To Stream or Not to Stream — That is Not the Question

Artjoms Šinkarovs and Sven-Bodo Scholz

Heriot-Watt University, Edinburgh, Scotland
a.sinkarovs@hw.ac.uk, s.scholz@hw.ac.uk

Abstract. This paper proposes a unification of stream programming and array programming. It introduces an array calculus that supports infinite ranges for multi-dimensional arrays. We show that such a setting is not only suitable to elegantly specify array programs as well as streaming applications, we also demonstrate that such a unified core language opens up completely novel opportunities. Programs can be written in a generic fashion allowing for finite as well as for infinite applications of the same algorithm; several algebraic properties of array operations on finite arrays can be carried over into the infinite case; arrays can be defined in a recursive fashion without requiring the programmer to statically identify dependencies between array elements; and such arrays can be seen as a memoisation mechanism for discrete functions.
We provide a definition of such an array language, sketch its capability to shallowly embed classical streaming languages, show a few universal equalities of the language, and we present many examples showcasing the various benefits of the array-only setting. Finally, we provide a brief discussion of a prototypical implementation of the language.

1 Introduction

In high performance computing, the term *streaming application* has at least two connotations. First, the term refers to applications that need to process data arriving as a sequence of items, *e.g.* a stream of observations or a stream of frames captured by a video camera. Secondly, the term streaming refers to applications that operate on finite data sets but choose to split those sets into a sequence of subsets. This can be done for efficient processing through FPGA or GPU compute devices, or for minimising the amount of data that has to be stored in RAM at any given time.

Despite the fundamental similarities between these two application types, their specifications, *i.e.* the source codes, differ significantly. The first type of applications explicitly operates on frames of data, and progression from one frame to the next is explicit. The second type has no notion of the temporal element at all. Instead, their source codes simply iterate over array assignments. Rather than being explicit, the notion of streaming in these cases typically is introduced by elaborate compiler technology *e.g.* polyhedral code generators.

This paper proposes to *unify* these two forms of streaming on the specificational level, *i.e.* at the level of the programming language, entirely shifting decisions on whether or how to stream any of these applications into the realm of a compiler.

The potential advantages of such a unification are manifold. Algorithms that are written with finite structures in mind can be applied to potentially infinite streams without any modifications and vice versa. Compiler technology developed for one context directly helps the other context; in particular when considering advanced code generation for parallel executions this is a huge benefit. Having a unified framework for streaming applications and loop nests over array assignments also enriches potential reasoning about programs as properties can possibly be carried over from one framework to the other.

The key insight of the proposed unification lies in the interpretation of streams as transfinite arrays. Based on a formal semantics, we introduce the unifying framework, its initial implementation and we demonstrate how programs can be written in a *stream-agnostic* style, liberating programmers from the necessity to answer the question "to stream or not to stream". We discuss performance related aspects when trying to compile such programmes into highly efficient parallel codes. In particular, we show how several existing techniques can be applied without any or with minor extensions only. Finally, we identify further research directions by sketching solutions for some of the challenges that arise when integrating the proposed framework into existing compilers.

2 Streaming Finite Arrays

Before diving into the world of infinite objects, let us work out the finite case first. The core of this work is based on the observation that array-based languages capture enough information to turn regular programs into streaming programs.

Array-based languages make it natural to express operations on entire arrays rather than on individual scalars. As examples of such languages you may consider Matlab, APL, Julia or R. It turns out that this idea of array programming can be embedded nicely into the setting of functional programming languages. Examples for such an embedding are SaC [18], DpH [6], Accelerate [7], Futhark [11], or Feldspar [1]. These languages combine high-level abstractions with powerful opportunities for program optimisations and code generation. Most importantly, they all support generation of high-performance codes for parallel architectures.

The core of functional array languages is typically built around a small set of array generating skeletons. Usually these are variants of map, reduce or scan combinators that relate generators of index sets to computations of corresponding array elements. We refer to these generically with the term *array comprehensions*.

For our discussion, the choice of a particular language is irrelevant. In order to keep the discussion language independent, we use an abstract minimalistic language called Heh. It is an applied λ-calculus extended by an array compre-

hension construct called *imap* (index mapping). At this stage, the reader can assume his favourite array programming language instead, as Heh can be easily mapped into languages such as SaC, Futhark or Julia. We explain the basics of Heh as we need them in our examples. More details on Heh can be found in [] as well as in Section 4.

As a motivating example, consider a simple convolution kernel. We look at a classical 1-dimensional three point stencil code over an input array a. In Heh, this code can be specified as

```
letrec stencil = λa.
    letrec n = |a|.[0] in
    imap [n] { [0] <=   iv < [1]:   a.iv,
               [1] <=   iv < [n-1]: (a.[iv.[0]-1]
                                     + a.iv + a.[iv.[0]+1]) / 3,
               [n-1] <= iv < [n]:   a.iv
```

We define a function `stencil` that takes one argument called `a`. It first computes the length `n` of the 1-dimensional array `a` and then uses the *imap* operator to define the result of the convolution. The first parameter of *imap* specifies the overall shape of the result to be a vector of length `n`. The shape-expression is followed by the symbol { after which a definition of how the elements at any index position `iv` are to be computed follows. For convenience, we allow the element computations to be defined piecewise for one or more index ranges. The programmer can specify pairs of index range and element-computations which we refer to as *partititions*. In our example, we distinguish three different index ranges: The first and the last partition describe static boundaries — the value there is computed as selections into `a` at the corresponding positions. The second partition describes that for a given index `iv`, we evaluate selections into `a` at index positions `iv`, `iv-1` and `iv+1`, we add those values and divide them by three.

Using vertical bars around an expression computes the given expression's shape and the infix dot operator denotes element selection. One aspect to notice here is that in Heh the indices for selection need to be vectors rather than scalar values. Another aspect to notice is that the above specification of the *imap* operator never specifies dependencies between the elements of the resulting array. As the name of the operator suggests, it is an index mapping. This means that at runtime, the above index space can be evaluated in parallel or in arbitrary order.

Advanced compiler technology is capable to either produce parallel code which computes a result array in freshly allocated memory or to produce sequential code which performs the computation in place, *i.e.* the result is stored in the same memory as the input array. The latter roughly leads to C code of the form:

```
double * stencil (double * a, size_t n) {
    double t0 = a[0], t1 = a[1], t2 = a[2];
    for (size_t i = 1; i < n-1; i++)
      {
        a[i] = (t0 + t1 + t2) / 3;
        t0 = t1;
        t1 = t2;
```

```
            t2 = a[i+2];
        }
    return a;
}
```

The function `stencil` gets two arguments, a 1-dimensional array `a` and its length `n`. It traverses the array left-to-right, leaving out the first and the last elements. At every iteration, the variables `t0`, `t1` and `t2` are being updated and the actual stencil operation is being computed for the given index.

Effectively, `t0`, `t1` and `t2` implement a circular buffer where at every iteration a new element is being pushed into it and one elements is being pushed out. Secondly, the array `a` can be seen as a stream that unfolds one element at a time. If we would output our result into a different array, after `a[i]` has been computed, it can immediately be discarded. Finally, this code is invariant to the length of the array `a`, it is perfectly suitable to operate on an infinite array `a`.

A key to this optimisation is the observation that under the given traversal of the index space, a buffer of three elements is sufficient to implement update in-place. Whenever the scope of array accesses can be determined such a transformation into a streaming code can be automated by the compiler. The needed information can either be inferred by static analysis or provided through some program annotation or a combination of both. Many approaches to this are possible: we can blindly believe programmer annotations or we can reject a program in case the annotations cannot be verified statically or dynamically. Irrespectively of these choices, we can see that it is possible to obtain streaming implementations from array program specifications making highly efficient streaming solutions from array specifications plausible.

Such a unification of specifications offers a number of immediate benefits. First, if we express streaming as array operations and the input data turns out to be finite, we immediately benefit from the full power of the compiler technology of existing array languages.

Secondly, such a unification implies that streaming-specific optimisations will find their way into the context of array compiler technology as well. This will not only benefit classical streaming problems but also streaming solutions in classical finite array problems as well.

Finally, we can write programs that are polymorphic over finite arrays and streams *i.e.* programs that are applicable to finite and infinite objects alike. We can use this representation to construct a proof that a given program can operate on infinite data in constant space.

3 Streaming Applications as Programs on Arrays

Before fleshing out more details about a unified calculus for arrays and streams we first want to look at typical streaming codes and investigate if and how these can be re-formulated in an array-based language. Many streaming languages have been proposed over time. They all differ in various details but have the overall setup in common. They typically have some basic building blocks which

compute streams of output values from streams of input values. In the context of this paper, we look at the streaming language Streamit [19] as a point point of reference.

In Streamit, the basic building blocks are named *filters*. A filter is either a stateless or stateful computation that maps input values to output values. More complex filters can be composed of the basic filters by using three combinators that are hard-wired into the definition of Streamit: *Pipeline*, *SplitJoin*, and *FeedbackLoop*. The Pipeline combinator combines two filters into a single one by feeding the output of the first filter into the input of the second filter. SplitJoin cuts a stream into n sub-streams, applies each sub-stream to one of n filters and rejoines the outputs into a single stream again. FeedbackLoop allows the output of one filter to be partially fed back to its input. The data that is fed back is merged into the input stream of the FeedbackLoop itself.

While all these constructs may seem radically different from those found in non-streaming programs, when looking at individual applications, a re-formulation in an array style turns out surprisingly natural. Let us consider the construction of Streamit-like filters first. Let us look at the FIR example from the paper [] on Streamit:

```
class FIRFilter extends Filter {
   float[] weights;
   int N;
   void init(float[] weights) {
      setInput(Float.TYPE);
      setOutput(Float.TYPE);
      setPush(1); setPop(1); setPeek(N);
      this.weights = weights;
      this.N = weights.length;
   }
   void work() {
      float sum = 0;
      for (int i=0; i<N; i++)
         sum += input.peek(i)*weights[i];
      input.pop();
      output.push(sum);
}
```

The `FIRfilter` contains two initialisation variables `weights` and `N`. `weights` is supposed to hold a vector of `N` weights of floating point type. The body of the filter definition contains two methods: `init` for initialising the filter and `work` for doing the actual computation.

The `init` method initialises the two variables and it determines how this filter interacts with the input and output streams. In particular, it defines that each invocation of the filter will consume one element from the input stream, and produce one element to the output stream: `setPop(1)` and `setPush(1)`, respectively. It also defines that the filter will need a preview on the subsequent `N` elements: `setPeek(N)`.

The actual operation of the filter is defined in the method `work`. It peeks into the next `N` elements of the input stream and computes the dot product of these values with the vector `weights`. Subsequently, it consumes the first input of the stream and it pushes the computed result into the output stream.

`FIRfilter` can be expressed in Heh directly as[1]:

```
letrec FIRfilter = λinput.λweights.
    letrec n = |weights|.[0] in
    letrec l = |input|.[0] in
    imap [l - n + 1]
         {_(iv): dotp (takeat iv n input) weights
```

We compute an output stream from input stream and an array of weights. The length of the output is $N - 1$ elements shorter than the input. The functions `dotp` and `takeat` are defined as follows:

```
letrec dotp = λa.λb.
    reduce plus 0 (imap |a| {_(iv): a.iv * b.iv)

letrec takeat = λidx.λlen.λa.
    imap [len] {_(iv): a.(plus idx iv)
```

`dotp` computes the dot product of two vectors. The function `takeat` takes a sub-segment of an array `a`. It expects 3 arguments: an offset `idx` where to select from, a number of elements `len` to select, and the vector `a` to select from. With these definitions, `FIRfilter` implements the FIR filter for arrays of any size.

Let us now demonstrate how we can arrive at the same code by providing Streamit abstractions in Heh. For simplicity of presentation let us first assume that the filter reads one input element and produces one output element at a time. The number of elements filter can peek may vary.

We define a higher-order function `si_filter` to separate the handling of input and output streams from the filter-specific work:

```
letrec si_filter = λwork.λpeek.λinput.
    letrec sh = [|input|.[0] - peek + 1] in
    imap sh {_(iv): work (takeat iv peek input)
```

The function `si_filter` takes three arguments: the work function, the number of elements in the input stream a filter will peek into, and the input stream. All this function does is to invoke the `work` function for each element of the result stream, providing it with the corresponding section of the input stream.

With this definition, we can define now the FIR filter as

```
letrec work = (dotp weights) in
letrec peek = |weights|.[0] in
si_filter work peek
```

We define the `work` function through a partial application of the `dotp` function and we compute the peek size from the length of the `weights`. The full filter then can be defined as a partial application of `si_filter` to the work function and the peek length.

Push and Pop of larger sizes When filters consume and produce more than one element, we can model this by generating two-dimensional streams which we subsequently flatten. For example, consider a filter function that produce two elements at a time. In this case the definition of `si_filter` can be adjusted such that we first create a stream of pairs and then flatten it:

[1] A complete program is available at https://github.com/ashinkarov/heh/blob/master/examples/fir.heh.

```
letrec work = λi.[1,2] in

letrec si_filer = λwork.λpeek.λinput.
    ...
    lertrec out2d = imap sh|[2] {_(iv): work ... in
    flatten out2d
```

When worker function consumes multiple elements from the output stream, we have to adjust the computation of the length of the output. For example, if a function consumes m elements at a time, the length of the output can be computed as:

```
...
    letrec sh = [(|input|.[0] - peek + 1 + (m-1)) / m] in
    imap sh {_(iv):  work (takeat [iv.[0] * m] peek input)
```

Both of these cases can be combined, and both parameters can be passed via the arguments of `si_filter`:

```
letrec si_filter = λwork.λpeek.λpush.λpop.λinput.
    letrec sh = [(|input|.[0] - peek + 1 + (pop-1)) / pop] in
    lertrec out2d = imap sh|[push] {_(iv): work (takeat [iv.[0] * pop]
                                                peek input)
    in flatten out2d
```

Remaining Streamit combinators In a similar fashion, the streaming combinators of Streamit can be defined in Heh. *Pipeline* boils down to simple function composition:

```
letrec si_pipeline = λf.λg.λinput.g (f input)
```

The *SplitJoin* combinator splits the input stream into n sub-streams and re-joins the result streams into a single stream of results. Again, the scaffolding can be expressed through a higher-order function responsible for re-shuffling the stream data as needed. Round robin splitting and joining of n sub-streams can be expressed as

```
letrec si_split_join = λfs.λinput.
   letrec n = |fs| in
   letrec m = [|input|.[0] / n] in
   letrec out2d = imap n|m {_(iv):
                            fs.iv (take_each_from n iv input)
   in flatten (transpose out2d)
```

`si_split_join` takes two arguments, the first argument `fs` is a vector of n filter functions and the second argument is the input list. In the body, we apply each filter function in `fs` to a sub-stream of the input stream. We compute these sub-streams using a function `take_each_from` which takes each nth element of the argument stream starting at offset `iv`. In the same way as in the implementation of the function `si_filter` with non-unit output, we specify a two-dimensional stream where each its row represents the result of applying the `iv`th filter function in `fs` to the corresponding sub-stream. A round-robin merge can be achieved by flattening the transposed two dimensional array of result-streams.

The *FeedbackLoop* combinator makes it possible to create cycles in a stream. This can be achieved in Heh by creating recursive imaps. In [] the motivating example for the FeedbackLoop combinator was a stream of fibonacci numbers, which can be expressed in Heh as:

```
letrec fib = imap [N] {[0] <= iv < [2]: 1,
                       [2] <= iv < [N]: fib.[iv.[0]-1]
                                        + fib.[iv.[0]-2]
```

While the definitions presented here merely sketch how a streaming language such as Streamit can be shallowly embedded into an array language such as Heh, we can observe several aspects that are instrumental for a rather straight-forward embedding:

Firstly, support for multi-dimensional infinite arrays turns out to be useful for dealing with sub-streams. Secondly, support for higher-order functions is crucial to define the compositional primitives of the streaming language. Finally, the ability to specify arrays in a recursive fashion is instrumental for supporting feedback mechanisms. All three of these aspects are reflected in the definition of our array language Heh.

4 Extending Arrays to Infinity

We present Heh in two steps. First, we define a finite strict fragment called λ_α, and then we extend λ_α with the ability to handle infinite arrays. λ_α is an idealised, data-parallel array language, based on an applied λ-calculus. The key aspect of λ_α is built-in support for shape- and rank-polymorphic array operations, similar to what is available in APL [12], J [14], or SaC [10].

In the array programming community, it is well-known [13, 8] 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.

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 hyperrectangle.* 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.

4.1 Syntax Definition and Informal Semantics of λ_α

$$
\begin{array}{lll}
c ::= & 0, 1, \ldots, & \text{(numbers)} \\
\mid & \mathit{true}, \mathit{false} & \text{(booleans)} \\
\\
e ::= & c & \text{(constants)} \\
\mid & x & \text{(variables)} \\
\mid & \lambda x.e & \text{(abstractions)} \\
\mid & e\ e & \text{(applications)} \\
\mid & \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e & \text{(conditionals)} \\
\mid & \mathit{letrec}\ x = e\ \mathit{in}\ e & \text{(recursive let)} \\
\mid & e + e, \ldots & \text{(built-in binary)} \\
\mid & [e, \ldots, e] & \text{(array constructor)} \\
\mid & e.e & \text{(selections)} \\
\mid & |e| & \text{(shape operation)} \\
\sim & &
\end{array}
\qquad
\begin{array}{lll}
\sim & & \\
\mid & \mathit{reduce}\ e\ e\ e & \text{(reduction)} \\
\mid & \mathit{imap}\ s \begin{cases} g_1 : & e_1, \\ \quad \ldots & \\ g_n : & e_n \end{cases} & \text{(index map)} \\
\\
s ::= & e & \text{(scalar imap)} \\
\mid & e|e & \text{(generic imap)} \\
g ::= & (e\ \texttt{<=}\ x\ \texttt{<}\ e) & \text{(index set)} \\
\mid & \_(x) & \text{(full index set)}
\end{array}
$$

Fig. 1. The syntax 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 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: $\left|[1,2,3,4]\right| = [4]$ and $\left|[[1,2],[3,4]]\right| = [2,2]$. DD 5 and DD 2 imply that we have:

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

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:

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

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:

[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.

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

assuming that $\left|s\right| = [n]$. 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:

$$v_1 \texttt{<=} v_2 \implies \left|v_1\right|.[0] = \left|v_2\right|.[0] \wedge \forall 0 \texttt{<=} i < \left|v_1\right|.[0] : v_1.[i] \texttt{<=} v_2.[i]$$

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:

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

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:

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

[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 $\oplus \in \{+, -, \cdots\}$.

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 [4, 2, 3]. The above *imap* expression therefore needs to be written as:

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

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.

5 Transfinite Arrays

Adding the notion of infinity in λ_α requires two adjustments: turning infinite imaps into lazy constructs, and deciding arithmetics on infinity. The first adjustment is straight-forward, the interface of the *imap* does not change, but we do not force all the elements when evaluating an *imap* expression. Instead we create a closure, and we update this closure lazily on every selection. For more details see [].

Arithmetics with infinities is a bit more subtle question. It turns out that definition of these operations have consequences on the array-algebraic low that will or will not hold. We extend λ_α with cardinal infinity ∞ in a standard way:

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

The following operations are undefined:

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

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

5.1 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

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

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 [16], Mathematics of Arrays [17], and Array Algebras [9]. 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 [8] 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 $o2i\ (i2o\ iv\ s)\ s = iv$ 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:

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

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.

5.2 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 oversimplification, 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:

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

For more details on ordinals including formal definitions, operations and their implementation can be found in [15].

5.3 Heh: Adding Ordinals to λ_α

The key contribution of this paper is the introduction of Heh, 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
$e ::= \cdots$		
$\mid$ *islim*	(limit ordinal predicate)	
$c ::= \cdots$		
$\mid$ $\omega, \omega + 1, \ldots$	(ordinals)	

Fig. 2. The syntax of Heh.

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.

5.4 Array Equalities Revisited

With the support of Ordinals in Heh, we can now revisit our equalities Eq. 1 and 2. Let us first look at the counter example for Eq. 1: from Section 5.1: With

[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).

an array shape $s = [2, \omega]$ and a legal index vector into s $[1, n]$, we now obtain:

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

The crucial difference to the situation from λ_α^∞ in Section 5.1 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 Heh, and, based on this proof, Eq. 1 can be shown as well.

6 Examples

The shift from natural numbers to ordinals as indices in Heh 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.

Transfinite tail 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 $(tail\ a).[\omega]$ should be a valid expression. For the result shape of tail a, we obtain $[\omega + 42] \dot{-} [1] = [\omega + 42]$. A selection (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 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 $[1] \dot{+} iv$ is an index expression, not $iv \dot{+} [1]$. 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

$a.([\omega + 41]\dot{+}[1])$ 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 Heh 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 Heh, 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 Heh 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 $\nwarrow$ and $\searrow$. 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[5]: 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
       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.

7 Implementation

We implement two flavours of Heh[6]:

1. an interpreter which includes ordinal-indexed arrays, and
2. a compiler for the strict and finite subset of Heh.

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[] into Ocaml code. We carefully implement updates in-place for *imap* 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 Heh is achieved by translating Heh 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 Heh programs in parallel on these types of architectures. The

[5] See this video by John Scholes for more details: https://youtu.be/a9xAKttWgP4
[6] The implementation is available on Github https://github.com/ashinkarov/heh.

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 [15]. 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.

7.1 Performance considerations

Our compiler for the strict and finite sublanguage of Heh shows that this part of the language can be mapped into languages such as SaC, leading to high-performance execution potential on variouss platforms [23, 20]. Whether the full-fledged version of Heh 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 [22] 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 [22]. 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 [5, 10]. 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 Heh is limited to ordinals up to ω^ω. For further details see [21, Sec. 4]

8 Conclusions and Future Work

This paper demonstrates a unified framework for arrays and streams. Within this framework streams are interpreted as transfinite arrays — infinite arrays indexed by ordinals. Under such an interpretation it becomes possible to write generic programs that equally operate on finite and infinite input data.

We show that within such a framework, typical streaming abstractions, similar to those found in the StreamIt language, can be encoded in a straight-forward and natural way. This gives a rise to mechanical translation of streaming applications into the proposed framework. By doing so, the same specification can be used to generate the code for finite and infinite input data. In finite case, a large body of existing work on array optimisations becomes immediately applicable.

Overall, this unification removes a typical distinction between opaque filters and filter combinators that exists in most of the streaming languages. Without such a distinction, program optimisations that change the granularity of streaming networks can be easily expressed.

The array-based nature of the proposed framework brings the built-in concept of multi-dimensional arrays in the world of streams. A number of multi-dimensional problems can be specified as streams. For example consider Ackerman function or a Game of Life problem from our examples. As ordinals are being used for indexing arrays, a number of program transformations that are based on array-algebraic laws can be applied to streams.

The concept of transfinite arrays opens up exciting perspectives for future research. As we mentioned in the implementation, it is not yet fully clear what is the best representation for infinite arrays. Current implementation provides no primitives to encode hints to a compiler that certain expressions should be streamed. Application of existing memory contraction techniques to the proposed

framework would require some extensions to handle arrays of transfinite shapes. Finally, the question on how to estimate a static space bound for the given specification requires further investigation.

References

1. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajdax, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010). pp. 169–178 (July 2010). https://doi.org/10.1109/MEMCOD.2010.5558637
2. Bernecky, R.: An introduction to function rank. ACM SIGAPL Quote Quad **18**(2), 39–43 (Dec 1987)
3. Bernecky, R.: The role of APL and J in high-performance computation. ACM SIGAPL Quote Quad **24**(1), 17–32 (Aug 1993)
4. Bernecky, R., Iverson, K.E.: Operators and enclosed arrays. In: APL Users Meeting 1980. pp. 319–331. I.P. Sharp Associates Limited, I.P. Sharp Associates Limited, Toronto, Canada (1980)
5. Cann, D.: Compilation Techniques for High Performance Applicative Computation. Tech. Rep. CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California (1989)
6. Chakravarty, M.M.T., Keller, G., Lechtchinsky, R., Pfannenstiel, W.: Nepal — Nested Data Parallelism in Haskell, pp. 524–534. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
7. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore gpus. In: Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011. pp. 3–14. ACM (2011)
8. Falster, P., Jenkins, M.: Array Theory and Nial (1999)
9. Glasgow, J.I., Jenkins, M.A.: Array theory, logic and the nial language. In: Proceedings. 1988 International Conference on Computer Languages. pp. 296–303 (Oct 1988). https://doi.org/10.1109/ICCL.1988.13077
10. Grelck, C., Scholz, S.: SAC - A functional array language for efficient multi-threaded execution. International Journal of Parallel Programming **34**(4), 383–427 (2006). https://doi.org/10.1007/s10766-006-0018-x, http://dx.doi.org/10.1007/s10766-006-0018-x
11. Henriksen, T., Oancea, C.E.: Bounds checking: An instance of hybrid analysis. In: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. pp. 88:88–88:94. ARRAY'14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2627373.2627388, http://doi.acm.org/10.1145/2627373.2627388
12. Iverson, K.E.: A Programming Language. John Wiley & Sons, Inc., New York, NY, USA (1962)
13. Jenkins, M.A., Mullin, L.R.: A Comparison of Array Theory and a Mathematics of Arrays, pp. 237–267. Springer US, Boston, MA (1991)
14. Jsoftware, I.: Jsoftware: High performance development platform. http://www.jsoftware.com/ (2016)
15. Manolios, P., Vroon, D.: Ordinal arithmetic: Algorithms and mechanization. Journal of Automated Reasoning **34**(4), 387–423 (2005).

https://doi.org/10.1007/s10817-005-9023-9, http://dx.doi.org/10.1007/s10817-005-9023-9
16. More, T.: Axioms and theorems for a theory of arrays. IBM J. Res. Dev. **17**(2), 135–175 (Mar 1973). https://doi.org/10.1147/rd.172.0135, http://dx.doi.org/10.1147/rd.172.0135
17. Mullin, L.M.R.: A Mathematics of Arrays. Ph.D. thesis, Syracuse University (1988)
18. Scholz, S.B.: Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting **13**(6), 1005–1059 (2003). https://doi.org/10.1017/S0956796802004458
19. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Proceedings of the 11th International Conference on Compiler Construction. pp. 179–196. CC '02, Springer-Verlag, London, UK, UK (2002), http://dl.acm.org/citation.cfm?id=647478.727935
20. Šinkarovs, A., Scholz, S., Bernecky, R., Douma, R., Grelck, C.: 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
21. Šinkarovs, A., Scholz: Operational semantics of lambda-omega. (2018), https://goo.gl/MeZpbr
22. Šinkarovs, A., Scholz, S.B., Stewart, R., Vießmann, H.N.: Recursive array comprehensions in a call-by-value language. (to appear). In: Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages. IFL '17 (2017)
23. Wieser, V., Grelck, C., Haslinger, P., Guo, J., Korzeniowski, F., Bernecky, R., Moser, B., Scholz, S.: 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