introduce a few generic helper functions:

```
(or) ≡ λa.λb.if a then a else b
(and) ≡ λa.λb.if a then b else a
any ≡ λa.reduce or false a
gen ≡ λs.λv.imap s {_(iv): v
↖ ≡ λv.λa.imap |a| {_(iv): if any (iv∔v >̇= |a|) then 0 else a.(iv∔v)
↘ ≡ λv.λa.imap |a| {_(iv): if any (iv <̇ v) then 0 else a.(iv∸v)
```

or and and encode logical conjunction and disjunction, respectively. any folds an array of boolean expressions with the disjunction, and gen defines an array of shape s whose values are all identical to v. More interesting are the functions ↖ and ↘. Given a vector v and an array a, they shift all elements of a towards decreasing indices or increasing indices by v elements, respectively. Missing elements are treated as the value 0.

Now, we define a single step of the 2-dimensional Game of Life in APL style[6]: two-dimensional array a by:

```
gol_step ≡ λa.
    letrec F = [↖ [1,1], ↖ [1,0], ↖ [0,1], λ x. ↖ [1,0] (↘ [0,1] x),
                ↘ [0,1], ↘ [1,0], ↘ [1,1], λ x. ↘ [1,0] (↖ [0,1] x)]
    in letrec
       c = (reduce (λf.λg.λx.f x ∔ g x) (λx.gen |a| 0) F) a
    in
```

[6]See this video by John Scholes for more details: https://youtu.be/a9xAKttWgP4

```
imap |a| { _(iv): if (c.iv = 2 and a.iv = 1) or (c.iv = 3)
                  then 1
                  else 0
```

We assume an encoding of a live cell in a to be 1, and a dead cell to be 0. The array F contains partial applications of the two shift functions to two-element vectors so that shifts into all possible directions are present. The actual counting of live cells is performed by a function which folds F with the function $\lambda f.\lambda g.\lambda x.f\ x + g\ x$. This produces c, an array of the same shape as a, holding the numbers of live cells surrounding each position. Defining the shift operations $\nwarrow$ and $\searrow$ to insert 0 ensures that all cells beyond the shape of a are assumed to be dead.

The definition of the result array is, therefore, a straightforward *imap*, implementing the rules of birth, survival and death of the Game of Life.

5 TRANSFINITE ARRAYS *VS*. STREAMS

Streams have attracted a lot of attention due to the many algebraic properties they expose. [Hinze 2010] provides a nice collection of examples, many of which are based on the observation that streams form an applicative functor. Transfinite arrays are applicative functors as well, not only for arrays of shape $[\omega]$, but also for any given shape *shp*. With definitions:

```
pure ≡ λx.imap shp {_(iv): x
(⋄) ≡ λa.λb.imap shp {_(iv): a.iv b.iv
```

we obtain for arbitrary arrays u, v, w, and x of shape *shp*:

$$(\text{pure } \lambda x.x) \diamond u == u \qquad (\text{pure } (\lambda f.\lambda g.\lambda x.f\ (g\ x))) \diamond u \diamond v \diamond w == u \diamond (v \diamond w)$$

$$(\text{pure } f) \diamond (\text{pure } x) == \text{pure } (f\ x) \qquad u \diamond (\text{pure } x) == (\text{pure } (\lambda f.f\ x)) \diamond u$$

This shows that arbitrarily shaped arrays of finite size have this property, as also shown by [Gibbons 2017], and that these properties can be expanded into ordinal-shaped arrays. Classical streams are a special instance of these, *i.e.* arrays of shape $[\omega]$.

For stream operations that insert or delete elements, it is less obvious whether these can be easily extended into ordinal-shaped arrays other than shape $[\omega]$. As an example, let us consider the function *filter*, which takes a predicate p and a vector v and returns a vector that contains only those elements x of v that satisfy $(p\ x)$. A direct definition of *filter* can be given as:

```
filter ≡ λp. λv. if (p v.[0]) then v.[0] ++ filter p (tail v)
                              else filter p (tail v)
```

This definition, in principle, is applicable to arrays of any ordinal shape, but the use of *tail* in the recursive calls inhibits application beyond ω. Furthermore, the strict semantics of λ_ω inhibits a terminating application to any infinite array, including arrays of shape $[\omega]$. For the same reason, a definition of *filter* through the built-in *reduce* is restricted to finite arrays.

To achieve possible termination of the above definition of *filter* for transfinite arrays, we would need to change to a lazy regime for all function applications in λ_ω and we would need to change the semantics of *imap* into a variant where the shape computation can be delayed as well. Even if that would be done, we would still end up with an unsatisfying solution. The filtering effect would always be restricted to the elements before the first limit ordinal ω. This limitation breaks several fundamental properties, like those defined in [Bird 1987], that hold in the finite and stream cases. As an example, consider distributivity of filter over concatenation:

$$\mathit{filter}\ p\ (a \mathbin{+\!\!+} b) == (\mathit{filter}\ p\ a) \mathbin{+\!\!+} (\mathit{filter}\ p\ b) \tag{3}$$

This property holds for finite arrays, but fails with the above definition of *filter* in case a is infinite.

To regain this property for transfinite arrays, we need to apply *filter* to all elements of the argument array, not only those before the first limit ordinal ω. When doing this in the context of λ_ω, the necessity to have a strict shape for every object forces us to "guess" the shape of the filtered result in advance. The way we "guess" has an impact on the filter-based equalities that will hold universally.

In this paper we propose a scheme that respects the above equality. For finite arrays *filter* works as usual, and for the infinite ones, we postulate that the result of filtering will be of an infinite-shape:

$$\forall p \forall a : |a| \geq \omega \implies |\mathit{filter}\ p\ a| \geq \omega$$

This is further applied to all infinite sequences contained within the given shape as follows:

$$\forall i < |a| : (\exists \mathit{islim}\ \alpha : i < \alpha \leq |a|) \implies (\exists k \in \mathbb{N} : p\ (a.(i+k)) = \mathit{true})$$

We assume that each infinite sequence contains infinitely many elements for which the predicate holds. Consequently, any limit ordinal component of the shape of the argument is carried over to the result shape and only any potential finite rest undergoes potential shortening. Consider a filter operation, applied to a vector of shape $[\omega \cdot 2]$. Following the above rationale, the shape of the result will be $[\omega \cdot 2]$ as well. This means that the result of applying *filter* to such an expression should allow indexing from $\{0, 1, \dots\}$ as well as from $\{\omega, \omega + 1, \dots\}$ delivering meaningful results.

This decision can lead to non-termination when there are only finitely many elements in the filtered result. For example:

```
filter (λx.x > 0) (imap [ω+2] {_(iv): 0)
```

reduces to an array of shape $[\omega]$, which effectively is empty. Any selection into it will lead to a non-terminating recursion.

The overall scheme may be counter-intuitive, but it states that for every index position of the output, the computation of the corresponding value is well-defined.

Assuming the aforementioned behaviour of *filter*, Eq. 3 holds for all transfinite arrays. Another universal equation that holds for all transfinite vectors concerns the interplay of *filter* and *map*:

$$\mathit{filter}\ p\ (\mathit{map}\ f\ a) == \mathit{map}\ f\ (\mathit{filter}\ (p \cdot f)\ a)$$

The proposed approach does not only respect the above equalities, but it also behaves similarly to filtering of streams that can be found in languages such as Haskell: *filter* applied to an infinite stream cannot return a finite result.

In principle, the chosen filtering scheme can be defined in λ_ω by using the *islim* predicate within an *imap*. However, the resulting definition is neither concise, nor likely to be runtime efficient. Given the importance of *filter*, we propose an extension of λ_ω. Fig. 4 shows the syntactical extension of λ_ω.

λ_ω with filters — extends λ_ω

$$\begin{array}{rcll} e & ::= & \cdots & \\ & | & \mathit{filter}\ e\ e & \text{(filter operation)} \end{array}$$

Fig. 4. The syntax of λ_ω with filters.

As *filter* conceptually is an alternative means of constructing arrays, its semantics is similar to that of *imap*. In particular, it constitutes a lazy array constructor, whose elements are being evaluated upon demand created through selections. Technically, this means that we have to introduce a new value to keep *filter*-closures, a rule that builds such a closure from *filter* expression, and we need to define the selection operation that forces evaluation within the filter closure.

We introduce as new value for *filter*-closures:

$$\left[\!\!\left[\mathit{filter}\; p_f\; p_e \begin{cases} \alpha_1 & v_r^1\; v_i^1 \\ \dots & \\ \alpha_n & v_r^n\; v_i^n \end{cases} \right]\!\!\right]$$

which contains the pointer to the filtering function p_f, the shape of the argument we are filtering over (p_e) and the list of partitions that consist of a limit ordinal, and a pair of partial result and natural number: v_r and v_i correspondingly.

On every selection at index $[\xi + n]$, where ξ is a limit ordinal or zero, and n is a natural number, we find a ξ partition within the filter closure or add a new one if it is not there. Every partition keeps a vector with a partial result of filtering (v_r), and the index (v_i) with the following property: the element in the array we are filtering over at position $\xi + (v_i - 1)$ is the last element in the v_r, given that $v_r > 0$. This means that if n is within v_r, we return $v_r.[n]$. Otherwise, we extend v_r until its length becomes $n + 1$ using the following procedure: inspect the element in p_e at the position $\xi + v_i$ — if the predicate function evaluates to *true*, append this element to v_r and increase v_i by one, otherwise, increase v_i by one.

A formal description of this procedure can be found in [Anonymous-1 2018, Sec. 2.1.4].

6 IMPLEMENTATION

We implement λ_ω in a system called Heh, which can be found in the anonymous supplementary materials. Heh contains:

(1) an interpreter for λ_ω covering the full language, and
(2) a compiler for the strict and finite subset of λ_ω.

The interpreter can be seen as a proof of concept that the proposed semantics is implementable. The implementation is an almost literal translation of the semantic rules provided in the paper into Ocaml code. We carefully implement updates in-place for *imap* and *filter* closures, ensuring that these constructs are evaluated lazily rather than in normal order. All examples provided in the paper can be found in that repository, and run, correctly, in Heh.

Compilation of the finite subset of λ_ω is achieved by translating λ_ω programs into SaC programs and subsequently using the compiler `sac2c` to produce binaries. Multi-core and GPU backends of `sac2c` can be leveraged to execute strict and finite λ_ω programs in parallel on these types of architectures. The Heh implementation comes with more than a 100 unit tests for its internal components.

In the interpreter, ordinals are represented by their Cantor Normal Form. The algorithms for implementing operations on ordinals are based on [Manolios and Vroon 2005]. In the same paper, we also find an in-depth study of the complexities of ordinal operations: comparisons, additions and subtractions have complexities $O(n)$, where n is the minimum of the lengths of both arguments; multiplications have the complexity $O(n \cdot m)$, where m and n are the lengths of the two argument representations.

6.1 Performance considerations

Our compiler for the strict and finite sublanguage of λ_ω shows that this part of the language can be mapped into languages such as SaC, leading to high-performance execution potential on variouss platforms [Šinkarovs et al. 2013; Wieser et al. 2012]. Whether the full-fledged version of λ_ω can be compiled into high-performance codes as well, mainly relies on the answers to two questions:

(1) how can we handle finite expressions that are defined by means of recursive *imap*s, and
(2) what is the most efficient representation for transfinite arrays.

Recursive imaps. Strict data parallel languages like SaC rarely support recursive *imap* constructs, even if the shape of the result is finite. There are two difficulties: (i) the evaluation of recursive *imap*s results in the necessity to support *imap* closures; (ii) parallel implementation of a recursive *imap* becomes trickier because of potential dependencies between the elements of an array. In [Anonymous-2 2018] we propose an elegant solution to this problem. We introduce a mechanism that switches from strict to lazy evaluation of a potentially recursive *imap*. It is demonstrated that the lifetime of *imap* closures is kept to a minimum and that a parallel implementation is possible. Furthermore, the proposed solution enables the detection of cyclic array definitions that diverge under strict semantics.

Data structures. The current semantics prescribes that, when evaluating selections into a lazy *imap*, the partition that contains the index that is to be selected from has to be split into a single-element partition and the remainder. This means that, as the number of selections into the *imap* increases, the structure that stores partitions of the *imap* will have to deal with a large number of single-element arrays. Partitions can be stored in a tree, providing $O(\log n)$ look-up; however triggering a memory allocation for every scalar is likely to be very inefficient. An alternate approach would be to allocate larger chunks, each of which would store a subregion of the index space of an *imap*. When doing so, we would need to establish a policy on the size of chunks and chose a mechanism on how to indicate evaluated elements in a chunk. Another possibility would be to combine the chunking with some strictness speculation, using a technique similar to the one presented in [Anonymous-2 2018]. That way, a single element selection could trigger the evaluation of an entire chunk.

Memory management. An efficient memory management model is not obvious. In case of strict arrays, reference counting is known to be an efficient solution [Cann 1989; Grelck and Scholz 2006]. For lazy data structures, garbage collection is usually preferable. Most likely, the answer lies in a combination of those two techniques.

The *imap* construct offers an opportunity for garbage collection at the level of partitions. Consider a lazy *imap* of boolean values with a partition that has a constant expression:

```
imap [ω] {..., l <= iv < u: false, ...
```

Assume further that neighbouring partitions evaluate to *false*. In this case, we can merge the boundaries of partitions and instead of keeping values in memory, the partition can be treated as a generator. However, an efficient implementation of such a technique is non-trivial.

Ordinals. An efficient implementation of ordinals and their operations is also essential. Here, we could make use of the fact that λ_ω is limited to ordinals up to ω^ω. For further details see [Anonymous-1 2018, Sec. 4]

7 RELATED WORK

Several works propose to extend the index domain of arrays to increase expressibility of a language. A straightforward way to do this is to stay within cardinal numbers but add a notion of ∞, similarly to what we have proposed in λ_α^∞. Similar approach is described in [McDonnell and Shallit 1980]; in J [Jsoftware 2016] infinity is supported as a value, but infinite arrays are not allowed. As we have seen, by doing so we lose a number of array equalities.

In [More 1973, page 137] we read: 'A restriction of indices to the finite ordinal numbers is a needless limitation that obscures the essential process of counting and indexing.' We cannot agree more. [More 1973] describes an axiomatic array theory that combines set theory and APL. The theory is self contained and gives rise to a number of array equalities. However, the question on how this theory can be implemented (if at all) is not discussed.

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 the plethora of array languages that evolved from it, 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, rendering all evaluation lazy.

Coinduction and codata are the usual way to introduce infinite data structures in programming languages [Jeannin et al. 2012; Kozen and Silva 2016]. Key to the introduction of codata typically is the use of coinductive semantics [Leroy and Grall 2009]. In our paper, the use of ordinals keeps the semantics inductive and deals with infinite objects by means of ordinals. In [Turner 1995], the author investigates a model of a total functional language, in which codata is used to define infinite data objects.

Streams are also related to dataflow models, such as [Estrin and Turn 1963; Kahn 1974; Petri 1962]. The computation 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 multi-dimensional infinite streams that cache their parts for better efficiency.

Two array representations, called *push arrays* and *pull arrays*, are presented in [Svensson and Svenningsson 2014]. Pull arrays are treated as objects that have a length and an index-mapping function; push arrays are structures that keep sequences of element-wise updates. The *imap* defined here can be considered an advanced version of a pull array, with partitions and transfinite shape. The availability of partitions circumvents a number of inefficiencies, (*e.g.* embedded conditionals) of classical pull arrays; the ordinals, in the context of the *imap*-construct, enable the expression of streaming algorithms.

The #Id language, presented in [Heller 1989], is similar to λ_ω; It combines the idea of lazy data structures with an eager execution context.

In [Atkey and McBride 2013; Møgelberg 2014], the authors propose a system that makes it possible to reason whether a computation defined on an infinite stream is productive[7] — a question that can be transferred directly to λ_ω. Their technique lies in the introduction of a clock abstraction which limits the number of operations that can be made before a value must be returned. This approach has some analogies with defining explicit "windows" on arrays, as for example proposed in [Hammes et al. 1999], or guarantees that programs run in constant space in [Lippmeier et al. 2016].

One of the key features of the array language described in this paper is the availability of strict shape for any expression of the language. Combining this with updates in place, which can be

[7]The computation will eventually produce the next item, *i.e.* it is not stuck.

achieved by means of monads [Wadler 1995], uniqueness typing [Barendsen and Smetsers 1996] or reference counting [Grelck and Scholz 2006], very efficient code generation becomes possible.

Strict shapes can be encoded in types as well. Specifically in the dependently-typed system, such an approach can be very powerful. The work on container theory [Abbott et al. 2005] allows a very generic description of indexed objects capturing ideas of shapes and indices in types. A very similar idea in the context of arrays is described in [Gibbons 2017]. The work on dependent type systems for array languages include [Slepak et al. 2014; Trojahner and Grelck 2009; Xi and Pfenning 1998]. Finally, a way to extend a type theory to include the notion of ordinals can be found in [Hancock 2000].

8 CONCLUSIONS AND FUTURE WORK

This paper proposes *transfinite arrays* as a basis for an applied λ-calculus λ_ω. The distinctive feature of transfinite arrays is their ability to capture arrays with infinitely many elements, while maintaining structure within that infiniteness. The number of axes is preserved, and individual axes can contain infinitely many infinite subsequences of elements. This capability extends many structural properties that hold for finite arrays into the transfinite space.

The embedding of transfinite arrays into λ_ω allows for recursive array definitions, offering an opportunity to transliterate typical list-based algorithms, including algorithms on infinite lists for stream processing, into a generic array-based form. The paper presents several examples to this effect, and provides some efficiency considerations for them. It remains to be seen if these considerations, in practice, enable a truly unified view of arrays, lists, and streams.

The array-based setting of λ_ω allows this recursive style of defining infinite structures to be taken into a multi-dimensional context, enabling elegant specification of inherently multi-dimensional problems on infinite arrays. As an example, we present an implementation of Conway's Game of Life which, despite looking very similar to a formulation for finite arrays, is defined for positive infinities on both axes. Within λ_ω, accessing neighbouring elements along both axes can be specified without requiring traversals of nested cons lists.

We also present an implementation for the Ackerman function, using a 2-dimensional transfinite array, one axis per parameter. The resulting code adheres closely to the abstract declarative formulation of the function, while also implicitly generating a basis for a memoising implementation of the algorithm.

An interesting aspect of transfinite arrays is that ordinal-based indexing opens up an avenue to express transfinite induction in data in very much the same way as *nil* and *cons* are duals to the principle of mathematical induction. This can not be done easily using *cons* lists as there is no concept of a limit ordinal in that context. It may be possible to encode this principle by means of nesting, but then one would need a type system or some sort of annotations to distinguish lists of transfinite length from nested lists. The *imap* construct from the proposed formalism can be seen as an elegant solution to this.

The fact that *imap* supports random access and is powerful enough to capture list and stream expressions alike opens up an exciting perspective for the implementation of λ_ω. When arrays are finite, it is possible to reuse one of the existing efficient array-based implementations. When arrays are infinite, we can use list or stream implementations to encode λ_ω, but at the same time the properties of the original λ_ω programs open the door to rich program analysis and alternative representations. We believe that many functional languages striving for performance could benefit from the proposed design, at least when the destinction between finite and infinite arrays can be statically determined be it through annotation or inference.

The concept of transfinite arrays as proposed in this paper offers several new and interesting possibilities for further investigation. As discussed in the implementation section, it is not yet clear what the most efficient implementations for our proposed infinite structures are. Choices of representation affect both memory management design and the guarantees that our semantics can provide.

Further research into type systems for λ_ω would also be interesting. Not only could type systems guarantee absense of indexing errors but they could deliver the destinction between finite and infinite casesi as well. The type system we describe in the paper can serve as a starting point. Decidability aspects around ordinals have raised interest independently. The first-order theory of ordinal addition is known to be decidable [Büchi 1990], but more complex ordinal theories can quickly get undecidable [Choffrut 2002].

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. 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.

Anonymous-1. 2018. Operational Semantics of Lambda-omega. Can be found in the anonymous supplementary materials. (2018). heh/semantics/lambda-omega-semantics.pdf

Anonymous-2. 2018. Recursive Array Comprehensions in a Call-by-Value Language. Can be found in the anonymous supplementary materials. (2018). heh/array-comprehensions/rec-array-comprehensions.pdf

Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/2500365.2500597

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.

Robert Bernecky. 1993. The Role of APL and J in High-performance Computation. *ACM SIGAPL Quote Quad* 24, 1 (Aug. 1993), 17–32.

Robert Bernecky and Paul Berry. 1993. *SHARP APL Reference Manual* (2nd ed.). Iverson Software Inc., Iverson Software Inc., 33 Major St., Toronto, Canada.

Robert Bernecky and Kenneth E. Iverson. 1980. Operators and Enclosed Arrays. In *APL Users Meeting 1980*. I.P. Sharp Associates Limited, I.P. Sharp Associates Limited, Toronto, Canada, 319–331.

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

Larry M. Breed, Roger D. Moore, Luther Woodrum, Richard Lathwell, and Adin Falkoff. 1972. (1972). http://www.computerhistory.org/atchm/the-apl-programming-language-source-code The APL\360 source code is available at the Computer History Museum.

J. Richard Büchi. 1990. *Transfinite Automata Recursions and Weak Second Order Theory of Ordinals*. Springer New York, New York, NY, 437–457. https://doi.org/10.1007/978-1-4613-8928-6_24

D.C. Cann. 1989. *Compilation Techniques for High Performance Applicative Computation*. Technical Report CS-89-108. Lawrence Livermore National Laboratory, LLNL, Livermore California.

Christian Choffrut. 2002. *Elementary Theory of Ordinals with Addition and Left Translation by ω*. Springer Berlin Heidelberg, Berlin, Heidelberg, 15–20. https://doi.org/10.1007/3-540-46011-X_2

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

Martin Elsman and Martin Dybdal. 2014. Compiling a Subset of APL Into a Typed Intermediate Language. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, USA, Article 101, 6 pages. https://doi.org/10.1145/2627373.2627390

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

Peter Falster and Michael Jenkins. 1999. *Array Theory and Nial.*

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

J. I. Glasgow and M. A. Jenkins. 1988. Array theory, logic and the Nial language. In *Proceedings. 1988 International Conference on Computer Languages.* 296–303. https://doi.org/10.1109/ICCL.1988.13077

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

Jeffrey P. Hammes, Bruce A. Draper, and A. P. Willem Böhm. 1999. *Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 83–97. https://doi.org/10.1007/3-540-49256-9_6

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

S.K. Heller. 1989. *Efficient lazy data structures on a data-flow machine. Technical report.* Number TR-438.

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

Roger K.W. Hui and Kenneth E. Iverson. 1998. *J Dictionary.*

IBM. 1994. *APL2 Programming: Language Reference* (second ed.). IBM Corporation. SH20-9227.

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

Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2012. *CoCaml: Programming with Coinductive Types.* Technical Report http://hdl.handle.net/1813/30798. Computing and Information Science, Cornell University.

Michael A. Jenkins and Lenore R. Mullin. 1991. *A Comparison of Array Theory and a Mathematics of Arrays.* Springer US, Boston, MA, 237–267. https://doi.org/10.1007/978-1-4615-4002-1_14

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

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

G. Kahn. 1987. Natural semantics. In *STACS 87*, FranzJ. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Lecture Notes in Computer Science, Vol. 247. Springer Berlin Heidelberg, 22–39. https://doi.org/10.1007/BFb0039592

Dexter Kozen and Alexandra Silva. 2016. Practical coinduction. *Mathematical Structures in Computer Science* FirstView (February 2016), 1–21. https://doi.org/10.1017/S0960129515000493

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (Feb. 2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004

Ben Lippmeier, Fil Mackay, and Amos Robinson. 2016. Polarized Data Parallel Data Flow. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing (FHPC 2016).* ACM, New York, NY, USA, 52–57. https://doi.org/10.1145/2975991.2975999

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

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.

Rasmus Ejlers Møgelberg. 2014. A Type Theory for Productive Coprogramming via Guarded Recursion. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14).* ACM, New York, NY, USA, Article 71, 10 pages. https://doi.org/10.1145/2603088.2603132

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

Lenore Mullin and Scott Thibault. 1994. *A Reduction Semantics for Array Expressions: The PSI Compiler.* Technical Report CSC-94-05. University of Missouri-Rolla.

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

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

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410.* Springer-Verlag New York, Inc., New York, NY, USA, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

Robert Stephens. 1997. A survey of stream processing. *Acta Informatica* 34, 7 (01 Jul 1997), 491–541. https://doi.org/10.1007/s002360050095

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

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 179–196. http://dl.acm.org/citation.cfm?id=647478.727935

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).

D. A. Turner. 1995. *Elementary strong functional programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–13. https://doi.org/10.1007/3-540-60675-0_35

A. Šinkarovs, S.B. Scholz, R. Bernecky, R. Douma, and C. Grelck. 2013. SAC/C Formulations of the All-Pairs N-Body Problem and their Performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience* (2013). https://doi.org/10.1002/cpe.3078

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

V. Wieser, C. Grelck, P. Haslinger, J. Guo, F. Korzeniowski, R. Bernecky, B. Moser, and S.B. Scholz. 2012. Combining High Productivity and High Performance in Image Processing Using Single Assignment C on Multi-core CPUs and Many-core GPUs. *Journal of Electronic Imaging* 21, 2 (2012). https://doi.org/10.1117/1.JEI.21.2.021116

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