# Reversible Computation of Array Combinators

Victor Alexander Schmidt
rqc908@alumni.ku.dk

June 20, 2025

**Abstract**

The Futhark programming language [3] is built with parallel execution of array combinatorics in mind. In this assignment, we will take a look at how array combinators may become reversible by constructing a Futhark-like language, defining its syntax and semantics, such that the defined language is inherently reversible.

The goal will be to make a collection of as many reversible array combinators as possible, without compromising reversibility, collectively defining the constructs of a programming language. Then, using this language, it would be nice to conclusively provide some examples, of what the reversible array combinators are capable of. This would be done through program examples, for instance writing a run-length encoder equivalent to the Janus implementation, seen earlier in this course.

The task of reversing array combinators have been attempted before [6], but there is no definitively *correct* way to make constructs reversible. This assignment will be distinct in the sense, that the proposed language will be a small reversible language like Janus [5], that adopts elements from Futhark. This assignment will **not** go into detail about how the parallelism of Futhark primitives (array combinators) is achieved. We will, however, take a look at how array combinators are traditionally defined, or rather how Futhark defines them, and what changes that could make such combinatorics reversible (there may be more than one way to do so).

If time permits, after defining the proposed language, writing a prototype interpreter & reverse interpreter for this simple language in a language like Haskell would be the natural next step. Then, if there is any more time left, the remaining time could be spent on writing a compiler from the proposed language to Futhark. This would be particularly interesting, as writing both a program inverter and a compiler would allow us to compile programs and inverted programs directly to Futhark, meaning we take a small stab at creating reversible computation in Futhark.

# Contents

# 1 Introduction

In this report, we will cover parts of defining a programming language, culminating in the syntactic and semantic definition of a reversible programming language, which can call primitive array combinator functions. A collection of these array combinator functions will be presented as is, and updated such that it is reversible given an inversion rule. Together with exisiting theory on reversible language design, we will define the syntax and semantics by presenting a grammar for the language and propose a few example programs written in the resulting language.

# 2 Language Prelude

## 2.1 Language Syntax

This section will cover basic language syntax. Determining which symbols and keywords to reserve for language constructs, through observing what works in already existing reversible languages (Janus). Also, introducing notions to support the coming array combinators.

### 2.1.1 Adopting Janus Syntax

Janus is a well-defined, reversible language. To achieve success in defining a new reversible language, it is paramount to adopt reversible elements, atomics and constructs, from Janus [4, 8]. For this purpose, the following grammar can be adopted from the language of Janus, without major modification:

$$
\begin{aligned}
p &::= vdec^{*}\ (\texttt{procedure}\ id\ s)^{+} \\
vdec &::= x \mid x[c] \\
e &::= c \mid e \otimes e \mid x\texttt{[}e\texttt{]} \mid x \\
c &::= 1 \mid 2 \mid \ \ldots\ \mid 4294967295 \\
s &::= x\ \oplus\texttt{=}\ e \mid x\texttt{[}e\texttt{]}\ \oplus\texttt{=}\ e \mid x\ \texttt{<=>}\ x \mid \\
&\qquad \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{fi}\ e \mid \\
&\qquad \texttt{from}\ e\ \texttt{do}\ s\ \texttt{loop}\ s\ \texttt{until}\ e \mid \\
&\qquad \texttt{skip} \mid s\ s \mid \texttt{call}\ id \mid \texttt{uncall}\ id \\
\oplus &::= \texttt{+} \mid \texttt{-} \mid \texttt{\^{}} \\
\otimes &::= \ \oplus\ \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \\
&\qquad \texttt{\&\&} \mid \texttt{||} \mid \texttt{<} \mid \texttt{>} \mid \texttt{==} \mid \texttt{!=} \mid \\
&\qquad \texttt{<=} \mid \texttt{>=}
\end{aligned}
$$

### 2.1.2 Introducing Lambdas

In Janus, a `lambda` function makes no sense to include, as small, uncallable functions have no use. Then why is it that `lambda`s are introduced here? `lambda`s give no real *semantical* difference from functions, other than providing convenience when calling array combinators that require functional arguments. `lambda`s help in this sense, by letting the programmer write the program they wish, without having to declare multiple trivial functions, just to use the ACs in the way that they intend. As a consequences, syntax may become more complex to read, but as stated right above, the amount of boilerplate code is possibly reduced significantly.

**Defining the syntax of `lambda`:** There is no reason to reinvent the wheel, simply adopting the `lambda` syntax from Futhark will be sufficient. For those who do not remember this specific `lambda` syntax, it can be described by the following grammar extension:

$$\texttt{lambda} ::= \ \texttt{(}\backslash x^{+}\ \texttt{->}\ \texttt{c)}$$

**What are the inversion rules of `lambda`?** Again, no reason to reinvent the wheel. As `call` and `uncall` is defined in Janus, an inversed lambda is trivially like `uncalling` it, as if it were a function (which it is):

$$\mathcal{I}[\![\texttt{(\textbackslash}x^+\ \texttt{->}\ \texttt{s)}]\!] \equiv \texttt{(\textbackslash}x^+\ \texttt{->}\ \mathcal{I}[\![\texttt{s}]\!]\texttt{)}$$

**Allowing Tuples** To adopt two essential array functions (`zip` and `unzip`), tuples are allowed as a valid datatype. Tuples can be dereferenced using indexing, and generally behave semantically equivalent to arrays. Tuples can be a valid construct in this language outside of the specific functions, but we will not get into any more detail regarding this, as this is not the main point of this report.

# 3 Reversing Array Combinators

So given an array combinator, how would one `uncall` or reverse it? How can we "undo" the combinatorics? This section will answer this question for each array combinator in table 1. We will also get into exactly what each array combinator does, i.e. the semantics of a forward execution, and then state or discuss possible reversing techniques for backward execution.

## 3.1 Relevant Array Combinators

The array combinator functions can be found in the Futhark documentation [2], in either the SOACs (**S**econd-**O**rder **A**rray **C**ombinators) or array tab. The most interesting array combinators found in the language of Futhark, are the functions `map` and `scan`, among others (but particularly the aforementioned inspired the topic for this final assignment).

To cut the chase, the array combinators exhibited in table 1 are generally useful when programming in Futhark, while most of these are suitable for this project. It is then implicit that some of these combinators probably will not work, when doing reverse computation (RC).

| function | type signature $\forall a, b$ |
|---|---|
| `map` | $(a \to b) \to [a] \to [b]$ |
| `reduce` | $(a \to a \to a) \to a \to [a] \to a$ |
| `scan` | $(a \to a \to a) \to a \to [a] \to [a]$ |
| `scatter` | $[a] \to [\texttt{int}] \to [a] \to [a]$ |
| `partition` | $(a \to \texttt{bool}) \to [a] \to ([a], [a])$ |
| `filter` | $(a \to \texttt{bool}) \to [a] \to [a]$ |
| `split` | $\texttt{int} \to [a] \to [a]$ |
| `reverse` | $[a] \to [a]$ |
| `concat` | $[a] \to [a] \to [a]$ |
| `iota` | $\texttt{int} \to [a]$ |
| `rotate` | $\texttt{int} \to [a] \to [a]$ |
| `zip` | $[a] \to [b] \to [(a, b)]$ |
| `unzip` | $[(a, b)] \to ([a], [b])$ |

Table 1: Table of array combinators together with their type signatures. The table also includes typical array functions. Note that not all combinators may be reversible (e.g. `filter`), but they may be *filtered* out later in this paper. Type signatures (almost) adhere to the Futhark documentation [2].

So why array combinators? Array combinators give way for a language to modify collections of data uniformly, without having to create `loop`ing logic for each modification. Array combinators also allow for parallelization techniques and performance boost compared to sequential loops [3].

### 3.2 Reversing `map`

The `map` function is defined as applying a function `f` to each element of an array `A`. Abstractly, we can define an array applied with a `map` as:

```
forall values of type a and type b
A ::= [a,a,...,a] : [a]
f : a -> b
map f A === [f a, f a,..., f a] === [b,b,...,b] : [b]
```

Or concretely, take this simple example:

```
A ::= [1,2,3]
map (\x -> x += 1) A == [2,3,4]
```

With a simple `lambda` function `f` that adds 1 to its input location. `f` may also transform a value from one type into another; as long as the injective change is reversible (which injective changes inherently are), we can allow any injective `f`. For instance, a later definition of the function `iota` dynamically allocates an array, which may be `map`pable if we allow for irregularly nested arrays (not allowed in Futhark, due to irregular parallelism).

The reason `f` should be reversible in this case, is due to the reversible nature of a `map`: a `map` is trivially **reversed by uncalling its application function** `f` on each element, thus `map`ping the uncall. In other words, we are inversing the function `f`:

$$\mathcal{I}[\![\texttt{map f A}]\!] \equiv \texttt{map } \mathcal{I}[\![\texttt{f}]\!] \texttt{ A}$$

If we repeat the example from before but run it in the backwards direction, the inversion of `f` adding one to a value would be subtracting one:

```
map (\x -> x -= 1) A == [1,2,3]
A ::= [2,3,4]
```

### 3.3 Reversing `scan`

The `scan` combinator is traditionally called, using a combinator function `f` on an array `A` as such: `scan f A`. In Futhark, `scan` also requires a neutral element for parallelization, as seen in the type signature of `scan` in table 1, and as explained in the Futhark documentation [1, 2]. The neutral element also serves as the first argument of the application of `f` between the neutral element and the first element. We will forego this definition of `scan`, in favor of a different, more transparent `scan`: Do however note that in future language updates, this may be changed to incoorporate a neutral element to allow for parallelisation.

Now define `scan f A` when called to run `f` on each pair of `f A[i-1] A[i]`, which usually in the case of binary operator `+` (in this injective language, we use `+=`) is also called a prefix sum [1], though any two parameter function will work. In the example programs section, we will see examples of both prefix sums and other two-parameter functions. `scan` can more formally be defined as such:

$$\texttt{scan f A} \equiv \texttt{[A[0], (f A[0] A[1]), (f (f A[0] A[1]) A[2]),...]}$$

Here is a concrete example of using `scan` with an injective (binary) operator:

```
A ::= [1,2,3,4]
scan (x y -> y += x) A == [1,(1+2),((1+2)+3),(((1+2)+3)+4)]
                       == [1,3,6,10]
```

Where the parentheses can be omittet, since `+` is associative.

There is a trivial *inversion* rule, such that the `scan` combinator can be inverted for *any* two parameter function (which we will get to shortly):

$$\mathcal{I}[\![\texttt{scan f A}]\!] \equiv \texttt{rscan } \mathcal{I}[\![\texttt{f}]\!] \texttt{ A}$$
$$\mathcal{I}[\![\texttt{rscan f A}]\!] \equiv \texttt{scan } \mathcal{I}[\![\texttt{f}]\!] \texttt{ A}$$

Where this definition of `rscan` can be expressed as such:

`rscan f A` $\equiv$ `[A[0],..., (f A[n-3] A[n-2]), (f A[n-2] A[n-1])]`

Please note that the above definition of `rscan` requires parallel computation: Not exactly, but an alternative interpretation of `rscan` can be seen as a combination of array functions`map` and `rotate`, which are inherently parallelisable. The reason `rscan` is a different construct than `scan`, is that `scan` does not provide any easy inversion of itself, so we define a new construct to be its inverse. While adding new constructs for the sake of reversibility may seem cumbersome, it provides clarity; we could have defined `rscan` as `uncall scan`, however since the language does not `call scan`, the syntax should not `uncall scan` either (even though conventions could be semantically equivalent to what we just defined).

Showing that this inversion rule works on the previous example:

```
A ::= [1,3,6,10]
rscan (\x y -= y -= x) A == [1,(3-1),(6-3),(10-6)]
                         == [1,2,3,4]
```

Why does this work? As a forward directional run of the program would "accumulate" a sequential run-through of applying `f` to `A[i-1]` and `A[i]`, saving the result in `A[i+1]` and repeating this process, means that we only need to "unaccumulate" each array index. E.g. in the case for `f := (\x y -> y += x)`, we have: `A[i] += (A[0] + A[1] + ... + A[i-1])`. The trick to restore `A[i]` is to inverse the `'+='` operation, and it is known that the previous address of the array must contain this exact accumulated value, so without having to define some complex behaviour, we can restore `A[i]` with `A[i] -= A[i-1]` in this example.

**But does this apply to any function `f`?** Take a look at Futharks definition of the type signature of function `f` passed to `scan`, also referenced in table 1:

$$\texttt{f} : \forall a.a \rightarrow a \rightarrow a$$

The function expects two input values, to produce a single output value. Reversible functions would rather return a store, in which the input and output variables are both modified. If we define `f` to be purely injective, then any change to `x` and `y` in some `s` where `f := (\x y -> s)`, will lead to a reversible function application. This does however lead to some consequences, which will be discussed in a moment. The following type signature might then be more appropriate:

$$\texttt{f} : \forall a.(a, a) \rightarrow (a, a) \text{ or } a \times a \rightarrow a \times a$$

Which is coincidentally the type signature Mogensen arrived at [6]. This type signature says that `f` expects two input values to create two output values. In the case of `+=` where there is only one output value, we can say that the function takes `(a,b)` and outputs `(a,c)`, where `c = a + b`. Semantically we can allow the function `f` to act in a weird way. As `f` takes the input `A[i-1]` and `A[i]`, we traditionally only expect it to modify the index of `A[i]` in this call of `f`. It is however possible to allow for `f` to *also* modify `A[i]` in the same call, for instance allowing a function such as `(\x y -> x += 2 y += x)`. This would still be a reversible and valid function, however this is assumed to destroy any possible parallelism that `scan` may promise, as such functions are almost guaranteed non-associative.

So semantically, there is now two possible interpretations of `scan`:

1. To preserve parallelisation, preserve associativity. This means that **updates to the first parameter of `f` is either not allowed or greatly discouraged**. In concrete terms, running `f` with input `(a,b)` outputs `(a,c)` for any `f, a, b,` and `c`, is a requirement for `scan`.
2. Allow any `f : (a -> a) -> (a -> a)`.

A parallelisation enthusiast, programming language designer would surely advocate for the former. The latter leaves more freedom with the programmer. Due to time restrictions, the latter is easier to implement and is therefore also the conclusion. However even if any function is allowed, an interpreter/compiler can still optimize functions that maintain associativity.

### 3.4 Reversing `reduce`

`reduce` is a tricky case. In Futhark, `reduce ne op A` behaves like a scan, but only returns the accumulated value when `op` is applied to each element of `A`. `ne` acts as the neutral element or the initial accumulator value.

Earlier when writing this report, `reduce f A` was considered the best reversible identity of `reduce`, which only allowed `reduce` to act as a constant in expressions, in other words disallowing reversibility (injectivity) altogether for this construct. However it turns out that the injective nature of our language, allows us to do something cooler and more reasonable with `reduce`. Consider this interpretation:

$$\texttt{reduce x f A} \equiv \texttt{f (... (f (f x A[0]) A[1]) ...) A[n]}$$

Or more precisely described as sequential statements semantically equivalent to `reduce`:

```
reduce x f A == f x A[0]
                f x A[1]
                ...
                f x A[n]
```

Where the programmer can make updates to the variable `x`, through the injective function `f`; do however note that `f` here also has a reference to an array index, which may also be updated through injective statements in `f`. This is a side effect of passing variable locations with `f`, and has the interesting semantic property that while a `map` can not update an array with an out-of-scope variable, a reduce `reduce` can pass a variable `x`, giving multiple uses for this interpretation of `reduce`. Here are the expected semantic behaviour of `reduce`:

```
-- Note: In the below lambda:
--       a represents the accumulator value x,
--       b is the iteration-based, indexed value of the array A
reduce x (\a b -> a += b) A ==
    x += A[0]
    x += A[1]
    ...
    x += A[n]
--  x == x + A[0] + A[1] + ... + A[n]
```

We should also cover a way to reverse `reduce`: As you have probably guessed, the above sequence of statements is trivially reversible. Simply apply Janus' inversion rules, and we have the inversed `reduce`. In a somewhat funny twist of fate, we can completely avoid defining a reverse construct for `reduce`, unlike we did for `scan` with `rscan`:

$$\mathcal{I}[\![\texttt{reduce x f A}]\!] \equiv \texttt{reduce x } \mathcal{I}[\![\texttt{f}]\!] \texttt{ (reverse A)}$$

To describe this inversion, we are simply applying the inverse function, in the reverse order of application. This works, since doubly inverting the construct reverses `A` twice and inverts `f` twice, and double inversions is the same as no inversions at all (holds true in this context). However there is a catch - due to array combinators being completely injective (each call updates the store), reversing `A` in the context of application unfortunately applies the `reverse A` to the location of `A` in the store. Hence we can not have the inversion rule for `reduce` using `reverse A`, as this interferes with the rest of the program, resulting in an incorrect computation, as `A` is reversed one too many times, if the program contains a `reduce` call and is run backwards.

Going back on our word of not introducing a new construct, we define `rreduce x f A`:

$$\mathcal{I}[\![\texttt{reduce x f A}]\!] \equiv \texttt{rreduce x } \mathcal{I}[\![\texttt{f}]\!] \texttt{ A}$$
$$\mathcal{I}[\![\texttt{rreduce x f A}]\!] \equiv \texttt{reduce x } \mathcal{I}[\![\texttt{f}]\!] \texttt{ A}$$

Where `rreduce` is semantically equivalent to:

```
rreduce x f A ==
  reverse A
  reduce x f A
  reverse A
```

`rreduce` properly reverses the array `A`, performs the `reduce`, and returns `A` to its original order by re-reversing. Or at least it does something that is semantically equivalent (it may just run through backwards, without having to reverse input array `A`).

### 3.5 Reversing `scatter`

As `scatter` is inherently destructive, the primitive is irreversible in the defined state. However, there may be ways to make this array combinator reversible.

First, `scatter` is called by `scatter X I A` [2], "shooting" each value of `A` using its cornensponding indicies from `I`, to overwrite elements in `X` as such:

```
scatter [0,0,0,0,0] [1,3,7] [4,5,6] == [0,4,0,5,0]
```

Okay, but how do we make this reversible? Simple; instead of "shooting" values from `A` into `X`, we may define some change that is injective instead of desctructive.

Take for instance the swap operator `'<=>'`. If we define the `scatter` function in terms of swapping values of `X` and `A`, all of a sudden `scatter` becomes reversible. This also works for other binary operators, and as in the case of `scan`, can work for *any* two parameter function.

Now, we can define a reversible `scatter` call as `scatter f X I A`. The inversion rule is also trivially reversed:

$$\mathcal{I}[\![\texttt{scatter f X I A}]\!] \equiv \texttt{scatter } \mathcal{I}[\![\texttt{f}]\!] \texttt{ X I A}$$

`scatter` will in this case work as a pointwise `map`, call it an *injective*-`scatter`. The input function *could* be predefined, such that we for instance only allow for the use of the swap operator, however it provides more freedom to the programmer if a function should be passed to the construct. There are many ways to define a reversible compromise of this construct.

### 3.6 Reversing `partition`

`partition p A` partitions an array `A`, separating elements that succeed and fail under the predicate `p` into different arrays. The result is then two arrays `(B,C)`.

`partition` is not inherently reversible, as after the separation of elements under the predicate, the information as to which is the original indicies of succeeding and failing elements is lost.

To reverse `partition`, the result also needs to preserve some kind of indexing, which is required to recover the original `A`. However it turns out that we can create a `partition p A` function, from our previously established array combinators and other functions. This will come later in the report, in the example programs.

### 3.7 Reversing `filter`

Unfortunately, this is impossible. The correct way to reverse a `filter`, is to also keep track of elements that fail the predicate: This is already defined as `partition`. So `filter` has no chance of having an original definition, while being reversible, meaning it is redundant or considered syntactic sugar at best.

### 3.8 Reversing `iota`

`iota n` creates an array, given a size `n`, with element values corresponding to its index position as integers, i.e. `(iota n)[i] == i`.

Adopting the `atoi` function directly from Mogensen [6], the behaviour can be defined as:

```
iota n == [0,1,...,n-1]
atoi [0,1,...,n-1] == n
```

Where `iota` and `atoi` are direct inverses of each other, i.e. they have the inversion rules:

$$\mathcal{I}[\![\texttt{iota n}]\!] \equiv \texttt{atoi [0,1,...,n-1]}$$
$$\mathcal{I}[\![\texttt{atoi [0,1,..,n-1]}]\!] \equiv \texttt{iota n}$$

### 3.9 Trivial Inversions

#### 3.9.1 `concat` and `split`

`concat A B` concatenate arrays `A` and `B`, while `split k A` splits array `A` into two arrays between index `k` and `k+1`. These two are each others' inverse and will be defined as described by Mogensen [6]. Note that this means that `concat A B` returns both the concatenated array and at which index `A` stops and `B` starts, as to make `split` the proper inverse of `concat`. To make the inversion consistent between forwards and backwards execution, we define the following:

```
for array A with length i and array B with length j:
concat A B => A := i-1; B := [A[0],...,A[i-1],B[0],...,B[j]]

for integer x and array C with length h:
split x C  => x := [C[0],...,C[x]]; C := [C[x+1],...,C[h-1]]
```

This leads to the trivial inversion rules:

$$\mathcal{I}[\![\texttt{concat A B}]\!] \equiv \texttt{split x C}$$
$$\mathcal{I}[\![\texttt{split x C}]\!] \equiv \texttt{concat A B}$$

#### 3.9.2 `zip` and `unzip`

The trick in `zip` and `unzip` lies not in its trivial definition, but in its semantics: Given only an injective statement `zip A B` without any assignment operator `:=`, which conventions do we define for `zip A B`, and by extension which conventions do we define for `unzip A B` (foreshadowing, `unzip` takes two parameters)?

The conclusion may not be elegant, but it is simple and trivially reversible: Let `zip A B` zip together `A` and `B`, and overwrite each index of `A`. In more concrete terms:

```
forall i in [0..n], where n is the length of A:
zip A B      =>    A[i] := (A[i],B[i]);    B := 0
```

This means that the store stores the zipped result in its first argument, while consuming the second argument (i.e. set the second argument to a default value like 0). Very alike `zip`, `unzip` then *unpacks* the first argument, into `A` and `B` like so:

```
forall i in [0..n], where n is the length of A:
unzip A B    =>    A[i] := A[0][i];   B := A[1][i]
```

As a result, in most contexts `zip` is used in a sequence of applications, where the sequence ends with an `unzip`.

**But wait a minute...** does that not make `unzip` destructive in its second argument? Yes, indeed it does. We can then enforce the constraint on the second argument of `unzip`, that it *must* contain the "default value" that `zip` leaves in the second variable. In this case, if `B == 0` and `A` contains an array of tuples, then `unzip A B` is a valid operation; otherwise it is undefined.

#### 3.9.3 `rotate`

`rotate x A` shifts each index of `A`, `x` indicies to the left. A negative rotation is also supported [2], and a negative rotation is trivially the inverse of a rotation:

$$\mathcal{I}[\![\texttt{rotate x A}]\!] \equiv \texttt{rotate (-x) A}$$

#### 3.9.4 `reverse`

`reverse A` reverses the ordering of elements in array `A`. `reverse` is its own inverse.

### 3.10 Syntactic Sugar

Some extra array functions can be defined together with their trivial inversions:

- `zeros` and `ones`; given integer input `x` returns an array of length `x` of elements all zeros or ones (trivially syntactic sugar of `iota` and each other, given `map` as a language construct). These will be useful for dynamically allocating space for arrays on the fly. Their inversion rules are ambiguous to the one for `iota`, and we can define that e.g. `zeros [0,0,0] == 3`, i.e. given an array of zeroes, the function acts as its own inversion, returning an integer instead (ambiguous for `ones`). Instead of overloading the functions, one could also introduce new reversed constructs as we did for `iota` giving us `seroz` and `seno`, though this is a matter of taste.
- `copy` and `uncopy`; `copy A B` copies the array at location `B` into location `A`, assuming `A` contains some default value, say 0. This is syntactic sugar of previously defined functions, funnily enough the just defined `zeros`, `zip`, `map (+=)`, and `unzip`. `uncopy A B` trivially returns `A` to 0, though `A` and `B` must be identical.
- Utility functions found in functional programming languages, such as: `len`, `head`, `last`, and so on. These can be considered constants, and therefore require no inversion rules.

## 4 Language Definition

Now we are ready to define a language, using some of Janus' syntax and semantics, while including the reversible array combinators, which we call Arc from now on (abbreviation for **A**rray **R**eversible **C**ombinatorics). We present the syntax through a final grammar, and discuss semantic conventions within the language of Arc.

### 4.1 Semantic Conventions in Arc

We intent to discuss some of the syntactic and semantic changes that differentiates Arc from Janus, which will play a huge role in tackling the addition of array combinator functions.

This section is somewhat speculative, feel free to disagree with any proposed changes.

#### 4.1.1 Utilization of Uninitialized Variables

In Janus, program conventions require the programmer to declare all variables before the intended program is run. For instance take the fibonacci program from the Janus-Playground [7] as an example:

```
procedure main()
  int x1
  int x2
  int n
  n += 4
  call fib(x1, x2, n)
```

x1, x2 and n are here initialized to 0 and then n is assigned to 4, by using the injective += operator.

This is juxtaposed by Futhark's (and other functional languages') dynamic variable assignment, where type inference, combined with **let**-assignments, easily allows for the introduction of new variables. This is however not as easy in Janus, as all variable declarations start at a default value, and arrays with a static size, to enable consistent reversibility.

The change we are attempting to make, is to make all uninitialized variable act as the integer 0 in all contexts. If an uninitialized variable is injected a value, it is entered into the store (i.e. 0 is used in its place, and if that variable is subject to an injective change, only then it is entered into the store), and by definition becomes initialized. Once a variable returns to 0 within the store through a command, we can remove it from the store, thereby making it uninitialized again, though the variable still silently represents 0. Arc should still allow for variable declarations, to allow for initialization of, for instance, statically sized arrays, however Arc directly allows for dynamic array allocations by supporting array functions iota, zeros, and ones.

### 4.1.2 Injective Function Application Using Array Combinators

One significant syntactical and semantical issue is, how one would call the array combinator functions. Mogensen [6] does it as such, in the first line of the Inner Product program:

```
(Xs:  [int], Prods:  [int]) := unzip (map ** (zip (Xs, Ys)));
```

This works great in the context of input consumption, though this can be more difficult to understand when executing the program in the reverse direction, though it is at times a more elegant solution. Agni [6] represents programs as a more "straight line", if you will. Arc is based on Janus, which is injective at its core. Some combinator function f applied in the context of an array combinator, may benefit from the innate injectivity that Janus provides syntactically and semantically. Consequently, not defining array combinators as injective statements, goes against the existing definitions in Janus, as for instance lambda function may differ substantually from the way functions are defined already, if Arc borrows its core functionality from Janus, which it does. Paradoxically, defining semantics in Arc may seem like a predicament, but adhering to the semantic and injective style already defined by Janus, injective array combinator statements now seem inevitable. Like the statement a += 1 in Janus that adds 1 to the variable a, map (\x -> x += 1) a for every i in the length of a, conducts the injective change a[i] += 1. Array combinators can be seen as a sequence of (parallelisable) Janus statements.

**What about constant array input, when f should be injective?** There are two possibilities, if the previously stated changes are assumed to pertain: 1; allow for modifcation of constant array input, resulting in the allowance of non-injetive input f, or 2; disallow constant input of array combinators altogether.

Whilst the latter is what the interpreter will go with due to time constraints, a proper type-system would allow for both injective and constant function inputs, making the former the correct approach, i.e. allowing both map f1 A and map f2 [1,2,3] to be evaluated (though for different contextual applications), where f1 is injective and f2 is not.

### 4.1.3 Function Type Signatures and Parameters

Functions in Arc should adhere closely to the function calling/uncalling conventions in Janus, as the rest of the language is quite similar, especially in relation to the injective changes compared to other languages like Futhark or Agni.

The type signature of a function in Arc is expected to be

$$f : A \to A$$

Where $A$ is *any* valid set of values.

**Which conventions should Arc use, for passing a store to a function?** There are two possibilities for passing stores to a function:

1. Pass an empty store, only with bound parameter variables. This would force functions to complete their computations within the scope of its parameters, which adheres to the just before defined function ascriptions of all functions in Arc. Functions in this context should only return stores with modified parameter values, no other values should be modified, as to uphold total atomicity.

2. Pass the entire store, binding select input variables to parameter variables. This allows for a global scope inside of called functions, which results in less atomicity/modularity and more coupling. However, this semantic convention allows for some convenience, when using certain array combinators, as we will get into at a moment. Notice that choosing to pass the entire store can be exceptionally dangerous for larger programs, as function calls may modify already existing values, which in the best case leads to unexpected results and in the worst case to type errors and undefined behaviour.

Thus we choose to pass an empty store. Sometimes this makes programming harder in Arc, but passing empty stores, makes it so the programmer need not worry about the overlapping of variables. A benefit of passing a contextless store, may be the modularity of programs being trivially easy to implement. An interesting compromise is theorized to lie here, allowing for lambda applications in array combinators to access the store of the calling function, to allow for easier indexing (sometimes this is useful in Futhark, but Futhark does not have the restrictions of reversibility on its store - Futhark uses let-bindings). To keep store passing consistent, *always* pass an empty store, besides parameters of course.

## 4.2 The Final Grammar

To define a final syntax for Arc, this following grammar incoorporates a modified Janus grammar 2.1.1, and the reversed array combinator functions:

```
Prog    ::= FunDef
FunDef  ::= FunDec FunDef
          | FunDec
FunDec  ::= fun FID FPar SExp1
FPar    ::= Var FPar
          | Var
SExp1   ::= SExp2 SExp1
          | SExp2
SExp2   ::= Var Op1 = Exp1
          | Var[Exp1] Op1 = Exp1
          | Var <=> Var
          | if Assert1 then SExp1 else SExp1 fi Assert1
          | from Assert1 do SExp1 loop SExp1 until Assert1
          | skip
          | CallFun
          | ArrComb
Assert1 ::= Assert2 && Assert1
          | Assert2 || Assert1
          | Assert2
Assert2 ::= Exp1 == Exp1
          | Exp1 <  Exp1
          | Exp1 >  Exp1
          | Exp1 <= Exp1
          | Exp1 >= Exp1
```

```
          | Exp1 != Exp1
Exp1    ::= Exp2 Op2 Exp1
          | Exp2
Exp2    ::= ConstInt
          | Var
          | Var[Exp1]
          | len Var
          | first Var
          | last Var
Op1     ::= + | - | ^ | * | /
Op2     ::= Op1 | % | & | |
CallFun ::= call FID FPar
          | uncall FID FPar
          | (\ FPar -> SExp1 ) FPar
ArrComb ::= map CallFun
          | scan CallFun
          | rscan CallFun
          | reduce Var CallFun
          | rreduce Var CallFun
          | scatter CallFun
          | iota Exp2
          | atoi Var
          | rotate Exp1 Var
          | reverse Var
          | concat Var Var
          | split Var Var
          | ones Var
          | seno Var
          | zeros Var
          | sorez Var
          | copy Var Var
          | uncopy Var Var
```

Assume that `ConstInt` is an unsigned 32-bit integer and `Var` refers to a string, that refers to a location in the store.

# 5 Example Programs

## 5.1 Factorial

A fun, small, trivial program would be a program for calculating the factorial of a variable. Such a program is well suited for Arc, as it uses of the innate behaviour of `iota`, `map`, and `scan` to calculate a factorial using basic lambdas and arrays:

```
fun factorial x y             -- expected input: {x > 0, y = 0}
  iota x                      -- [0,1,...,x-1]
  map (\a -> a += 1) x        -- [1,2,...,x]
  y += 1                      -- avoid multiplication by 0 (assumed y = 0)
  reduce y (\a b -> a *= b) x -- y = y * 1 * 2 * ... * x = x! (if y = 1)
  map (\a -> a -= 1) x        -- undo map
  atoi x                      -- undo iota
  -- output = {x = x, y = x!}
```

Note that `y` is explicitly assumed to be 0, and that the input variable `x` is not consumed. Another way to write `factorial`, assuming you want all values of factorial up to and including `x`, while consuming `x`:

```
fun factorial x
  iota x                      -- [0,1,...,x-1]
```

```
  map (\a -> a += 1) x         -- [1,2,...,x]
  scan (\a b -> b *= a) x      -- [1,2,6,24,...,(x-1)!,x!]
  -- output = {x = [1,2,6,24,...,(x-1)!,x!]}
```

This approach allows for a single input/output pair; we only modify the location bound to `x`, but we are required to store the computed values in an array, as it contains the necessary information for the reverse computation.

## 5.2 Run-length Encoder

One of the original goals of this project was to use reversible array combinators to implement a run-length encoder, in order to highlight some of the differences between Janus and Arc. Due to report length constraints, the Janus encoder will not be shown here. The two `encoder`s will not be semantically equivalent. Consider the following Arc `encoder`:

```
fun encode text code
  -- step 1. create array that masks each sequnce using an index
  begs += len text
  zeros begs
  zip text begs
  scan (\x y ->
    if x[0] == y[0]
      then skip
      else y[1] += 1
    fi x[0] == y[0]) text
  unzip text begs
  scan (\x y -> y += x) begs

  -- step 2. create the encode as a zip-pair of (element,length)
  code += (last begs) + 1
  zeros code
  copy lengths code            -- for later use
  scatter (\x y -> x += y) code begs text
  scatter (\x y -> x += 1) lengths begs text
  zip code lengths
  map (\x -> x[0] /= x[1]) code
  unzip code lengths

  -- step 3. clean up `text`
  scatter (\x y -> y -= x) code begs text
  sorez text
  reduce text (\acc x -> acc -= x) lengths

  -- step 4. clean up `begs`
  rscan (\x y -> y -= x) begs
  scan (\x y -> y += x) lengths
  scatter (\x y -> x -= 1) begs lengths code
  rscan (\x y -> y -= x) lengths
  sorez begs
  reduce begs (\acc x -> acc -= x) lengths
  zip code lengths
```

To alleviate some doubt of correctness from you, the reader, and I, the author, here is a high-level run-through of the store changes of the Arc `encoder`; we use the example input from Assignment 2 (note that `i` and `j` from the Janus `encoder` are omittet, as they were used for indexing, however Arc using uniform array modifcation has no use for these indexing variables):

```
----
-- input store:
-- text := [7,7,7,3,3,5,5,5,5,0]
-- code := 0
----
-- step 1.
begs := 10
begs := [0,0,0,0,0,0,0,0,0,0]
text := [(7,0),(7,0),(7,0),(3,0),(3,0),(5,0),(5,0),(5,0),(5,0),(0,0)], begs := 0
text := [(7,0),(7,0),(7,0),(3,1),(3,0),(5,1),(5,0),(5,0),(5,0),(0,1)]
text := [7,7,7,3,3,5,5,5,5,0], begs := [0,0,0,1,0,1,0,0,0,1]
begs := [0,0,0,1,1,2,2,2,2,3]
-- step 2.
code := 4
code := [0,0,0,0]
lengths := [0,0,0,0]
code := [21,6,20,0], text := [7,7,7,3,3,5,5,5,5,0]
lengths := [3,2,4,1], text := [7,7,7,3,3,5,5,5,5,0]
code := [(21,3),(6,2),(20,4),(0,1)], lengths := 0
code := [(7,3),(3,2),(5,4),(0,1)], lengths := 0
code := [7,3,5,0], lengths := [3,2,4,1]
-- step 3.
code := [7,3,5,0], text := [0,0,0,0,0,0,0,0,0,0]
text := 10
text := 0
-- step 4.
begs := [0,0,0,1,0,1,0,0,0,1]
lengths := [3,5,9,10]
begs := [0,0,0,0,0,0,0,0,0,0], code := [7,3,5,0]
lengths := [3,2,4,1]
begs := 10
begs := 0
code := [(7,3),(3,2),(5,4),(0,1)], lengths := 0
----
-- output store:
-- text := 0
-- code := [(7,3),(3,2),(5,4),(0,1)]
----
```

Recall that variables bound to 0 is removed from the store, thus we do not need to specify them in the input/output store; the parameter variables bound to 0 is shown for clarity.

Notice that our result from the Arc `encoder` is a lot neater than the Janus `encoder`'s result, which was:

```
text[10] = [0,0,0,0,0,0,0,0,0,0]
code[20] = [7,3,3,2,5,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
i = 0
j = 0
```

However, it is painstakenly clear that the Arc `encoder` requires a bunch more work (constant operations) than the Janus `encoder`. Even if parallelism was employable in Arc, it is doubtful that the Arc implementation would outperform the Janus implementation, even if given input arrays of great sizes, which is where the parallelizable SOACs gain the most performance, compared to sequential implementations. Though, if you want to be technical, the Arc `encoder` has a span of $O(\ln n)$, which is better than the span of the Janus `encoder` (which is $O(n)$).

### 5.3 Partition

```
fun partition mask succ arr
  reduce a (\acc x ->
    if x == 1
      then x += acc
           acc += 1
      else skip
    fi x == acc) mask
  reduce a (\acc x ->
    if x == 0
      then x += acc + 1
           acc += 1
      else skip
    fi x == acc) mask
  map (\x -> x -= 1) mask
  idx += len arr
  iota idx
  tmp += len arr
  zeros tmp
  scatter (\x y -> x <=> y) tmp mask arr
  scatter (\x y -> x <=> y) arr idx tmp
  scatter (\x y -> x <=> y) tmp mask idx
  zip arr tmp

  -- undo tmp (which has been scattered to idx)
  sorez idx
  idx -= len arr

  -- restore mask and separate arr using ‘split‘
  rreduce a (\acc x ->
    if x == acc
      then acc -= 1
           x -= acc + 1
      else skip
    fi x == 0) mask
  succ += a - 1
  split succ arr
  rreduce a (\acc x ->
    if x == acc
      then acc -= 1
           x -= acc
      else skip
    fi x == 1) mask

  -- consume mask through mechanical elemination
  unzip succ idx
  scatter (\x y -> x -= 1) mask idx idx
  zip succ idx
  sorez mask
  mask -= (len succ) + (len arr)
```

This `partiton` interpretation takes the following input: `mask`, which indicates a predicate `p` run on the input array, where element positions of `arr` that passes under the `p` equals 1 in `mask`, and failing element positions equals 0 in `mask`. `succ` is 0 as input and is the return location of succeeding elements under `p`, zipped together with the *element's original index in* `arr`. `arr` is the array to partition as input, and is the return location the failing elements at function termination, zipped together with each element's original index.

Consider this valid input/output pair for `partition`:

```
input store:
mask := [1,0,0,1,0,0,1]
arr  := [2,3,3,1,3,3,0] -- mask here represents ‘less than 3‘
succ := 0

output store:
mask := 0
arr  := [(3,1),(3,2),(3,4),(3,5)]
succ := [(2,0),(1,3),(0,6)]
```

As stated in the section on reversing `partition`, `partition` is required to keep track of the original indexing, to know in which array locations the elements succeeding and failing the predicate are originally located. This information can not be destroyed using a valid program sequence, no matter how hard the programmer tries; and if the programmer *could* destroy this information, backwards execution would fail to deterministically create valid input, meaning the language is not completely reversible. Meaning, if in Arc we could destroy this index information in some way, the langauge itself is syntactically and/or semantically flawed, as some construct within the language does not properly uphold reversibility, thus making Arc an ordinary programming language and rather than a reversible one.

# 6 Conclusion

We have presented reversible versions of array combinators and their inversion rules. We have defined a reversible programming language Arc, achieving reversibility through injetivity in the introduced constructs. Arc introduces reversible array combinators as primitives, compromising some of the traditional functionality to ensure reversibility, as seen in the introduced injective function in the definition of `scatter` and the general expected injectivity of passed functions.

# 7 Future Work

There are some improvements to be made to the language, by streamlining construct calls through updating the grammar (as seen in the Reflection section), and defining trivial semantics for non-injective combinator calls.

This project *really* wanted to include an interpreter for Arc written in Haskell, however time ran out and the interpreter remains in its infancy; rather, it is incomplete. In the future, finishing the interpreter and validating the proposed example programs is the natural next step for this project.

# References


[1] DIKU. Futhark Examples: Scan-Reduce. https://futhark-lang.org/examples/scan-reduce.html. Accessed: 2025-05-30.

[2] DIKU. Futhark Library Documentation. https://futhark-lang.org/docs/prelude/. Accessed: 2025-05-27.

[3] DIKU. Why Futhark? https://futhark-lang.org/. Accessed: 2025-05-14.

[4] Robert Glück and Tetsuo Yokoyama. Program Inversion and Reversible Computation. PAT lecture slides (04RC). Accessed: 2025-06-05.

[5] Robert Glück and Tetsuo Yokoyama. Reversible computing from a programming language perspective. *Theoretical Computer Science*, 953:113429, 2023.

[6] Torben Ægidius Mogensen. Reversible functional array programming. In Shigeru Yamashita and Tetsuo Yokoyama, editors, *Reversible Computation*, pages 45–63, Cham, 2021. Springer International Publishing.

[7] DIKU TOPPS. Janus Extended Playground. https://topps.diku.dk/pirc/?id=janusP. Accessed: 2025-06-14.

[8] Wikipedia. Janus (time-reversible computing programming language) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Janus%20(time-reversible%20computing%20programming%20language)&oldid=1239894493, 2025. [Online; accessed 05-June-2025].

[9] Wikipedia. Prefix sum — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Prefix%20sum&oldid=1295367725, 2025. [Online; accessed 13-June-2025].