Reversible Functional Array Programming

Torben Ægidius Mogensen(✉)

DIKU, University of Copenhagen, Universitetsparken 5,
2100 Copenhagen O, Denmark
torbenm@di.ku.dk

Abstract. Functional array programming is a style of programming that enables massive parallelism through use of combinators (such as map and reduce) that apply functions to whole arrays. These can be readily parallelised when the functions these combinators are applied to are pure and, in some cases, also associative.

We introduce reversible variants of well-known array combinators and show how these can be implemented in parallel using only reversible operations and without accumulating garbage.

We introduce a simple reversible functional array programming language, Agni, and show some examples of use.

1 Introduction

The world of computing is becoming more and more parallel. This is seen in the exploding number of cores in general-purpose processors but even more clearly in the increasing use of highly parallel vector processors such as graphics processors. Graphics processors are known to be power hungry, so the potentially much lower energy requirements of reversible logic could be a way of reducing power use of highly parallel programming. Reversibility adds extra constraints to programming, but graphics processors already have significant constraints to their programming, so users may be more willing to accept the constraints of reversible programming in this setting than for general-purpose programming. Languages, such as Futhark [2], are being developed to provide a machine-independent high-level abstraction on top of graphics processors without significantly impacting performance, often through functional array programming.

Functional array programming typically uses a predefined set of parallelisable combinators such as map and reduce. A typical set includes combinators like those shown in Fig. 1. You can combine these to make more complex parallel functions, and a compiler can use fusion to optimise nested combinators to reduce the overhead and exploit parallelism better than if the combinators are applied one at a time.

The combinators shown in Fig. 1 are not all reversible, nor are the functions that are passed to map, filter, scan, or reduce typically reversible, so we need to modify these to a reversible setting, and we need to create a reversible language

S. Yamashita and T. Yokoyama (Eds.): RC 2021, LNCS 12805, pp. 45–63, 2021.
https://doi.org/10.1007/978-3-030-79837-6_3

`map` : $\forall('a, 'b).('a \to 'b) \to ['a] \to ['b]$ If $ys = \texttt{map}\, f\, xs$, then $ys[i] = f(xs[i])$.
`filter` : $\forall'a.('a \to \texttt{bool}) \to ['a] \to ['a]$ If $ys = \texttt{filter}\, f\, xs$, then ys contains exactly the elements $xs[i]$ where $f(xs[i]) = \texttt{true}$ in the same order that these elements appear in xs.
`reduce` : $\forall'a.('a \times 'a \to 'a) \to ['a] \to 'a$ `reduce` f [] is undefined, `reduce` $f\,[x] = x$ `reduce` $f\,[x_0, x_1, \ldots, x_n] = f(x_0, f(x_1, f(\ldots, x_n)))$, if $n > 0$. If f is associative, `reduce` can be parallelised. If f has a neutral element i, `reduce` f [] can be defined to be equal to i.
`scanl` : $\forall'a.('a \times 'a \to 'a) \to ['a] \to ['a]$ If $ys = \texttt{scanl}\, f\, xs$, then $ys[0] = xs[0]$ and $ys[i+1] = f(ys[i], xs[i+1])$. If f is associative, `scanl` can be parallelised.
`zip` : $\forall('a, 'b, m).['a]^m \times ['b]^m \to ['a \times 'b]^m$ If $zs = \texttt{zip}\,(xs\, ys)$, then $zs[i] = (xs[i], ys[i])$.
`unzip` : $\forall('a, 'b).['a \times 'b] \to ['a] \times ['b]$ If $(ys, zs) = \texttt{unzip}\, xs$, then $(ys[i], zs[i]) = xs[i]$.
`iota` : $\texttt{int} \to [int]$ If $ys = \texttt{iota}\, m$, then $ys[i] = i$ for $0 \leq i < m$.

Fig. 1. Typical array combinators

that can use the modified combinators. We also need to argue that this language can realistically be implemented on future reversible computers that combine reversible general-purpose processors and reversible vector processors.

2 Modifying for Reversibility

We will indicate reversible functions by using an alternative function-space arrow: $\rightleftharpoons$. Combinators like `map` are not fully reversible – you can't get both a function and a list back by applying `map` in reverse, but a partial application of `map` to a reversible function is reversible. So we will use the following type signature for `map`:

$$\texttt{map} : \forall('a, 'b).('a \rightleftharpoons 'b) \to ['a] \rightleftharpoons ['b]$$

The "normal" function arrow $\to$ indicates an irreversible function space where $\rightleftharpoons$ indicates a reversible function space. $['a]$ indicates an array with elements of type $'a$, which is a type variable. To run `map` backwards, we need to supply both a function of type $a \rightleftharpoons b$ and an array of type $[b]$ for some types a and b.

That reversible languages contain irreversible elements should not be surprising: Janus [4] allows arbitrary irreversible expressions in reversible updates, for example `x += y mod 2` is allowed in Janus, even though the expression `y mod 2` is not reversible: There is no way to get from the result (0 or 1) to the value of `y`. This is allowed in Janus because, after the update to `x` (which is reversible), the

expression can be "uncomputed". Generally, if we retain the values of all variables in an expression, we can reversibly compute the value of the expression, use the value in a reversible operation, and then uncompute the expression, leaving no net garbage. We will exploit this to a larger degree than in Janus: We will allow local variables and functions to be defined using irreversible expressions, as long as these can be uncomputed at the end of their scope.

Other array combinators are clearly reversible: `zip` and `unzip` are inverses of each other, and `iota` is inverted by a function that takes an array $[0, 1, \ldots, m-1]$ and returns m (and is undefined on inputs that do not have this form). Equally obviously, `filter` and `reduce` are not reversible: `filter` discards any number of elements, and `reduce` can, for example, reduce an array of numbers to their sum or their maximum, which throws away a lot of information about the original array. Also, `reduce` and `scanl` use functions of type $'a \times 'a \to 'a$, which are not usually reversible.

We first take a stab at `scanl`, modifying its type to $('a \times 'a \rightleftharpoons 'a \times 'a) \to ['a] \rightleftharpoons ['a]$, so it takes a reversible function as argument. We then define

$$\begin{array}{ll}
\texttt{scanl}\ f\ [] & = [] \\
\texttt{scanl}\ f\ [x] & = [x] \\
\texttt{scanl}\ f\ ([x_1, x_2]@xs) & = [y_1]@(\texttt{scanl}\ f\ ([y_2]@xs)) \quad \texttt{where}\ (y_1, y_2) = f\ (x_1, x_2)
\end{array}$$

where @ is concatenation of arrays. Note that this sequential definition does not imply that the work has to be sequential – a traditional `scan` can be parallelised if the operator is associative, and with suitable restrictions on the function argument, the reversible version can too. We will explore parallelising the reversible `scanl` and other reversible combinators in Sect. 3.

If we define reversible addition by $\texttt{++}(x, y) = (x, x + y)$ (with the inverse `--` defined by $\texttt{--}(x, y) = (x, y - x)$), we see that $\texttt{scanl ++}\ [1, 2, 3, 4] = [1, 3, 6, 10]$, so this works as we would expect a traditional scan using addition to do.

A reversible `reduce` will have to return an array as well as the reduced value, as it would rarely be possible to restore the original array from the reduced value alone. Letting `reduce` return its argument alongside the reduced value seems the most natural choice. We can define a reversible `reduce` by

$$\begin{array}{ll}
\texttt{reduce}\ f\ [x] & = (x, [x]) \\
\texttt{reduce}\ f\ ([x]@xs) & = (z_1, [z_2]@ys) \\
& \quad \texttt{where}\ (y, ys) = \texttt{reduce}\ f\ xs\ \texttt{and}\ (z_1, z_2) = f\ (x, y)
\end{array}$$

Note that this is undefined on empty lists, as we would otherwise need a default value. $\texttt{reduce ++}\ xs$ will return $(sum\ xs, xs)$. Note that, since we use x twice in the first rule, the inverse of `reduce` is only defined if these are equal.

We, additionally, need combinators to combine and split arrays: $\texttt{concat} : \forall 'a.['a] \times ['a] \rightleftharpoons \texttt{int} \times ['a]$ concatenates two arrays and returns both the size of the first array and the concatenated array. The inverse $\texttt{splitAt} : \forall 'a.\texttt{int} \times ['a] \rightleftharpoons ['a] \times ['a]$ splits an array into two such that the first has the size given by a parameter. If the array is smaller than this size, the result is undefined.

`copy` : $\forall 'a.['a] \rightleftharpoons ['a] \times ['a]$ `copy` $x = (x, x)$
`uncopy` : $\forall 'a.['a] \times ['a] \rightleftharpoons ['a]$ `uncopy` $(x, x) = x$. If $x[i] \neq y[i]$ for any i, then `uncopy` (x, y) is undefined.
`map` : $\forall 'a, 'b.('a \rightleftharpoons 'b) \rightarrow ['a] \rightleftharpoons ['b]$ If $ys =$ `map` $f\, xs$, then $ys[i] = f(xs[i])$.
`zip` : $\forall('a, 'b).['a] \times ['b] \rightleftharpoons ['a \times 'b]$ If $zs =$ `zip` $(xs\, ys)$, then $zs[i] = (xs[i], ys[i])$. If xs and ys have different sizes, the result is undefined.
`unzip` : $\forall 'a, 'b.['a \times 'b] \rightleftharpoons ['a] \times ['b]$ If $(ys, zs) =$ `unzip` xs, then $(ys[i], zs[i]) = xs[i]$.
`iota` : `int` $\rightleftharpoons$ [`int`] `iota` $m = [0, 1, \ldots, m-1]$
`atoi` : [`int`] $\rightleftharpoons$ `int` `atoi` $[0, 1, \ldots, m-1] = m$. It is undefined on inputs not having this form.
`scanl` : $\forall 'a.('a \times 'a \rightleftharpoons 'a \times 'a) \rightarrow ['a] \rightleftharpoons ['a]$ See definition in Section 2.
`reduce` : $\forall 'a.('a \times 'a \rightleftharpoons 'a \times 'a) \rightarrow ['a] \rightleftharpoons 'a \times ['a]$ See definition in Section 2.
`concat` : $\forall 'a.['a] \times ['a] \rightleftharpoons$ `int` $\times ['a]$ If $(m, zs) =$ `concat` (xs, ys), then $m = \lvert xs\rvert$, $\lvert zs\rvert = m + \lvert ys\rvert$, and $xs[i] = zs[i]$ if $i < m$, and $zs[i] = xs[i]$ if $i < m$, and $zs[i] = ys[i - m]$ otherwise.
`splitAt` : $\forall 'a.$`int` $\times ['a] \rightleftharpoons ['a] \times ['a]$ If $(xs, ys) =$ `splitAt` (m, zs), then $m = \lvert xs\rvert$, $\lvert zs\rvert = m + \lvert ys\rvert$, and $xs[i] = zs[i]$ if $i < m$, and $ys[i] = zs[i + m]$ otherwise. If m is greater than the size of zs, the result is indefined.
`reorder` : $\forall 'a.$[`int` $\times 'a$] $\rightleftharpoons$ [`int` $\times 'a$] If $ys =$ `reorder` xs and $xs[i] = (j, v)$, then $ys[j] = (i, v)$.

Fig. 2. Reversible array combinators

Lastly, we might want to reorder the elements of an array. `reorder` : $\forall 'a.$[`int` $\times 'a$] $\rightleftharpoons$ [`int` $\times 'a$] takes a list of pairs of indices and values, and creates a new array where each element is at the given index, and the elements are paired with their old indices. It is its own inverse. If there are duplicated indices or any index is outside the array, the result is undefined. For example, `reorder [(2,17), (1,21), (0,13)] = [(2,13), (1,21), (0,17)]`. `reorder` is similar to the gather operation in normal array programming.

We will omit an explicit filter combinator, as this can not be made reversible. We will in the examples later show how you can code something similar to a filter using the other combinators.

The set of reversible array combinators and their types is shown in Fig. 2.

3 Parallel Implementation

We choose a simple model of vector-parallel reversible computers. We use Janus-like reversible updates [4], procedures with call-by-reference parameters (as in Janus), and add a parallel loop `parloop` that when given a variable i, a number n, and some reversible code that uses i, in parallel executes the code for i being equal to all numbers from 0 to $n-1$. It is assumed that the loop "iterations" are independent: No two iterations write to the same location, and if one iteration writes to a location, no other iteration may read from this location.

A function $f : a \rightleftharpoons b$ is implemented as a procedure f' that takes references to argument and result locations, and as a net effect clears the argument location, and puts the result in the result location (not necessarily in this order). A cleared array variable is a null pointer, a cleared number has the value 0, and a cleared pair has two cleared components. Some functions $g : a \rightleftharpoons a$ can be implemented in-place: g' returns its result in the location in which the argument was given. We do not allow aliasing of the parameters. A function $h : c \to a \rightleftharpoons b$ is implemented by a procedure h' that takes three arguments, where the first is a read-only pointer (typically to a procedure) and the other two are handled as above. We assume access to a reversible memory allocation procedure `alloc` that allocates a zero-initialised array of a given size, and its inverse that frees an array that is assumed to be all zeroes. Such an allocator is described in earlier literature [1]. We subscript `malloc` with the element type, as that affects the size of the allocation.

3.1 The Simple Cases

$y =$ `map` f x; where $f : a \rightleftharpoons b$ can be implemented in parallel using `parloop`.

```
procedure map'(f : a ⇌ b, x : [a], y : [b])
  local t : int;   t += size(x);   call alloc_b(y,t);
  parloop i t {   call f'(x[i], y[i]);  }
  uncall alloc_a(x,t);   t -= size(y);
end
```

If $a = b$, we can reuse the space for x instead of allocating new space:

```
procedure mapInPlace'(f' : a ⇌ a, x : [a])
  local t : int;   t += size(x);
  parloop i t { local u : a;  u <-> x[i];   call f'(u, x[i]); }
  t -= size(x);
end
```

Note that we need a local variable `u` to avoid aliasing in the call to f' and to obey the invariant that the second parameter to f' is initially clear and that the first will be cleared as a result of applying f'. If f'' is an in-place procedure implementing f, we can simplify even further:

```
procedure mapInPlace2'(f'' : a ⇌ a, x : [a])
  local t : int;   t += size(x);
  parloop i t {  call f''(x[i]); }
  t -= size(x);
end
```

(y, z) = `copy` x, where $x : [a]$ is similarly implemented:

```
procedure copy'(x : [a], yz : [a] × [a])
  local t : int, z : [a];   t += size(x);
  call alloc_a(z,t);
  parloop i t {  z[i] += x[i];  }
  t -= size(x);   call makePair(x,z,yz)
end
```

Note that we reuse x for y. `makePair`$(x,$`z`$,yz)$ is procedure that creates in yz a pair of x and `z` and resets x and `z` to zero.

z `= zip` (x, y); where $x : [a]$ and $y : [b]$ can be implemented by

```
procedure zip'(x : [a], y : [b], z : [a × b])
  local t : int;   t += size(x);   call alloc_{a×b}(z,t);
  parloop i t {   z[i].0 <-> x[i];    z[i].1 <-> y[i];  }
  uncall alloc_a(x,t);   uncall alloc_b(y,t);   t -= size(z);
end
```

where z`[i].0` and z`[i].1` are the first and second components of the pair z`[i]`. Note that we assume the sizes of the arrays x and y to be the same.

x `= iota` n can be implemented as

```
procedure iota'(n : int, x : [int])
  call alloc_int(x,n);
  parloop i n {  x[i] += i;  }
  n -= size(x)
end
```

(n, z) `= concat` (x, y), where $x : a$ can be implemented as

```
procedure concat'(x : [a], y : [a], nz : int × [a])
  local t : int, u : int;   t += size(x);   u += size(y);
  call alloc_a(nz.1,t+u);
  parloop i t {   nz.1[i] <-> x[i];  }
  parloop i u {   nz.1[i+t] <-> y[i];  }
  uncall alloc_a(x,t);   uncall alloc_a(y,u);
  u -= size(nz.1) - t;   nz.0 <-> t; end
```

y `= reorder` x, where $x : [$`int` $\times a]$ can be implemented as

```
procedure reorder'(x : [int × a], y : [int × a])
  local t : int;   t += size(x)   alloc_{int×a}(y,t);
  parloop i t {
  local j : int;
    j += x[i].0;   y[j].0 += i;
    y[j].1 <-> x[i].1;   j -= x[i].0;
  }
  parloop i t {
  local j : int;
    j += y[i].0;   x[j].0 -= i;   j -= y[i].0;
  }
  free_{int×a}(x,t);   t -= size(y)
end
```

The first parloop gives y its correct values and clears the second component of each pair in the x array. We need a second parloop to clear the first components of the pairs in the x array. Note that if `j` takes on the same value in different iterations of the parallel loop, there may be race conditions that makes the result undefined.

3.2 `reduce`

In the irreversible case, `reduce` can be parallelised if the reduction operator $\oplus : a \times a \to a$ is associative. In the reversible case, we work with functions of the type $f : a \times a \rightleftharpoons a \times a$, so we need to find conditions on such functions that allow parallel implementation. The method below works if f is the identity in its first parameter and associative in its second parameter, i.e., $f(m, n) = (m, m \oplus n)$, where $\oplus$ is an associative operator. An example is `++`, as `++`$(m, n) = (m, m+n)$.

(v, x) `=` `reduce` f x (note that we reuse x in the output) is implemented by the procedure `reduce'` in Fig. 3. It takes as arguments f', which is an implementation of f, v which is a location for the result, and x, which is the array to be reduced.

If the size of x is 1, we just copy the sole element of x into v. This is the base case of our induction. Otherwise, we allocate space for an intermediate array `y` of half the size of x. In the first parloop, we compute f' on pairs of consecutive elements of x, putting the identity part of the result back into x (leaving every other element as 0) and the sum part into `y`, so $x[2i] = x^0[2i]$, $x[2i+1] = 0$, and $\mathtt{y}[i] = x^0[2i] \oplus x^0[2i+1]$, where x^0 is the original x array before the updates. We then call `reduce'` recursively on `y`, which (by induction) leaves `y` unchanged and stores the result of the reduction in v. We now need to uncompute `y` and restore x to its original values. We do that in the second parloop which is the reverse of the first. The conditional after the second parloop handles odd-sized arrays: The last element of x is "added" to v. Finally, we free the `y` array.

We illustrate this by an example: applying `reduce ++` to an array $x = [x_0, x_1, x_2, x_3, x_4, x_5]$. After the parloop and before the first recursive call, we have $x = [x_0, 0, x_2, 0, x_4, 0]$ and $\mathtt{y} = [x_0+x_1, x_2+x_3, x_4+x_5]$. In the recursive

```
procedure reduce'(f' : a × a ⇌ a × a, v : a, x : [a])
  local t : int, y : [a];
  t += size(x);
  if t == 1 then
    v += x[0];
  else {
    call alloc_a(y, ⌊(t/2)⌋;
    parloop i ⌊(t/2)⌋ {
      local u : a × a, w : a × a;
      u.0 <-> x[2*i];  u.1 <-> x[2*i+1];
      call f'(u, w);
      w.0 <-> x[2*i];  w.1 <-> y[i];
    }
    call reduce'(f', v, y);
    parloop i ⌊(t/2)⌋ {
      local u : a × a, w : a × a;
      w.1 <-> y[i];  w.0 <-> x[2*i];
      uncall f'(u, w);
      u.1 <-> x[2*i+1];  u.0 <-> x[2*i];
    }
    if t%2 != 0 then {
      local u : a × a, w : a × a;
      u.0 <-> x[t-1];  u.1 <-> v;
      call f'(u, w);
      w.0 <-> x[t-1];  w.1 <-> v;
    }
    fi t%2 != 0
    uncall alloc_a(y, ⌊(t/2)⌋);
  }
  fi t == 1;
  t -= size(x);
end
```

Fig. 3. Parallel implementation of `reduce`

invocation, we have $x = [x_0 + x_1,\ x_2 + x_3,\ x_4 + x_5]$. After the parloop, we have $x = [x_0 + x_1,\ 0,\ x_4 + x_5]$ and $\mathtt{y} = [x_0 + x_1 + x_2 + x_3]$. In the next recursive call, the array size is 1, so it will return $v = x_0 + x_1 + x_2 + x_3$ and `y` unchanged. The second parloop is the inverse of the first, so it returns x to $[x_0 + x_1,\ x_2 + x_3,\ x_4 + x_5]$ and clear `y`. Since `t` is odd, we enter the conditional and modify v to $x_0 + x_1 + x_2 + x_3 + x_4 + x_5$. When we return, we have $x = [x_0,\ 0,\ x_2,\ 0,\ x_4,\ 0]$, $\mathtt{y} = [x_0 + x_1,\ x_2 + x_3,\ x_4 + x_5]$, and $v = x_0 + x_1 + x_2 + x_3 + x_4 + x_5$. The second parloop restores x to $[x_0,\ x_1,\ x_2,\ x_3,\ x_4,\ x_5]$ and clears `y`. Since `t` is even, we skip the conditional and return $v = x_0 + x_1 + x_2 + x_3 + x_4 + x_5$ along with the original x.

3.3 `scanl`

In the irreversible setting, scans of associative operators can be parallelised using a method called *parallel scan* or *prefix sum*. We use a variant of this that shares some structure with the implementation of `reduce` above and also requires that the reversible function used in the scan is identity in its first parameter and

```
procedure scanl'(f' : a × a ⇌ a × a, x : [a])
  local t : int, y : [a];
  t += size(x);
  if t < 2 then
    skip;
  else
    call alloc_a(y, ⌊(t/2)⌋);
    parloop i ⌊(t/2)⌋ {
      local u : a × a, w : a × a
      u.0 <-> x[2*i];  u.1 <-> x[2*i+1];
      call f'(u, w);
      w.0 <-> x[2*i];  w.1 <-> y[i];
    }
    call scanl'(f', y);
    parloop i ⌊((t-1)/2)⌋ {
      local u : a × a, w : a × a
      u.0 <-> y[i];  u.1 <-> x[2*i+2];
      call f'(u, w);
      w.0 <-> x[2*i+1];  w.1 <-> x[2*i+2];
    }
    if t%2 == 0 then
      x[t-1] <-> y[t/2-1];     fi t%2 == 0;
    uncall alloc_a(y, ⌊(t/2)⌋);
  fi t < 2;
  t -= size(x);
end
```

Fig. 4. Parallel implementation of `scanl`

associative in its second parameter. We output the result in the same array as the input. A procedure `scanl'` that implements $x := \texttt{scanl}(f, x)$ can be seen in Fig. 4. It takes as arguments f', which is an implementation of f and the array x, which is updated in place.

The base case is an array x of size less than two, which is left unchanged. In the general case, we allocate an array `y` of half the size of x. The first parloop is as in `reduce'`, and computes f' on pairs of consecutive elements of x, putting the identity part of the result back into x (leaving every other element as 0) and the sum part into `y`, so $x[2i] = x^0[2i]$, $x[2i{+}1] = 0$, and $\texttt{y}[i] = x^0[2i] \oplus x^0[2i{+}1]$, where x^0 is the original x array before the updates. We then call `scanl'` recursively on `y`, which by induction makes `y` the scan of the original `y`. The elements of this `y` are the odd-indexed elements of the reduced x^0. The odd elements of x are currently 0, so we can just swap the elements of y with the odd-indexed elements of x. The even-indexed elements of x contain the original values of the even-indexed elements of x^0. The first of these is correct, but the subsequent ones needs to be added to the preceding element of x. We do this in the second parloop with calls to f' and suitable swaps. Again, we need to do some fix-up to handle odd-sized arrays, where the last element of x (which is currently 0) is swapped with the last element of `y` (which contains the "sum" of all elements of x^0). Finally, `y`, which is now cleared, is freed.

We use `scanl ++`x, where $x = [x_0, x_1, x_2, x_3, x_4, x_5]$ as an example. After the parloop and before the first recursive call, we have $x = [x_0, 0, x_2, 0, x_4, 0]$

and y = $[x_0 + x_1, x_2 + x_3, x_4 + x_5]$. In the recursive invocation, we have $x = [x_0 + x_1, x_2 + x_3, x_4 + x_5]$. After the parloop, we have $x = [x_0 + x_1, 0, x_4 + x_5]$ and y = $[x_0 + x_1 + x_2 + x_3]$. The next recursive call will return y unchanged, as its size is 1. t = 3, so we do one iteration of the second parloop, getting $x = [x_0 + x_1, x_0 + x_1 + x_2 + x_3, x_0 + x_1 + x_2 + x_3 + x_4 + x_5]$ and y = [0]. Since t is odd, we skip the conditional, free y, and return. At the return, we get $x = [x_0, 0, x_2, 0, x_4, 0]$ and y = $[x_0 + x_1, x_0 + x_1 + x_2 + x_3, x_0 + x_1 + x_2 + x_3 + x_4 + x_5]$. t = 6, so we do two iterations of the second parloop and get $x = [x_0, x_0 + x_1, x_0 + x_1 + x_2, x_0 + x_1 + x_2 + x_3, x_0 + x_1 + x_2 + x_3 + x_4, 0]$ and y = $[0, 0, x_0 + x_1 + x_2 + x_3 + x_4 + x_5]$. t is now even, so we enter the conditional and get $x = [x_0, x_0 + x_1, x_0 + x_1 + x_2, x_0 + x_1 + x_2 + x_3, x_0 + x_1 + x_2 + x_3 + x_4, x_0 + x_1 + x_2 + x_3 + x_4 + x_5]$ and y = [0, 0, 0], so we can free y and return.

The scan is not entirely in-place, as we use local arrays y, the total size of which is almost that of the original. We could make it entirely in-place by doubling the stride in each recursive call instead of copying to a new array of half the size.

4 A Reversible Array Programming Language

We are now ready to define a reversible array programming language, which we will call "Agni", named after the two-faced Hindu god of fire. The syntax is shown in Fig. 5. Note that <=> is an easier-to-type alternative notation for ⇌. → is easily accessible on an international keyboard, so we have not replaced this with the pure-ASCII alternative ->.

We work with both heap-allocated and stack-allocated variables Heap-allocated variables are denoted $HVar$ in the grammar, and contain values that are consumed nad produced by reversible operations. Stack-allocated variables are denoted $SVar$, and contain values or functions that are locally defined using irreversible expressions and then uncomputed at the end of their scope. To distinguish these, heap-allocated variable names start with upper-case letters, while stack-allocated variable and function names start with lower-case letters. Heap-allocated variables have heap types (denoted $HType$) and stack-allocated variables have stack types (denoted $SType$).

A stack-allocated variable is introduced using a let-expression that initialises it, allows multiple uses of the variable inside its body, and uncomputes its value at the end. In reverse, the uncomputation and initialisation swap roles. The expressions used by initialisation and uncomputation need not be reversible. This is analogous to how reversible updates in Janus can use irreversible expressions. A stack-allocated variable can not hold an array, but array elements and sizes can be used in its initialisation and uncomputation expressions. An irreversible expression is denoted Exp in the grammar.

Variables holding heap-allocated values are explicitly initialised using a reversible initialisation. The heap-allocated variables on the right-hand side of this initialisation are consumed and can no longer be used. In reverse, the variables on the left-hand side are consumed and those on the right are initialised.

$$
\begin{array}{lll}
SType & \rightarrow & \texttt{int} \\
SType & \rightarrow & TypeVar \\
SType & \rightarrow & SType \times SType \\
SType & \rightarrow & SType \ \rightarrow \ SType \\
SType & \rightarrow & HType \ \texttt{<=>} \ HType \\
 & & \\
HType & \rightarrow & \texttt{int} \\
HType & \rightarrow & TypeVar \\
HType & \rightarrow & HType \times HType \\
HType & \rightarrow & [HType] \\
 & & \\
Rinit & \rightarrow & \\
Rinit & \rightarrow & HPattern := Rexp; \\
Rinit & \rightarrow & Rinit \ Rinit \\
Rinit & \rightarrow & \texttt{let} \ SPattern = Exp \ \texttt{in} \ Rinit \ \texttt{end} \ SPattern = Exp; \\
Rinit & \rightarrow & \texttt{def} \ FunDef \ \texttt{in} \ Rinit \ \texttt{end}; \\
 & & \\
Rexp & \rightarrow & HVar \\
Rexp & \rightarrow & SVar \\
Rexp & \rightarrow & (Rexp, \ Rexp) \\
Rexp & \rightarrow & Fname \ Rexp \\
Rexp & \rightarrow & \texttt{uncall} \ Fname \ Rexp \\
Rexp & \rightarrow & Fname \ Svar \ Rexp \\
Rexp & \rightarrow & \texttt{uncall} \ Fname \ Svar \ Rexp \\
 & & \\
Exp & \rightarrow & IntConst \\
Exp & \rightarrow & SVar \\
Exp & \rightarrow & HVar[Exp] \\
Exp & \rightarrow & \texttt{size} \ HVar \\
Exp & \rightarrow & (Exp, \ Exp) \\
Exp & \rightarrow & Fname \ Exp \\
Exp & \rightarrow & \texttt{let} \ SPattern = Exp \ \texttt{in} \ Exp \\
Exp & \rightarrow & \texttt{def} \ FunDef \ \texttt{in} \ Exp \\
 & & \\
FunDef & \rightarrow & Fname \ SPattern = Exp \\
FunDef & \rightarrow & Fname \ HPattern = Rinit \ Rexp \\
FunDef & \rightarrow & Fname \ SPattern \ HPattern = Rinit \ Rexp \\
 & & \\
HPattern & \rightarrow & SVar \\
HPattern & \rightarrow & HVar : HType \\
HPattern & \rightarrow & (HPattern, \ HPattern) \\
 & & \\
SPattern & \rightarrow & SVar : SType \\
SPattern & \rightarrow & (SPattern, \ SPattern) \\
 & & \\
Program & \rightarrow & FunDef
\end{array}
$$

Fig. 5. Syntax of Agni

A heap-allocated variable is in scope from its initialisation to its consumption, and can in this scope be used in expressions that define stack-allocated variables. Reversible initialisations are denoted *Rinit* in the grammar, and can in addition to a simple initialisation be a (possible empty) sequence of initialisations or a local definition of a function or stack-allocated variable that is locally used for an initialisation.

Function definitions are defined using def-expressions, and have scope until the end of the def-expression. A function definition can not consume heap-allocated variables that are not given as parameters or initialised locally, and all parameters or locally initialised variables that are heap-allocated must be either consumed in the function body or returned as part of the result. In a function type, parameters that appear in the function type before a $\rightleftharpoons$ arrow are heap allocated, whereas parameters (including function parameters) that appear before a $\rightarrow$ arrow are stack-allocated. All stack-allocated parameters must occur before heap-allocated parameters. The body of a reversible function is an optional reversible initialisation followed by a reversible expression. The body of an irreversible function is an irreversible expression. Note that function definitions can occur in both reversible initialisations and in irreversible expressions with slightly different syntax.

A program is a single function definition that defines a reversible function, so it must have type $t_1 \rightleftharpoons t_2$ for some types t_1 and t_2. Its body is a reversible initialisation followed by a reversible expression.

A reversible expression is a variable, a pair of two reversible expressions, or a (possibly inverse) reversible function application.

A reversible expression can be a stack-allocated variable. This will not be consumed by the expression. Likewise, a reversible pattern can be a stack-allocated variable ($SVar$). This is not a defining instance (so no type needs to be given), but when a heap-allocated value is matched against a stack-allocated variable, it must have the same value as the variable, and is consumed by this. If the variable does not match the value, the behaviour is undefined. When a heap-allocated variable ($HVar$) occurs in a pattern for heap-allocated values, it defines a new variable, so a type is given.

The program is evaluated in a context that defines both a number of irreversible functions for use in initialisation and uncomputation of stack-allocated variables and a number of reversible functions for defining heap-allocated variables. The latter includes the array combinators shown in Fig. 2. Note that "normal" addition $+ : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$ and similar operators are part of the set of irreversible functions that can be used.

For simplicity, we do not have an explicit boolean type, so truth values are represented as integers. Non-zero integers are considered true and 0 is considered false. We have also omitted reals, as garbage-free reversible arithmetic on floating point numbers is still an open issue.

$$\frac{}{\rho, \gamma \vdash k : \texttt{int}} \qquad \frac{\gamma\, X = [t]}{\rho, \gamma \vdash \texttt{size}\ X : \texttt{int}} \qquad \frac{\gamma\, X = [t] \quad \rho, \gamma \vdash e : \texttt{int}}{\rho, \gamma \vdash X[e] : t}$$

$$\frac{\rho\, x = t}{\rho, \gamma \vdash x : t} \qquad \frac{\rho, \gamma \vdash e_1 : t_1 \quad \rho, \gamma \vdash e_2 : t_2}{\rho, \gamma \vdash (e_1,\, e_2) : t_1 \times t_2}$$

$$\frac{\rho\, f = \tau \quad t_1 \to t_2 = \mathit{instantiate}\, \tau \quad \rho, \gamma \vdash e : t_1}{\rho, \gamma \vdash f\ e : t_2}$$

$$\frac{\rho, \gamma \vdash e_1 : t_1 \quad \rho \vdash_S s_1 \rhd \rho_1/t_3 \quad t_1 = t_3 \quad \rho_1, \gamma \vdash e_2 : t_2}{\rho, \gamma \vdash \texttt{let}\, s_1 = e_1\, \texttt{in}\, e_2 : t_2}$$

$$\frac{\begin{array}{c}\rho \vdash_S s \rhd \rho_1/t_1 \quad \rho_1[f : t_1 \to t_2], \gamma \vdash e_1 : t_2 \\ \tau = \mathit{generalize}(t_1 \to t_2,\, \rho) \quad \rho[f : \tau], \gamma \vdash e_2 : t_3\end{array}}{\rho, \gamma \vdash \texttt{def}\, f\, s = e_1\, \texttt{in}\, e_2 : t_3}$$

$$\frac{}{\rho \vdash_S x : t \rhd \rho[x : t]/t} \qquad \frac{\rho \vdash_S s_1 \rhd \rho_1/t_1 \quad \rho_1 \vdash_S s_2 \rhd \rho_2/t_1}{\rho \vdash_S (s_1,\, s_2) \rhd \rho_2/(t_1 \times t_2)}$$

Fig. 6. Type rules for irreversible expressions and patterns

5 A Type System for Reversible Array Programming

We use different environments for stack-allocated variables and functions and for heap-allocated variables. Evaluating an expression has no net effect on the environment of stack-allocated variables and functions, but it may affect the environment of heap-allocated variables, as some of these are consumed and others initialised in a way that does not follow block structure. We use (possibly subscripted) ρ for environments of stack-allocated variables and functions, and γ for environments heap-allocated variables. When we evaluate an irreversible expression, we can use variables (and functions) from both environments, but modify none of them. When we evaluate a reversible expression, we can also use both, but we may remove variables from γ, as these are used. When evaluating a reversible initialisation, we can use variables from ρ and both remove and add variables in γ.

We start by defining type rules for irreversible expressions and patterns in Fig. 6. We use t to denote an $SType$, x to denote an $SVar$, X to denote an $HVar$, and s to denote an $SPattern$. The rules are straightforward except the function rule which uses implicit unification to allow recursive definitions, and use generalisation and instantiation to implement parametric polymorphism. τ denotes a polymorphic type. We omit descriptions of generalisation and instantiation, but note that these are as in Hindley-Milner type inference. The two last rules are for patterns, which both extend an environment with new bindings and build a type for the pattern. We assume no variable occurs twice in a pattern.

Figure 7 shows rules for reversible expressions. T denotes an $HType$, and r denotes an $RExp$. $\uparrow(t)$ transforms an $SType$ (excluding function types) to the equivalent $HType$. Note how γ is threaded around in the rules.

$$\frac{}{\rho, \gamma[X : T] \vdash_R X : T/\gamma} \qquad \frac{}{\rho[x : t], \gamma \vdash_R x :\uparrow (t)/\gamma}$$

$$\frac{\rho, \gamma \vdash_R r_1 : T_1/\gamma_1 \quad \rho, \gamma_1 \vdash_R r_2 : T_2/\gamma_2}{\rho, \gamma \vdash_R (r_1, r_2) : (T_1 \times T_2)/\gamma_2}$$

$$\frac{\rho\, f = \tau \quad (T_1 \rightleftharpoons T_2) = instantiate\, \tau \quad \rho, \gamma \vdash_R r : T_1/\gamma_1}{\rho, \gamma \vdash_R f\, r : T_2/\gamma_1}$$

$$\frac{\rho\, f = \tau \quad (T_1 \rightleftharpoons T_2) = instantiate\, \tau \quad \rho, \gamma \vdash_R r : T_2/\gamma_1}{\rho, \gamma \vdash_R \texttt{uncall}\ f\, r : T_1/\gamma_1}$$

$$\frac{\rho\, f = \tau \quad (t \to T_1 \rightleftharpoons T_2) = instantiate\, \tau \quad \rho\, x = t \quad \rho, \gamma \vdash_R r : T_1/\gamma_1}{\rho, \gamma \vdash_R f\, x\, r : T_2/\gamma_1}$$

$$\frac{\rho\, f = \tau \quad (t \to T_1 \rightleftharpoons T_2) = instantiate\, \tau \quad \rho\, x = t \quad \rho, \gamma \vdash_R r : T_2/\gamma_1}{\rho, \gamma \vdash_R \texttt{uncall}\ f\, x\, r : T_1/\gamma_1}$$

Fig. 7. Type rules for reversible expressions

Figure 8 shows rules for reversible initialisations and patterns. Like reversible expressions, reversible initialisations thread γ around, but they do not return values. The most complicated rule is for $\texttt{let}\, s_1 = e_1\, \texttt{in}\, I\, \texttt{end}\, s_2 = e_2$, where s_1 and s_2 are required to contain the same variables so the variables that are introduced in s_1 are eliminated in s_2. The rules for function definitions are similar to those for irreversible expressions, except that they also include reversible function definitions. Note that a reversible function definition starts and ends with empty γs, as they can only consume their arguments and produce their results with no remaining unconsumed $RVar$s.

Reversible patterns produce both a $Htype$ and a new γ. The first rule states that when using an $SVar$ in a pattern, its $SType$ is converted to the equivalent $HType$ using the $\uparrow$ operator. Only non-functional $STypes$ can be converted.

6 Examples

Since the reversible functional array programming language is limited compared to irreversible array programming languages, we need to justify that it can be used to solve real problems. We do so by showing some example programs.

6.1 Inner Product

An inner product of two vectors reduces these vectors to a single number, so we need to return these vectors along with the result. The code is

$$\frac{}{\rho, \gamma \vdash_I \rightsquigarrow \gamma} \qquad \frac{\rho, \gamma \vdash_I I_1 \rightsquigarrow \gamma_1 \quad \rho, \gamma_1 \vdash_I I_2 \rightsquigarrow \gamma_2}{\rho, \gamma \vdash_I I_1\ I_2 \rightsquigarrow \gamma_2}$$

$$\frac{\rho, \gamma \vdash_R r : T/\gamma_1 \quad \rho, \gamma_1 \vdash_P p \rhd \gamma_2/T}{\rho, \gamma \vdash_I p := r; \rightsquigarrow \gamma_2}$$

$$\frac{\begin{array}{c}\rho, \gamma \vdash e_1 : t_1 \quad \rho \vdash_S s_1 \rhd \rho_1/t_3 \quad t_1 = t_3 \quad \rho, \gamma \vdash_I I \rightsquigarrow \gamma_1 \\ \rho, \gamma_1 \vdash e_2 : t_2 \quad \rho \vdash_S s_2 \rhd \rho_1/t_4 \quad t_2 = t_4\end{array}}{\rho, \gamma \vdash \texttt{let}\ s_1 = e_1\ \texttt{in}\ I\ \texttt{end}\ s_2 = e_2; \rightsquigarrow \gamma_1}$$

$$\frac{\begin{array}{c}\rho \vdash_S s \rhd \rho_1/t_1 \quad \rho_1[f : t_1 \to t_2], \gamma \vdash e_1 : t_2 \\ \tau = \mathit{generalize}(t_1 \to t_2, \rho) \quad \rho[f : \tau], \gamma \vdash_I I \rightsquigarrow \gamma_1\end{array}}{\rho, \gamma \vdash \texttt{def}\ f\ s = e\ \texttt{in}\ I; \rightsquigarrow \gamma_1}$$

$$\frac{\begin{array}{c}\rho, [] \vdash_P p \rhd \gamma_1/T_1 \quad \rho_1[f : T_1 \rightleftharpoons T_2], \gamma_1 \vdash_I I_1 \rightsquigarrow \gamma_2 \\ \rho_1, \gamma_2 \vdash_R r : T_2/[] \quad \tau = \mathit{generalize}(f : T_1 \rightleftharpoons T_2, \rho) \quad \rho[f : \tau], \gamma \vdash_I I_2 \rightsquigarrow \gamma_3\end{array}}{\rho, \gamma \vdash \texttt{def}\ f\ p = I_1\ r\ \texttt{in}\ I_2; \rightsquigarrow \gamma_3}$$

$$\frac{\begin{array}{c}\rho \vdash_S s \rhd \rho_1/t \quad \rho, [] \vdash_P p \rhd \gamma_1/T_1 \quad \rho_1[f : t \to T_1 \rightleftharpoons T_2], \gamma_1 \vdash_I I_1 \rightsquigarrow \gamma_2 \\ \rho_1, \gamma_2 \vdash_R r : T_2/[] \quad \tau = \mathit{generalize}(f : t \to T_1 \rightleftharpoons T_2, \rho) \quad \rho[f : \tau], \gamma \vdash_I I_2 \rightsquigarrow \gamma_3\end{array}}{\rho, \gamma \vdash \texttt{def}\ f\ s\ p = I_1\ r\ \texttt{in}\ I_2; \rightsquigarrow \gamma_3}$$

$$\frac{T = \uparrow (\rho\, x)}{\rho, \gamma \vdash_P x \rhd \gamma/T} \qquad \frac{}{\rho, \gamma \vdash_P X : T \rhd \gamma[X : T]}$$

$$\frac{\rho, \gamma \vdash_P p_1 \rhd \gamma_1/T_1 \quad \rho, \gamma_1 \vdash_P p_2 \rhd \gamma_2/T_1}{\rho, \gamma \vdash_P (p_1, p_2) \rhd \gamma_2/(T_1 \times T_2)}$$

Fig. 8. Type rules for reversible initialisations and patterns

```
fun inner (Xs: [int], Ys: [int]) =
  (Xs: [int], Prods: [int]) := unzip (map ** (zip (Xs, Ys)));
  (Ip: int, Prods: [int]) := reduce ++ Prods;
  (Ip, unzip (map // (zip (Xs, Prods))))
```

We note that `**`$(x, y) = (x, x * y)$ and `//(x,y)` $= (x, y/x)$, so they are inverses and both undefined if $x = 0$.

We first zip the two vectors, map `**` to get the product of each pair (while retaining one operand), unzip to get separate arrays for the product and the copies, reduce with `++` to get the inner product, and undo the multiplications to get the original vectors back. Note that we redefine `xs`, but since the original `xs` has already been consumed at this point, it leads to no ambiguity.

This is, admittedly, more cumbersome than doing inner product in a normal irreversible language. We could shorten it somewhat by adding a `zipWith` combinator that combines `map` and `zip`.

6.2 Counting the Number of Elements that Satisfy a Predicate

We don't have a separate boolean type, so we use zero/nonzero instead. A reversible predicate has the type $'a \rightleftharpoons 'a \times \texttt{int}$ for some type $'a$ and will pair a value with the result of the predicate, which is 1 for true and 0 for false. We can map this on an array, extract the numbers, add them using `reduce`, zip the numbers back to the array, and map the inverse of the predicate (which is done by uncalling map) to eliminate the numbers. The result is a pair of the count and the original list.

```
count (p: 'a <=> '{a}\times{int}) (Xs: ['a]) =
  (Xs: ['a], Ps: [int]) := unzip (map p Xs);
  (Count: int, Ps: [int]) := reduce ++ Ps;
  Xs: ['a] := uncall map p (zip (Xs,Ps));
  (Count, Xs)
```

Note that the third line is the inverse of the first line.

6.3 Separation by Predicate

As noted in Sect. 2, we don't include a filter operator, but sometimes, we will need to separate the elements where the predicate is true from the elements where it is false. This can be used, e.g., for quicksort or radix sort. Such a separation is not reversible, so we should expect some garbage output as well. In this example, this garbage is two arrays of integers, each the size of the original array:

$$\texttt{separate} : ('a \rightleftharpoons 'a \times \texttt{int}) \rightarrow ['a] \rightleftharpoons ['a] \times ['a] \times [\texttt{int}] \times [\texttt{int}]$$

The garbage can be reduced to a copy of the original array by calling `separate`, copying the separated arrays, uncalling `separate`, and combining the separated arrays with the original, as shown in the function `separateClean` below. The `separate` function works in the following steps:

1. Map the predicate over the array, pairing each element with (1,0) if the predicate is true and (0,1) if the predicate is false.
2. Use `scanl` twice to compute the number of true and false values before each array element.
3. Extract the total number of true booleans `tmax` from the last element of the new array.
4. Use `map findLoc` to compute the new location of each element, where `findLoc` chooses between the number of previous true elements and the number of false elements + `tmax` depending on the predicate.
5. Use `reorder` to place elements in their new locations.
6. Split into true and false arrays.
7. Returns these array and the garbage arrays.

The code is shown in Fig. 9

```
separate (p: 'a <=> 'a×int) (Xs: ['a]) =
  let one: int = 1 in
    def tf N: int = --(N,one) in
      (Xs: ['a], Ps: [int]) := unzip (map p Xs);
      (Ts: [int], Fs: [int]) := unzip (map tf Ps);
      Tsbefore: [int] := scanl ++ Ts;
      Fsbefore: [int] := scanl ++ Fs;
      let tmax: int = Tsbefore[size Tsbefore - 1]) in
        Bsbefore: [int×int] := zip (Tsbefore, Fsbefore);
        Bsxs: [(int×int)×'a] := zip (Bsbefore, Xs);
        def findLoc ((Tsb: int, Fsb: int), X: 'a) =
          (X: 'a, P: int) := p X;
          (tmax, Fsb1: int) := ++(tmax, Fsb);
          (P: int, Loc: int, G: int) := cswap(P, Fsb1, Tsb);
          Loc: int := dec Loc;
          X: 'a := uncall p (X, P);
          ((Loc, X), G)
        in
          (Lsxs: [int×'a], G0: [int]) := unzip (map findLoc Bsxs);
        end;
        (G1: [int], Newxs: ['a]) := unzip (reorder Lsxs);
        (Txs: ['a], Fxs: ['a]) := splitAt (tmax, Newxs);
      end tmax = size Txs
    end
  end one: int = 1;
  (Txs, Fxs, G0, G1)

separateClean (p: 'a <=> 'a×int) (Xs: ['a]) =
  (Txs: ['a], Fxs: ['a], G0: [int], G1: [int]) := separate p Xs;
  (Txs: ['a], Txs1: ['a]) := copy Txs;
  (Fxs: ['a], Fxs1: ['a]) := copy Fxs;
  Xs: ['a] := uncall separate p (Txs, Fxs, G0, G1);
  (Txs1, Fxs1, Xs)
```

Fig. 9. Implementation of `separate`

Note that the uncomputation expression for the local variable `tmax` is different from its initialisation expression. `cswap` is a predefined function that does a conditional swap: It returns the first argument unchanged, and if this is nonzero, returns the two other arguments swapped, otherwise unchanged. It can likely be implemented by a single instruction on a reversible processor. `dec` is a predefined reversible function that decrements its argument. We need to locally define a variable to be equal to 1, because we can not use constants in patterns and reversible expressions.

7 Conclusion and Future Work

We have presented reversible implementations of a number of reversible array combinators, including `reduce` and `scanl`, and we have presented a reversible functional array language, Agni, that uses these combinators. The reversible implementations of the combinators are interesting in their own right, and can be used for other languages.

Agni is, in its current form, somewhat limited, and it can be challenging to code non-trivial functions in Agni, as witnessed by the complexity of the `separate` function. We believe that the potential of getting highly parallel reversible code will make it worthwhile. Adding extra combinators, for example `zipWith` that combines `zip` and `map`, and `map2` that combines `zip`, `map`, and `unzip`, would also make coding easier and would also reduce the number of intermediate values produced. A general fusion transformation that combines several sequentially applied combinators to a single combinator would be an useful optimisation. We have avoided conditionals (except conditional swap), as conditional execution does not fit well with vector parallelisation.

We do not at the time of writing have an implementation of Agni, but we have tested the reversible implementations of the array combinators using the imperative reversible language Hermes [3], albeit with sequential loops rather than parallel loops. If and when reversible vector processors become available, we will certainly attempt to implement Agni on these. Until then, we will have to do with implementations on classical hardware, where Agni has no obvious advantage over existing functional array programming languages, such as Futhark [2]. Nevertheless, we plan in the future to make sequential and parallel implementations of Agni on classical hardware. Experiences with this may spark modifications to the language.

Speaking of Futhark, this language has an interesting type system where arrays can be constrained by size (so you can, e.g., specify that a scan preserves array size). It would be interesting to adapt this idea to Agni. It would also be useful to add type inference, so many of the type declarations can be avoided.

References

1. Cservenka, M.H., Glück, R., Haulund, T., Mogensen, T.Æ.: Data structures and dynamic memory management in reversible languages. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 269–285. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_19
2. Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 556–571. ACM, New York (2017)

3. Mogensen, T.Æ.: Hermes: a language for light-weight encryption. In: Lanese, I., Rawski, M. (eds.) RC 2020. LNCS, vol. 12227, pp. 93–110. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-52482-1_5
4. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of the 5th Conference on Computing Frontiers, CF 2008, pp. 43–54. ACM, New York (2008)