Relating two Abstractions for Formal Array Languages

Josef Svenningsson
Department of Computer Science of Engineering
Chalmers University of Technology
josef.svenningsson@chalmers.se

Lenore M. Mullin
Emeritus Professor
Department of Computer Science
University at Albany, SUNY
lenore@albany.edu

Abstract

The aim of this paper is to rectify some of the fragmentation within the high-performance array programming community. We look at two formalisms for functional, high-performance array computations: Mathematics of Arrays and Pull arrays, two seemingly different formalisms that share many principles and properties. This paper is meant to identify the striking similarities between the two and to act as a guide to enable research communications.

1. Introduction

Computing with arrays has been a constantly important paradigm in the history of computing. For this reason we have seen a plethora of formalisms and languages being developed to aid theoreticians and developers to produce more suitable abstractions and faster programs. Examples include FORTRAN, APL and Matlab but the list goes on and on. Having many different options to choose from can be a good thing but it can also lead to a balkanization of the community. When researchers dedicated to different formalisms can no longer talk to each other, it inhibits collaborations and limits the impact of innovation.

In this paper we aim to rectify some of the fragmentation of the high-performance array programming community by relating two particular formalisms: Mathematics of Arrays and Pull arrays. On the surface they appear rather different but at their core they share many principles and properties which we will examine in this paper. In particular we will highlight the following points:

- We give an introduction to both MoA (section 2) and Pull arrays (section 3).
- The two formalisms are compared with respect to types (section 4.1), fusion/ψ-reduction (section 4.2), how they deal with higher dimensions (section 4.3) and notation (section 4.4).

We would like to stress that we are not looking to declare a winner. Instead our intention is that this paper can serve as a help for the two communities to be able to learn about each others work.

2. Overview of MoA

MoA, an acoronym for *A Mathematics of Arrays[1]*, is a theory of arrays with both an algebra and a calculus of indexing based on shapes, i.e. sizes of each dimension. The index calculus, refered to as *The Psi Calculus*, gets its names from the *Psi* function which is the foundational function in the theory. These results began with the awareness that Iverson's APL[21] was inspired by Sylvester's Universal Algebra[44] and the fact that he and Cayley invented matrices.

MoA and the Psi Calculus, as a theory, have also been used to describe and verify hardware[18, 40, 41], describe the partitioning and scheduling of array operations (and their costs)[10, 12, 27, 31, 34, 38, 39], and the formulation of parallel and distributed processing[37]. MoA and ψ-Calculus, can be used to abstract complex processor memory layouts for tensor/array expressions, reduce these expressions first to a semantic normal form (Denotational Normal Form-DNF) then to an Operational Normal Form (ONF).

Elements of the theory

Indexing and Shapes

The central operation of MoA is the indexing function

$$p\psi A$$

in which a vector of n integers p is used to select an item of the n-dimensional array A. The operation is generalized to select a partition of A, so that if q has only $k < n$ components then

$$q\psi A$$

is an array of dimensionality $n - k$ and q selects among the possible choices for the first k axes. In MoA zero origin indexing is assumed. For example, if A is the 3 by 5 by 4 array

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 \end{bmatrix} \begin{bmatrix} 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 \\ 36 & 37 & 38 & 39 \end{bmatrix} \begin{bmatrix} 40 & 41 & 42 & 43 \\ 44 & 45 & 46 & 47 \\ 48 & 49 & 50 & 51 \\ 52 & 53 & 54 & 55 \\ 56 & 57 & 58 & 59 \end{bmatrix}$$

then

$$< 1 > \psi A = \begin{bmatrix} 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 \\ 36 & 37 & 38 & 39 \end{bmatrix}$$

$$< 2\ 1 > \psi A \ = \ < \ 44 \ \ 45 \ \ 46 \ \ 47 \ >$$

$$< 2\ 1\ 3 > \psi A \ = \ 47$$

We now introduce notation to permit us to define ψ formally. We will use $A, B, ...$ to denote an array of numbers (integer, real,

or complex). An array's dimensionality will be denoted by d_A and will be assumed to be n if not specified.

The shape of an array A, denoted by s_A, is a vector of integers of length d_A, each item giving the length of the corresponding axis. The total number of items in an array, denoted by t_A, is equal to the product of the items of the shape. The subscripts will be omitted in contexts where the meaning is obvious.

A full index is a vector of n integers that describes one position in an n-dimensional array. Each item of a full index for A is less than the corresponding item of s_A. There are precisely t_A indices for an array(due to a zero index origin). A partial index of A is a vector of $0 \leq k < n$ integers with each item less than the corresponding item of s_A.

We will use a tuple notation (omitting commas) to describe vectors of a fixed length. For example,

$$< i\ j\ k >$$

denotes a vector of length three. $<>$ will denote the empty vector.

For every n-dimensional array A, there is a vector of the items of A, which we denote by the corresponding lower case letter, here a. The length of the vector of items is t_A. A vector is itself a one-dimensional array, whose shape is the one-item vector holding the length. Thus, for a, the vector of items of A, the shape of a is

$$s_a =< t_A >$$

and the number of items or total number of components a[1] is

$$t_a = t_A.$$

The precise mapping of A to a is determined by a one-to-one ordering function, $gamma$ [2]. Scalars are introduced as arrays with an empty shape vector.

There are two equivalent ways of describing an array A:

(1) by its shape and the vector of items, i.e. $A = \{s_A, a\}$, or

(2) by its shape and a function that defines the value at every index p.

These two forms have been shown to be formally equivalent [19]. We wish to use the second form in defining functions on multidimensional arrays using their Cartesian coordinates (indices). The first form is used in describing address manipulations to achieve effective computation.

To complete our notational conventions, we assume that $p, q, ...$ will be used to denote indices or partial indices and that $u, v, ..$ will be used to denote arbitrary vectors of integers. In order to describe the i_{th} item of a vector a, either a_i or $a[i]$ will be used. If u is a vector of k integers all less than t_A, then $a[u]$ will denote the vector of length k, whose items are the items of a at positions u_j, $j = 0, ..., k-1$.

Before presenting the formal definition of the ψ indexing function we define a few functions on vectors:

$u \ +\!\!+ \ v$	catentation of vectors u and v
$u + v$	itemwise vector addition assuming $t_u = t_v$
$u \times v$	itemwise vector multiplication
$n + u$, $u + n$	addition of a scalar to each item of a vector
$n \times u$, $u \times n$	multiplication of each item of a vector by a scalar
$\iota\ n$	the vector of the first n integers starting from 0
$\pi\ v$	a scalar which is the product of the components of v
$k \ \bigwedge \ u$	when $k \geq 0$ the vector of the first k items of u) and when $k < 0$ the vector of the last k items of u
$k \ \bigvee \ u$	when $k \geq 0$ the vector of $t_u - k$ last items of u) and when $k < 0$ the vector of the first $t_u - \|k\|$ items of u

[1] We also use τa, δa, and ρa to denote total number of components, dimensionality and shape of a.

[2] There are a family of gamma functions: $\gamma_{row}, \gamma_{column}, \gamma_{regularsparse}$, etc. In this case we assume row major ordering.

DEFINITION 1. *Let A be an n-dimensional array and p a vector of integers. If p is an index of A,*

$$p\psi A = a[\gamma(s_A, p)],$$

where

$$\begin{aligned} \gamma(s_A, p) &= x_{n-1} \quad \textit{defined by the recurrence} \\ x_0 &= p_0, \\ x_j &= x_{j-1} * s_j + p_j, \quad j = 1, ..., n-1. \end{aligned}$$

If p is partial index of length $k < n$,

$$p\psi A = B$$

where the shape of B is

$$s_B = k \bigtriangledown s_A,$$

and for every index q of B,

$$q\psi B = (p \ +\!\!+ q)\psi A$$

The definition uses the second form of specifying an array to define the result of a partial index. For the index case, the function $\gamma(s, p)$ is used to convert an index p to an integer giving the location of the corresponding item in the row major order list of items of an array of shape s. The recurrence computation for γ is the one used in most compilers for converting an index to a memory address [23].

COROLLARY 1. $<> \psi\ A = A$.

The following theorem shows that a ψ selection with a partial index can be expressed as a composition of ψ selections.

THEOREM 1. *Let A be an n-dimensional array and p a partial index so that $p = q \ +\!\!+ r$. Then*

$$p\psi A = r\psi(q\psi A).$$

Proof: The proof is a consequence of the fact that for vectors u,v,w

$$(u \ +\!\!+ v) \ +\!\!+ w = u \ +\!\!+ (v \ +\!\!+ w).$$

If we extend p to a full index by $p \ +\!\!+ p'$, then

$$\begin{aligned} p'\psi(p\psi A) &= (p \ +\!\!+ p')\psi A \\ &= ((q \ +\!\!+ r) \ +\!\!+ p')\psi A \\ &= (q \ +\!\!+ (r \ +\!\!+ p'))\psi A \\ &= (r \ +\!\!+ p')\psi(q\psi A) \\ &= p'\psi(r\psi(q\psi A)) \\ p\psi A &= r\psi(q\psi A) \end{aligned}$$

which completes the proof.

We can now use *psi* to define other operations on arrays. For example, consider definitions of *take* and *drop* for multidimensional arrays.

DEFINITION 2 (take: $\bigwedge$). *Let A be an n-dimensional array, and k a non-negative integer such that $0 \leq k < s_0$. Then*

$$k \bigwedge A = B$$

where

$$s_B =< k > \ +\!\!+ (1 \bigtriangledown s_A)$$

and for every index p of B,

$$p\psi B = p\psi A.$$

(In MoA $\bigwedge$ is also defined for negative integers and is generalized to any vector u with its absolute value vector a partial index of A. The details are omitted here.)

DEFINITION 3 (reverse: Φ). *Let A be an n-dimensional array. Then*

$$s_{\Phi A} = s_A$$

and for every integer i, $0 \leq i < s_0$,

$$< i > \psi \Phi A =< s_0 - i - 1 > \psi A.$$

This definition of Φ does a reversal of the 0th axis of A.

Note also that all operations are over the 0th axis. The operator Ω [35] extends operations over all other dimensions.

Higher Order Operations

Thus far operation on arrays, such as concatenation, rotation, etc., have been performed over their 0th dimensions. We introduce the higher order binary operation Ω, which is defined when its left argument is a unary or binary operation and its right argument is a vector describing the dimension upon which operations are to be performed, or which sub-arrays are used in operations. The dimension upon which operations are to be performed is often called the *axis* of operation. The result of Ω is a unary or binary operation.

From Semantic to Operational Forms

Consider the expression

$$X \quad = \quad (2 \bigwedge (\Phi A)) \times (1 \bigvee (\Phi A)) \tag{1}$$

That is, take the first two planes after reversing A along the primary axis then multiply that array by the last two planes after reversing A. We want to concurrently perform both 5 by 4 multiplications. A is the array previously defined with shape $< 3\ 5\ 4 >$. Suppose also that A is in shared memory and accessible to 2 processors. First, we *Psi-Reduce* our expression to its Denotational Normal Form (DNF) or semantic normal form. This normal form can be used to prove the equivalence of two array expressions.

Get shape:

$$\begin{aligned} \rho X &= \rho(2 \bigwedge (\Phi A)) \\ &= 2 +\!\!+ ((1 \bigvee (\rho(\Phi A)))) \\ &= 2 +\!\!+ ((1 \bigvee (\rho A))) \\ &= 2 +\!\!+ < 5\ 4 > \\ &= < 2\ 4\ 5 > \end{aligned} \tag{2}$$

Now that we have the *shape*, we can Psi-Reduce.

$$\forall \quad i,j,k \quad \ni \quad 0 \quad \leq i,j,k < \quad < 2\ 4\ 5 >$$

$$< i\ j\ k > \psi X$$

$$\begin{aligned} &= < i\ j\ k > \psi(2 \bigwedge (\Phi A)) \times (1 \bigvee (\Phi A)) \\ &= < j\ k > \psi(< i > \psi(2 \bigwedge (\Phi A)) \times (1 \bigvee (\Phi A))) \\ &= < j\ k > \psi((< i > \psi(2 \bigwedge (\Phi A))) \\ &\quad \times (< i > \psi(1 \bigvee (\Phi A)))) \\ &= < j\ k > \psi((< i > \psi(\Phi A)) \times (< i+1 > \psi(\Phi A))) \\ &= < j\ k > \psi((< ((\rho A)[0] - 1 - i) > \psi A) \\ &\quad \times (< ((\rho A)[0] - 1 - (i+1)) > \psi A)) \\ &= < j\ k > \psi((< 2 - i > \psi A) \times (< 1 - i > \psi A)) \\ &= (< (2-i)\ j\ k > \psi A) \times (< (i-i)\ j\ k > \psi A) \end{aligned} \tag{3}$$

This is the DNF, or semantic normal form. To proceed further we must know more about the data, i.e. layout, and architecture.

Applying the Psi Correspondence Theorem: PCT

We now want to *transform* the DNF above to a *Generic* Operational Normal Form (ONF) or way to *build the code*. Recall that γ maps a *full* index , $\vec{i}$, of an arbitrary array A to an offset from the start of A, $@A$ in memory, denoted by $a[\gamma(\vec{i}; (\rho A))]$. The PCT [30], generalizes the mapping function gamma, γ, and turns *short* indices into start, stops, and strides. Note that we still use *Psi-Reduction* but now just on vectors. We start by analyzing the shape of the result in conjunction with the indicies involved in the computation and observe that the indices for j and k are contiguous allowing us to only use the primary axis index, a short index, in the mapping. Thus, let Y denote $(< 2 - i > \psi A) \times (< 1 - i > \psi A)$. Then the shape of Y, i.e.

$$\rho Y = 1 \bigvee (\rho A) = 1 \bigvee \ < 3\ 5\ 4 >=< 5\ 4 > \tag{4}$$

i.e. the bounds for j and k above. Continuing with the PCT to create the *generic* ONF on p=2 processors, we use the index of the primary axis, i, $\ni \quad 0 \leq i < p$, since $\rho X =< 2\ 4\ 5 >$. Using γ_{row} for row major ordering, Y is transformed into:

$$a[((\gamma(< 2 - i >; < 1 \bigwedge (\rho A >))) \times \pi(1 \bigvee (\rho A))) + \iota\pi 1 \bigvee \rho A]$$

$$\times a[((\gamma(< 1 - i >; < 1 \bigwedge (\rho A >))) \times \pi(1 \bigvee (\rho A))) + \iota\pi 1 \bigvee \rho A]$$

Reducing, we get

$$\begin{aligned} &a[((\gamma(< 2 - i >; < 3 >))) \times \pi(< 5\ 4 >) + \iota\pi < 5\ 4 >] \\ \times &a[((\gamma(< 1 - i >; < 3 >))) \times \pi(< 5\ 4 >)) + \iota\pi < 5\ 4 >] \\ &= a[((2-i) \times 20) + \iota 20] \\ &\times a[((i-1) \times 20) + \iota 20] \end{aligned} \tag{5}$$

which transforms the indexing into start, stride, and stop, a universal hardware description and easiliy transformed to the mnemonics used at any hardware level [18].

The *iota* operation depicts 20 sequential accesses. If we wanted to map these operations to other than scalar registers we could conceptually add more dimensions to analyze prefetching, buffer sizes, vector registers, caches, etc. Suppose in this example we have two vector registers to do the addition, both with length 4, we could abstract this operation into a 5 by 4 where we can now analyze the cost of 5 vector register loads and stores and determine if that is more cost effective than 20 scalar operations. Thus prior to code generation we *could* analyze which menomnic to map to from the ONF.

3. Overview of Pull arrays

The functional programming community has recently gained an increased interest in high performance parallel array programming [3, 5, 9, 22, 24, 25, 43]. One of the central abstractions in this line of work is *pull arrays*, also know as delayed arrays. The can be defined as follows (we use Haskell [26] to demonstrate functions programs).

```
data Pull a = Pull { ixf    :: (Int -> a)
                   , length :: Int
                   }
```

Arrays are represented as functions from index to element. We refer to this function as an *index function*. Additionally arrays also have a length.

This representation has several advantages:

- *Parallelism*. Since each element is computed independently, they can easily be computed in parallel.

```
map :: (a -> b) -> Pull a -> Pull b
map f (Pull ixf l) = Pull (f . ixf) l

zipWith :: (a -> b -> c)
        -> Pull a -> Pull b -> Pull c
zipWith f (Pull ixf1 l1) (Pull ixf2 l2)
  = Pull (\i -> f (ixf1 i) (ixf2 i)) (min l1 l2)

take :: Int -> Pull a -> Pull a
take n (Pull ixf l) = Pull ixf (min n l)

drop :: Int -> Pull a -> Pull a
drop n (Pull ixf l)
  = Pull (\i -> ixf (i + n)) (max (l - n) 0)

rotate :: Int -> Pull a -> Pull a
rotate k (Pull ixf l)
  = Pull (\i -> ixf ((i + k) `mod` l)) l

reverse :: Pull a -> Pull a
reverse (Pull ixf l) = Pull (\i -> ixf (l - i - 1) l

enumFromTo :: Int -> Int -> Pull Int
enumFromTo f t = Pull (\i -> i + f) (t - f + 1)
```

Figure 1. Example functions on pull arrays

- *Fusion*. When functions on pull arrays are composed, the intermediate arrays can be automatically removed.
- *Compositionality*. Pull arrays provide a wealth of compositional and highly reusable combinators. These can be composed together to write high-level programs which typically are easier to understand than monolithic array processing code.

Figure 1 shows some examples of functions on Pull arrays. Things to note are that most functions are polymorphic; they can operate on arrays of any type of element.

3.1 Fusion

As mentioned, one of the most important advantages of Pull arrays is that they support fusion[4, 14, 22]. Fusion is a program transformation which removes intermediate data structures. Fusion supersedes transformations such as loop fusion in optimizing compilers for imperative languages. Pull arrays make it particularly simple to achieve fusion. It simply amounts to inlining the definitions of functions.

As an example of fusion consider the example from the section which was used to illustrate ψ-reductions. Here is how it is written using Pull arrays for the one-dimensional case:

```
zipWith (*) (take 2 (reverse a)) (drop 1 (reverse a)).
```

Conceptually there are many intermediate arrays here, but they will be removed by fusion. Fusion will proceed by inlining the definition of the functions as follows:

```
zipWith (*) (take 2 (reverse a))
            (drop 1 (reverse a))
  => { arr }
zipWith (*) (take 2 (reverse (Pull ixf l))
            (drop 2 (reverse (Pull ixf l)))
  => { reverse * 2 }
zipWith (*) (take 2 (Pull (\i -> ixf (l - i - 1)) l))
            (drop 1 (Pull (\i -> ixf (l - i - 1)) l))
  => { take , drop }
zipWith (*) (Pull (\i -> ixf (l - i - 1)) (min 2 l))
            (Pull (\i -> ixf (l - i)) (max (l - 1) 0))
  => { zipWith }
Pull (\i -> ixf (l - i - 1) * ixf (l - i))
  (min (min 2 l) (max (l - 1) 0))
```

The result is a single Pull array, the intermediate array has been removed. The result is the same in MoA, modulo syntax and the fact that we here use a one-dimensional array of unknown size.

The correctness of fusion relies crucially on the language being *pure*. If side effects were allowed in the indexing function fusion would not be correct.

Fusion effectively makes Pull arrays *virtual* in that they will not be represented in memory. All intermediate arrays will be removed. But sometimes it is important to keep an array in memory [3], not least to produce the final result. For this reason, Pull arrays provide a function, variously known as `force`, `memorize` or `store`. We will refer to it as `store` here. The functional semantics of `store` is the identity function, i.e. it simply returns the same array as it was given. However, as a side effect it stores the array to memory.

A subtle point about `store` is that the language is still pure with this function. The reason is that there is no way to observe from within the language whether an array is stored to memory or not. Therefore, the side-effect of `store` cannot affect the value of a computation.

3.2 Implementing Fusion

There are two different ways of implementing Pull arrays in the literature. The simplest one comes from using Pull arrays in an embedded language, i.e. when the array language is implemented as a library in another host language. Examples of such languages are Pan[11], Feldspar[5], Obsidian[43] and Nikola[25]. By using the technique of implementing Pull arrays as *shallow embeddings* on top of a *deeply embedded* code language, fusion comes completely for free [42]. In the array library Repa[22], compiler supported rewrite rules are used to implement fusion [20].

3.3 Higher dimensions

The first uses of Pull arrays used a fixed number of dimensions, typically either one or two. However, the Repa library introduced a new expressive notion of higher dimensionality for Push arrays [22]. The number of dimensions of an array, referred to as the *shape*, are determined statically in the type system. The advantage of having the shape as part of the type is that any program which produces arrays with ill-defined shapes can be ruled out. A potential downside is that enforcing shapes in the type system might be overly constraining and exclude useful programs. A key innovation in Repa is the notion of *shape polymorphism* which makes it possible to write functions which accepts arrays of any shape. Below shows the definition of the type of shapes. It is a variation on how it is represented in Feldspar.

```
data Z
data i :. sh

data Shape sh where
  Z :: Shape Z
  (:.) :: Shape tail -> Int -> Shape (tail :. Int)
```

The first two data declarations introduces new types, `Z` and `:.`, without any constructors. They are used to create snoc-lists on the type level, the definition of `Shape`. The type `Shape` is a recursive data type with two constructors. The first constructor `Z` indicates that the shape has zero dimensions. The second constructor `:.` adds one dimension to another shape, and stores how many elements there are in that dimension. The type `Shape` has a type parameter which is a snoc list with one element for each dimension.

Below we show how higher dimensional Pull arrays can be defined using the `Shape` type. We also show some functions adapted

[3] Example: Indexing a column of a huge transposed array that has a few rows and many columns. In this case, non-materialization may have long strides causing many cache misses and page faults.

for the new representation. Note the shape polymorphism in the definitions of `map` and `zipWith`. They can work with arrays of any shapes, as long as they are the same shape. The function `reverse` is particularly noteworthy because the type reflects that the input and output arrays need to have at least one dimension.

```
data Pull sh a = Pull (Shape sh -> a) (Shape sh)

map :: (a -> b) -> Pull sh a -> Pull sh a
map f (Pull ixf sh) = Pull (f . ixf) sh

zipWith :: (a -> b -> b) ->
           Pull sh a -> Pull sh b -> Pull sh c
zipWith f (Pull ixf1 sh1) (Pull ixf2 sh2)
  = Pull (\sh -> f (ixf1 sh) (ixf2 sh2))

reverse :: Pull (sh :. Data Length) a ->
           Pull (sh :. Data Length) a
reverse (Pull ixf sh@(_ :. l)) =
 Pull (\(sh' :. i) -> ixf (sh' :. (l - i - 1))) sh
```

4. Comparison

In this section we relate MoA and Pull arrays.

An initial observation is that MoA and Pull arrays were developed for different purposes and that can be seen already by looking at the notation. Mathematics of Arrays has first and foremost been used for calculating array programs by hand. This is reflected in the short names and succinct notation which works very well when doing pencil-and-paper proofs.

Pull arrays on the other hand were conceived in the context of a typed functional language. The notation is completely inherited from the syntax of Haskell [26].

4.1 Types

When it comes to types MoA keeps things very simple. There is just one type, arrays. Scalars are arrays; their shape is the empty vector.

Pull arrays inherits the type discipline from the language they are implemented in. When used in Haskell it means that they have a rich type system to utilize. First and foremost, Pull arrays are parametric, meaning that they can carry elements of any type. In particular the elements can be Pull arrays as another way of representing higher dimensional arrays. This representation is seldom used as it has poor performance.

4.2 DNF, ψ-reductions and fusion

The correspondence between MoA and Pull arrays is most clear when considering Denotational Normal Form (DNF), ψ-reductions and fusion. Both formalisms define operations on arrays by how to index into the array. In MoA, the indexing function ψ is used, whereas for Pull arrays operations are defined as functions from index to the element at that index. These two forms are equivalent.

Pull arrays are always kept in the form of an indexing function and a length. This representation is preserved for all operations. In MoA, arrays are treated as abstract during derivations. After calculations, ψ-reductions are performed to bring the expression into Denotational Normal Form. DNF is equivalent to Pull arrays, it shows how to compute an element from an array directly from an index.

The process of applying ψ-reductions corresponds to performing fusion on Pull arrays. In both cases operations are inlined until there is a single expression for computing an element from an array given an index.

Pull arrays use the function `store` to prevent fusion and store an array in memory. MoA deals with memory differently. All arrays which are supposed to be allocated in memory are given a separate definition.

The last step in calculating a program in MoA is to derive an Operational Normal Form which is explicit about how the array is laid out in memory. Pull arrays do not have anything corresponding to ONF. Instead, the decision of how the array is laid out is done by the compiler, which uses a pre-defined convention.

MoA	Feldspar
δ	`dim . extent`
ϱ	`extent`
$\iota\xi^n$	`enumFromTo`
ψ	generalized `index`
rav	– No corresponding function
γ	`toIndex`
γ'	`fromIndex`
$s\ \hat{\varrho}\ \xi$	`reshape`
π x	`product`
τ	`size`
++	++
ξ_1 f ξ_2	`zipWith f` ξ_1 ξ_1
σ f ξ	`map f` ξ
ξ f σ	`map f` ξ
$\bigtriangleup$	`take`
$\bigtriangledown$	`drop`
op^{red}	`fold op`
Φ	`reverse`
Θ	`rotate`
$\oslash$	generalized `transpose`
$f\Omega_d\xi$	– No corresponding function

Figure 2. A correspondence between functions in MoA and Pull arrays

4.3 Higher dimensions

Shapes in Pull arrays are represented by a completely different type, whereas in MoA shapes are simply arrays. Representing shapes as arrays is challenging in a typed language such as Haskell because it leads to a form of infinite recursion and while there are a number of ways to break the recursion we have yet to find a way that does so and retains the efficiency and simplicity of Pull arrays.

Apart from the differences in the type system, MoA and Pull arrays treat shapes relatively similar. The type of shapes in Pull arrays can be seen as just another representation of arrays, one where the length is part of the type.

4.4 Functions

Figure 2 gives a translation between function names used in MoA and Pull arrays. The intention of this figure is to work as a Rosetta stone, helping to translate from one formalism to the other.

As can be seen in the figure, not all operations in MoA have a corresponding function in the Pull array literature. Some of these functions are missing because Pull arrays do not have a notion corresponding to ONF in MoA. In particular, the function rav is used in ONF to decide on the particular layout of the array, and this step is implicit for Pull arrays. However, Ω is a good candidate to transfer to the formalism of Pull arrays.

The difference between ψ in MoA and `index` i Pull arrays is that ψ can be used with partial indexes, a central notion in MoA. Pull arrays solve this problem in a more roundabout way, using a notion of *slices* [22]. We leave out the explanation of slices for reasons of space.

5. Conclusion and Future work

Mathematics of Arrays and Pull arrays are a lot more closely related than what they first might appear. In this paper we have outlined the relationship between fusion and ψ-reduction, Pull arrays and Denotational Normal Form, and the various types of operations used. Our hope is that this paper will help researchers from both communities interact and learn from each other.

We have identified a few points where MoA and Pull arrays can learn from each other. The treatment of higher dimensions in MoA is in general more thorough than Pull arrays. There are operations like Ω which haven't appeared in the literature on Pull arrays but which would provide a very useful addition.

Pull arrays has a dual representation: Push arrays [9]. It can deal with some of the shortcomings of Pull arrays, in particular more efficient concatenation and sharing of computations between array elements. It would be interesting to see how the ideas of Push array carries over to MoA.

References

[1] L. M. R. Mullin, A Mathematics of Arrays, Ph.D. Thesis, Syracuse University, December 1988.

[2] P. Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970.

[3] J. Ankner and J. D. Svenningsson. An edsl approach to high performance Haskell programming. *SIGPLAN Not.*, 48(12):1–12, Sept. 2013. ISSN 0362-1340.

[4] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

[5] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar – an embedded language for digital signal processing. In J. Hage and M. T. Morazán, editors, *IFL*, volume 6647 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2010. ISBN 978-3-642-24275-5.

[6] K. Berkling. Arrays and the lambda calculus. Technical report, CASE Center and School of CIS, Syracuse University, 1990.

[7] T. Budd. A parallel intermediate representation based on Lambda expressions. *Arrays, Functional Languages and Parallel Systems*, 1991.

[8] T. A. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984. ISSN 0164-0925.

[9] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded gpu kernel programming language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, pages 21–30, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1117-5.

[10] L. Coffin. Designing a new programming methodology for optimizing array accesses in complex scientific problems, 1994. UMR Undergraduate Research Program(OURE).

[11] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[12] L. M. et al. The PGI–PSI project: Preprocessing optimizations for existing and new F90 intrinsics in HPF using compositional symmetric indexing of the Psi calculus. In M. Gerndt, editor, *Proceedings of the 6th Workshop on Compilers for Parallel Computers*. Forschungszentrum Julich GmbH, 1996.

[13] S. Gerhart. *Verification of APL Programs*. PhD thesis, CMU, 1972.

[14] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232. ACM, 1993.

[15] L. J. Guibas and D. K. Wyatt. Compilation and delayed evaluation in APL. In *Conference Record of the Fifth Annual ACM Symposium o n Principles of Programming Languages*, pages 1–8, Jan. 1978.

[16] G. Hains and L. Mullin. Parallel functional programming with arrays. *The British Computer Journal of the British Computer Society: the society of information systems engineering*, 36(22), 1993.

[17] A. Hassitt and L. Lyon. Efficient evaluation of array subscripts of arrays. *IBM Journal of Research and Development*, January 1972.

[18] H. B. H. III, L. R. Mullin, D. J. Rosenkrantz, and J. E. Raynolds. A transformation–based approach for the design of parallel/distributed scientific software: the fft. *CoRR*, abs/0811.2535, 2008.

[19] M. Jenkins, 94. Research Communications.

[20] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[21] K.E.Iverson. *A Programming Language*. Wiley, New York, 1962.

[22] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. *ACM Sigplan Notices*, 45(9):261–272, 2010.

[23] P. Lewis, D. Rosenkrantz, and R. Stearns. *Compiler Design Theory*. Addison-Wesley, 1976.

[24] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in Haskell. *ACM SIGPLAN Notices*, 46(12):59–70, 2011.

[25] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In J. Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 67–78. ACM, 2010. ISBN 978-1-4503-0252-4.

[26] S. Marlow et al. Haskell 2010 language report. *Available onli ne http://www. haskell. org/onlinereport/haskell2010*, 2010.

[27] T. McMahon. Mathematical formulation of general partitioning of multi-dimensional arrays to multi-dimensional architectures using the Psi calculus, 1995. Undergraduate Honors Thesis.

[28] T. Miller. Tentative compilation: A design for an APL compiler. Technical Report 133, Yale University, 1979.

[29] L. Mullin. The Psi compiler project. In *Workshop on Compilers for Parallel Computers*. TU Delft, Holland, 1993.

[30] L. Mullin and M. Jenkins. Effective data parallel computation using the Psi calculus. *Concurrency – Practice and Experience*, September 1996.

[31] L. Mullin and T. McMahon. Parallel algorithm derivation and program transformation in a preprocessing compiler for scientific languages. Technical Report CSC-94-29, University of Missouri-Rolla, Dept of CS, 1994.

[32] L. Mullin and S. Thibault. Reduction semantics for array expressions: The Psi compiler. Technical Report CSC 94-05, Dept. of CS, University of Missouri-Rolla, 1994.

[33] L. Mullin, W. Kluge, and S. Scholtz. On programming scientific applications in SAC – a functional language extended by a subsystem for high level array operations. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages, Bonn/Germany*, 1996.

[34] L. Mullin, E. Rutledge, and R. Bond. Monolithic compiler experiments using C++ Expression Templates. In *Proceedings of the High Performance Embedded Computing Workshop HPEC 2002*, MIT Lincoln Laboratory, Lexington, MA, 2002.

[35] L. M. R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, December 1988.

[36] L. R. Mullin. Psi, the indexing function: A basis for FFP with arrays. In *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1991.

[37] L. R. Mullin. A uniform way of reasoning about array–based computation in radar: Algebraically connecting the hardware/software boundary. *Digital Signal Processing*, 15:466–520, 2005.

[38] L. R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for scheduling, routing and communication protocol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*. IEEE Computer Society, July 1993.

[39] H. Najafzadeh and L. R. Mullin. A dimension independent general partitioning algorithm to support HPF directives. In *Proceedings of the Seventh Workshop on Compilers for Parallel Computers, CPC98*, Linkoping, Sweden, 1998.

[40] H. Pottinger, W. Eatherton, J. Kelly, L. Mullin, and T. Schifelbein. Hardware assits for high performance computing using A Mathe matics of Arrays. In *Proceedings of INEL94*, 1994.

[41] H. Pottinger, W. Eatherton, J. Kelly, L. Mullin, and R. Ziegler. An FPGA based reconfigurable coprocessor board utilizing A mathematics of Arrays. In *Proceedings of the IEEE Circuits and Systems Sympos ium ISCAS95*, May 1995.

[42] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for edsl. In *Trends in Functional Programming*, pages 21–36. Springer, 2013.

[43] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In S.-B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 156–173. Springer, 2011. ISBN 978-3-642-24451-3.

[44] J. Sylvester. Lectures on the principles of universal algebra. In *American Journal of Mathematics: VI*, volume 4. reprinted in Mathematical Papers, 1884.

[45] H.-C. Tu. *FAC: A Functional Array Calculator and it's Applicaton to APL and Functional Programming*. PhD thesis, Yale University, 1985.

[46] H.-C. Tu and A. J. Perlis. FAC: A functional APL language. *IEEE Software*, 3(1):36–45, Jan. 1986.

[47] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical computer science*, 73(2):231–248, 1990.