Title Text

Subtitle Text, if any

Name1
Affiliation1
Email1

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

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

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. The name *Psi* was inspired by Dirac's *Psi Vectors* and his communications with Einstein to have a mathematical way to reason about them. These results began with the awareness that Iverson's APL[16] was inspired by Sylvester's Universal Algebra[40] and the fact that he and Cayley invented matrices. The programming language, APL, embodied Ken's notation into an interpretive computing environment. It was at this point that theoretical anamolies were introduced.

Not only did Ken believe that APL's array algebra was the ideal basis for a programming language, he (and others) believed that an ideal machine language supported array operations. This led to HP building an APL machine based on Phil Abram's dissertation[2]. Abrams was the first to identify ways to define an algebra of arrays based on shapes; others attempted to put closure on his work[12, 14]. Abrams believed that if array operations were based on indexing with shapes, all indexing in the array expression could be composed and thus could minimize the creation (and deletion) of temporary arrays, a huge overhead. One of the reasons HP's APL Machine did not succeed was that this analysis was done at run time. It was Alan Perlis, and later others[7, 23], who realized that these optimizations should be done at compile time. Attempts at compilation led Perlis to realize that anomalies in the language [11] made it difficult to optimize and impossible to verify correctness of programs leading him to recommend a functional array calculator based on the lambda calculus[41, 42]. This awareness led to Mullin's collaborations with Klaus Berkling to combine MoA, Psi Calculus, and the Lambda Lalculus, theoretically an ideal enviornment[5] and basis for a functional programming language with arrays[6, 13, 24, 27, 28, 31].

MoA and the Psi Calculus, as a theory, have also been used to describe and verify hardware[33, 37, 38], describe the partitioning and scheduling of array operations (and their costs)[9, 10, 22, 26, 29, 34, 35], and the formulation of parallel and distributed processing[32]. 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). The ONF describes how to *build* the code using *start, stop, stride*, and *count*: a universal machine abstraction. This is possible by *abstractly* lifting the dimension of an expression to include all aspects of a processor, memory, communication topology. For example, to map a 2-d array expression to n processors we would lift the dimenion to 3-d.

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

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

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

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

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

Most of the common array manipulation operations found in languages like Fortran90, Matlab, ZPL, etc., can be defined from ψ and a few elementary vector operations.

We now introduce notation to permit us to define ψ formally and to develop the *Psi Correspondence Theorem* [25], which is central to the effective exploitation of MoA in array computations. 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 which is also sometimes written as Θ.

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, $gamma$. Although the choice of ordering is arbitrary, it is essential in the following that a specific one be assumed. By convention we assume the items of A are placed in a according to the lexicographic ordering of the indices of A. This is often referred to as $row\ major\ ordering$. Many programming languages lay out the items of multidimensional arrays in memory in a contiguous segment using this ordering. Fortran uses the ordering corresponding to a transposed array in which the axes are reversed *column major*. 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 [15]. 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.

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

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
$k \ \bigtriangleup \ u$	when $k \geq 0$ the vector of the first k items of u,(called *take*) and when $k < 0$ the vector of the last k items of u
$k \ \bigtriangledown \ u$	when $k \geq 0$ the vector of $t_u - k$ last items of u,(called *drop*) and when $k < 0$ the vector of the first $t_u - \|k\|$ items of u
$k\ \theta\ u$	when $k \geq 0$ the vector of $(k \bigtriangledown u) +\!\!+ (k \bigtriangleup u)$ and when $k < 0$ the vector or $(k \bigtriangleup u) +\!\!+ (k \bigtriangledown u)$

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 [18].

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

$$k \bigtriangleup A = B$$

where

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

and for every index p of B,

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

(In MoA $\bigtriangleup$ 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 [30] extends operations over all other dimensions.

Example

Consider the evaluation of the following expression using the 3 by 5 by 4 array, A, introduced in Section 2.

$$< 1 \;\; 2 > \psi(2 \bigtriangleup \Phi A) \tag{1}$$

where A is the array given in the previous section. The shape of the result is

$$
\begin{aligned}
& 2 \bigtriangledown s_{(2 \bigtriangleup \Phi A)} \\
= \;& 2 \bigtriangledown (< 2 > +\!\!+ (1 \bigtriangledown s_{\Phi A})) \\
= \;& 2 \bigtriangledown (< 2 > +\!\!+ (1 \bigtriangledown s_A)) \\
= \;& 2 \bigtriangledown (< 2 > +\!\!+ < 5\ 4 >) \\
= \;& 2 \bigtriangledown < 2\ 5\ 4 > \\
= \;& < 4 > .
\end{aligned} \tag{2}
$$

The expression can be simplified using the definitions:

$$
\begin{aligned}
& < 1\ 2 > \psi(2 \bigtriangleup \Phi A) \\
= \;& < 1\ 2 > \psi\Phi A \\
= \;& < 2 > \psi(< 1 > \psi\Phi A) \\
= \;& < 2 > \psi(< 3 - 1 - 1 > \psi A) \\
= \;& < 1\ 2 > \psi A
\end{aligned} \tag{3}
$$

This process of simplifying the expression for the item in terms of its Cartesian coordinates is called *Psi Reduction*. The operations of MoA have been designed so that all expressions can be reduced to a minimal normal form [30]. Some MoA operations defined by *psi* are found in Fig.1.

Symbol	Name	Description
δ	Dimensionality	Returns the number of dimensions of an array.
ρ	Shape	Returns a vector of the upper bounds or sizes of each dimension in an array.
$\iota\xi^n$	Iota	When $n = 0$(scalar), returns a vector containing elements 0, to $\xi^0 - 1$. When $n = 1$(vector), returns an array of indices defined by the shape vector ξ^1
ψ	Psi	The main indexing function of the Psi Calculus which defines all operations in MoA. Returns a scalar if a full index is provided, a sub-array otherwise.
rav	Ravel	vectorizes a multi-dimensional array based on an array's layout($\gamma_{row}, \gamma_{col}, \gamma_{sparse}, ...$)
γ	Gamma	Translates indices into offsets given a shape.
γ'	Gamma Inverse	Translates offsets into indices given a shape.
$\vec{s}\ \widehat{\rho}\ \xi$	Reshape	Changes the shape vector of an array, possibly affecting its dimensionality. Reshape depends on layout(γ).
$\pi\vec{x}$	Pi	Returns a scalar and is equivalent to $\prod_{i=0}^{(\tau x)-1} x[i]$
τ	Tau	Returns the number of components in an array,($\tau\xi \equiv \pi(\rho\xi)$)
$\xi_l +\!\!+ \xi_r$	Catenate	Concatenates two arrays over their primary axis.
$\xi_l f \xi_r$	Point-wise Extension	A data parallel application of f is performed between all elements of the arrays.
$\sigma f \xi_r$ $\xi_l f \sigma$	Scalar Extension	σ is used with every component of ξ_r in the data parallel application of f.
$\bigtriangleup$	Take	Returns a sub-array from the beginning or end of an array based on its argument being positive or negative.
$\bigtriangledown$	Drop	The inverse of Take
$_{op}$red	Reduce	Reduce an array's dimension by one by applying op over the primary axis of an array.
Φ	Reverse	Reverses the components of an array.
Θ	Rotate	Rotates, or shifts cyclically, components of an array.
$\circlearrowleft$	Transpose	Transposes the elements of an array based on a given permutation vector
Ω	Omega	Applies a unary or binary function to array argument(s) given partitioning information. Ω is used to perform all operations (defined over the primary axis only) over all dimensions.

Figure 1. Summary of MoA Operations

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.

3. Overview of Pull arrays

The functional programming community has recently gained an increased interest in high performance parallel array programming [3, 4, 8, 17, 19, 20, 39]. One of the central abstractions in this line of work is the *pull array*, also know as delayed array. The can be defined as follows (we use Haskell [21] 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 has a length.

This representation has several advantages:

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

Figure 2. Example functions on pull arrays

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

3.1 Fusion

3.2 Higher dimensions

4. Comparison

4.1 Types

4.2 ψ reductions and fusion

4.3 Higher dimensions

4.4 Functions

MoA	Feldspar
δ	dim . extent
ϱ	extent
$\iota\xi^n$	– generalized enumFromTo
ψ	index
rav	– flattens to a one-dimensional array
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
Φ	reverse
Θ	rotate
$\circledcirc$	– generalized transpose
$f\Omega_d\xi$	– Apply f along the dimension d of ξ

4.5 Example programs

5. Related work

6. Future work

7. Conclusion

Acknowledgments

Acknowledgments.

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, 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

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

[17] 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.

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

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

[20] 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.

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

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

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

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] L. R. Mullin, H. B. H. III, D. J. Rosenkrantz, and J. E. Raynolds. A transformation-based approach for the design of parallel/distributed scientific software: the FFT. http://trr.albany.edu/documents/TR00002 (submitted to Computer Physics Communications).

[34] 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.

[35] 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.

[36] A. Perlis. Steps towards an APL compiler. Technical Report 24, Yale University, 1975.

[37] 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.

[38] 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.

[39] 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.

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

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

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