A Reduction Semantics for Array Expressions: The PSI Compiler

Lenore Mullin and Scott Thibault

CSC-94-05

March 9, 1994

Department of Computer Science
University of Missouri-Rolla
Rolla, Missouri 65401

A Reduction Semantics for Array Expressions: The PSI Compiler [*]

Lenore Mullin Scott Thibault [†]

March 9, 1994

Abstract

High Performance Computing is not only concerned with the performance of a program on a very fast processor, but also high performance on many fast processors perhaps of different speeds and topologies. Scientific computation and modeling often require real time visualizations on high resolution graphics terminals. The data structure most widely used in scientific computation is the array. Arithmetics operations and permutations are often used where its associated algebra transcends many scientific disciplines. In order to achieve high performance computing, the interfaces with which people interact with computers must be rapid and accurate with the ability to move across many platforms and adapt quickly to change. If a programming language supports an algebra useful to scientists, the programming language, and subsequent compiler, must optimize the resources of a particular machine or machines. That is, the compiled program must allocate the least amount of memory and do the least amount of arithmetic. Ideally, we'd like to minimize latency and bottlnecking and achieve load balancing of all used resources. These issues may be addressed by employing The Psi Calculus[4, 2, 5, 6, 3] as a backend compiler to scientific languages. The reduction rules of the Psi Calculus, which describe how to simplify array expressions, were recently automated in the Psi Compiler[3]. This paper presents the results of this effort.

Shape Analysis

In order to perform reductions on array expressions, the shapes of all variables and partial results must be known. The only given shapes are those of the variables and are acquired while parsing the array declarations. The shape of each partial result is computed in terms of these given shapes. This process occurs from the innermost of the expression outward (a postfix order). We will see in the next section that the process of reduction occurs from the outermost inward (an infix order) [1]. So these are conceded distinct processes and require that the entire expression tree be retained.

Given that we have a complete expression tree, we can compute the shape of every node in the tree. The shape at a given node in the tree represents the resulting shape of the expression represented by the sub-tree rooted in the given node. The leaves of the tree will be variable accesses and the shape of them will be know from the variable declarations. Visiting the nodes in a postfix manner ensures that when a

[*]This work was supported by the National Science Foundation's Presidential Faculty Fellow Award Program

[†]L. Mullin, lenore@cs.umr.edu, and S. Thibault, sthib@cs.umr.edu, Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri 65401 USA

[1]Order is irrelevant since the ψ-calculus has the Church-Rosser Property.

node is visited that the shapes of it's arguments (children) have already been computed. The shape of the internal nodes is computed in terms of the shapes of it's children by the following rules[4]: [2]

Shape Calculation

- Dimension: $\rho(\delta\xi) \equiv \Theta$
- Shape: $\rho(\rho\xi) \equiv <\delta\xi>$
- Psi: $\rho(\vec{v}\psi\xi_r) \equiv (\tau\vec{v}) \bigtriangledown (\rho\xi_r)$
- Concatenation: $\rho(\xi_l +\!\!+ \xi_r) \equiv ((1 \bigtriangleup (\rho\xi_l)) + (1 \bigtriangleup (\rho\xi_r))) +\!\!+ (1 \bigtriangledown (\rho\xi_l))$
- Algebraic operators: $\rho(\xi_l \text{ op } \xi_r) \equiv \rho\xi_l \equiv \rho\xi_r$
- Scalar extension: $\rho(\sigma \text{ op } \xi) \equiv \rho\xi$
- Iota (scalar): $\rho(\iota\sigma) \equiv <\sigma>$
- Iota (vector): $\rho(\iota\vec{v}) \equiv \vec{v} +\!\!+ (\rho\vec{v})$
- Take: $\rho(\sigma \bigtriangleup \xi) \equiv <|\sigma|> +\!\!+ (1 \bigtriangledown (\rho\xi))$
- Drop: $\rho(\sigma \bigtriangledown \xi) \equiv ((1 \bigtriangleup (\rho\xi)) - |\sigma|) +\!\!+ (1 \bigtriangledown (\rho\xi))$
- Omega: $\rho(\xi_l \text{ op}\Omega_{<\sigma_l \sigma_r>}\xi_r) \equiv \vec{u} +\!\!+ \vec{v} +\!\!+ \vec{x} +\!\!+ \vec{w}$
  where

$$\begin{array}{rcl}
m & \equiv & \min\ (((\delta\xi_l) - \sigma_l), ((\delta\xi_r) - \sigma_r)) \\
\vec{u} & \equiv & (-m) \bigtriangledown ((-\sigma_l) \bigtriangledown \rho\xi_l) \\
\vec{v} & \equiv & (-m) \bigtriangledown ((-\sigma_r) \bigtriangledown \rho\xi_r) \\
\vec{x} & \equiv & (-m) \bigtriangleup ((-\sigma_l) \bigtriangledown \rho\xi_l) \\
\vec{w} & \equiv & \rho(((\vec{i} +\!\!+ \vec{k})\psi\xi_l)\text{op}((\vec{j} +\!\!+ \vec{k})\psi\xi_r)) \\
 & & \text{for} \quad 0 \leq^* \vec{i} <^* \vec{u}, \\
 & & 0 \leq^* \vec{j} <^* \vec{v},\ \ 0 \leq^* \vec{k} <^* \vec{x}
\end{array}$$

Having these rules we can compute the shape of every node in the tree. When each node has an expression for it's shape, shape analysis is complete and we can continue with the reduction of the expression.

Reduction

All array operators (excluding the algebraic operators) are available to allow the designer to shift things around and access sub-arrays of an array. If an assignment statement involves no algebraic operators and only array operators then no data is changed. It is only relocated in the resulting array. Since the data is only being relocated, no temporary variables should be required for execution of the assignment. In all previous work temporary variables are used for each array operation during the execution of an expression. The process of reduction involves reducing an array expression from one that uses complex array operators to one that only involves memory accesses. Thus we present a method of reduction to generate an efficient implementation of an array expression.

[2] The Appendix presents a brief introduction to a Mathematics of Arrays(MOA) and the psi calculus.

As an example lets look at the assignment of a MOA expression to an array.

$$A = B +\!\!+ (<1\ 2> \ \triangledown C)$$

where $\rho B = <2\ 4>$, $\rho C = <4\ 6>$ and A is conformable. It's easy to see that we can realize this assignment without any temporaries by the following expressions.

$$\begin{array}{lcll} \vec{i}\psi A & = & \vec{i}\psi B & ;\forall <0\ 0> \leq^* \vec{i} <^* <2\ 4> \\ (\vec{i}+ <2\ 0>)\psi A & = & (\vec{i}+ <1\ 2>)\psi C & ;\forall <0\ 0> \leq^* \vec{i} <^* <3\ 4> \end{array}$$

We would ideally like the compiler to be able to generate these two expressions and avoid the use of the temporary variables. This is in fact possible.

All array expressions in MOA have the property that they can be reduced to a normal form. The resulting normal form of an expression involves only memory accesses and algebraic operators. Furthermore, deriving the normal form of an expression is a mechanical process that is easily implemented on computational machines. So the normal form is indeed what we need the compiler to find, and it can be achieved mechanically.

The goal for the compiler will be to input an expression and reduce it into separate expressions (one for each variable in the expression) with the array operators eliminated. The input expression is in the form $A = B$ where B is an expression. We want each output statement to be in the form

$$\forall 0 \leq^* \vec{v} <^* \vec{b},\ \ (\vec{v} + \vec{l})\psi A = (\vec{v} + \vec{i})\psi B$$

In this expression $\vec{b}$ refers to the bound, $\vec{l}$ refers to the location of the assignment in the complete result (since the expressions are reduced to separate expression, each reduced expression represents only one part of the result), $\vec{i}$ is the top left-hand corner of the sub-array that is to be assigned and B is an array variable. If this is the output we desire and we're given an input expression $A = \xi$ then we'll start the reduction process by assuming that the reduced expression is just

$$\forall 0 \leq^* \vec{v} <^* \vec{b}, (\vec{v} + \vec{l})\psi A = (\vec{v} + \vec{i})\psi \xi$$

where $\vec{b} \equiv \rho\xi, \vec{l} \equiv^* 0$, and $\vec{i} \equiv^* 0$. This expression is correct and is in the right form. However, it may not be the reduced, normal form. If the top node of the expression tree for ξ is a variable access then this is the normal form and we're done. If the top node is an operator then the expression needs more work to find the reduced expression(s). In this case we need to eliminate the top operator by using a rewrite rule, creating one or two new expressions.

This process is probably best seen by an example. The example given earlier was

$$A = B +\!\!+ (<1\ 2> \ \triangledown C)$$

$\rho B = <2\ 4>$, $\rho C = <4\ 6>$ and A is conformable. Our starting assumption is

$$\forall 0 \leq^* \vec{v} <^* <5\ 4>, (\vec{v}+ <0\ 0>)\psi A = (\vec{v}+ <0\ 0>)\psi(B +\!\!+ (<1\ 2> \ \triangledown C))$$

. This expression is not free from array operators so it's not in normal form. We notice though that the expression can be rewritten, eliminating the $+\!\!+$, as

$$\begin{array}{llcl} \forall 0 \leq^* \vec{v} <^* <2\ 4>, & (\vec{v}+ <0\ 0>)\psi A & = & (\vec{v}+ <0\ 0>)\psi B \\ \forall 0 \leq^* \vec{v} <^* <3\ 4>, & (\vec{v}+ <2\ 0\ \ >)\psi A & = & (\vec{v}+ <0\ 0>)\psi(<1\ 2> \ \triangledown C) \end{array}$$

The first expression here is in normal form since it contains no array operators (simple ψ operations are only for memory access). The second expression is not in normal form so we continue reducing that expression. That can be done by noticing that we can rewrite this expression, eliminating the $\triangledown$, as

$$\forall 0 \leq^* \vec{v} <^* <3\ 4>, (\vec{v}+ <2\ 0>)\psi A = (\vec{v}+ <1\ 2>)\psi C$$

This expression is in normal form so now the reduction is complete and the final result is the two expressions

$$\begin{array}{lcl} \forall 0 \leq^* \vec{v} <^* <2\ 4>, \quad (\vec{v}+ <0\ 0>)\psi A & = & (\vec{v}+ <0\ 0>)\psi B \\ \forall 0 \leq^* \vec{v} <^* <3\ 4>, \quad (\vec{v}+ <2\ 0>)\psi A & = & (\vec{v}+ <1\ 2>)\psi C \end{array}$$

These are equivalent to the expressions given in the first example at the beginning of this section.

The only mystery in this process is rewriting the expression in order to eliminate the outermost operator (the top node of the sub-tree). This elimination can be systematically done based on a set of rules for each operator. Each operator is eliminated such that one or two expressions are produced. Each rule for an operator consists of an expression for computing the new bound, location, and indexes, for the one or two new expressions. The expressions in each rule are dependent only on the shapes of the arguments and the function represented by that rule. The shapes are already known from shape analysis. The rewrite rules that the compiler uses during the reduction are given in the next section.

Thus we have a deterministic way to perform the reduction of an array expression based on our previous results from shape analysis. We start with the assumption that the reduced expression is

$$\forall 0 \leq^* \vec{v} <^* \vec{b}, (\vec{v} + \vec{l})\psi A = (\vec{v} + \vec{i})\psi \xi$$

where $\vec{b} \equiv \rho\xi, \vec{l} \equiv^* 0$, $\vec{i} \equiv^* 0$, A is the result and ξ is the expression. We now examine the outermost operator, if it is not a simple variable access we apply the rule for that operator generating one or two new expressions, otherwise we are done. If we generate new expressions then we examine the outermost operator of these expressions and follow the same process in a recursive manner until no new expressions are generated. When this process is complete, we have a set of expressions that represent the reduced form of the original expression.

Reduction Rules

The compiler is built using standard techniques, plus the added function of performing shape analysis and the reduction. The shape analysis is performed as the input program is being parsed. The shape of an expression is calculated from the shapes of it's arguments by the definitions listed in the previous section. When an expression has been completely parsed it is reduced to a normal form by successively eliminating the outermost operator in the expression. The elimination is performed based on the definition of the operator. The rules used for the functions in our grammar (Figure 1) follow.

In the last section the general form that expressions were reduced to assumed that the dimensions of the arrays were equal. Thus the length of $\vec{v}$ would be the same as the length of $\vec{l}$, and $\vec{i}$. However in general this won't be true so instead the following general form is used.

$$\forall 0 \leq^* \vec{v} <^* \vec{b}, \quad (\vec{v} + \vec{l})\psi(\vec{m}\psi A) = (\vec{v} + \vec{i})\psi(\vec{n}\psi\xi)$$

In this form $\vec{l} +\!\!+ \vec{m}$ is a full index representing the location in A and $\vec{i} +\!\!+ \vec{n}$ is a full index representing the location in ξ. This allows $\tau\vec{v}$ to be smaller than the dimension of either array.

- Dim

$$\forall 0 \leq^* \vec{v} <^* \vec{b}, \quad (\vec{v} + \vec{l})\psi(\vec{m}\psi A) = (\vec{v} + \vec{i})\psi(\vec{n}\psi(\delta\xi)) \Rightarrow$$

$$\forall 0 \leq^* \vec{v} <^* \vec{b}, \quad (\vec{v} + \vec{l})\psi(\vec{m}\psi A) = (\vec{v} + \vec{i})\psi(\vec{n}\psi\xi.dim)$$

- Shape

$$\forall 0 \leq^* \vec{v} <^* \vec{b}, \quad (\vec{v} + \vec{l})\psi(\vec{m}\psi A) = (\vec{v} + \vec{i})\psi(\vec{n}\psi(\rho\xi)) \Rightarrow$$

$$\forall 0 \leq^* \vec{v} <^* \vec{b}, \quad (\vec{v} + \vec{l})\psi(\vec{m}\psi A) = (\vec{v} + \vec{i})\psi(\vec{n}\psi\xi.shape)$$

```
program := { procedure_definition }
procedure_definition := PROCEDURE_name ”(” formal_parameter_list ”)” block_body
formal_parameter_list := parameter_definition { ”,” parameter_definition }
parameter_definition := ”int” PARAMETER_name | ”array” PARAMETER_name array_definition
array_definition := ”ˆ” INTEGER_number ”<” { INTEGER_number |
INTEGER_PARAMETER_name } ”>”
block_body := ”{” definition_part statement_list ”}” definition_part := { constant_definition_part | vari-
able_definition_part | global_definition }
constant_definition_part := constant_definition ”;” | constant_scalar_definition ”;”
constant_definition := ”const” ”array” VARIABLE_name array_definition ”=” vector_constant
constant_scalar_definition := ”const” ”array” VARIABLE_name ”ˆ0 <>=” number
vector_constant := ”<” { number } ”>”
variable_definition_part := variable_definition ”;”
variable_definition := array VARIABLE_name array_definition
global_definition := ”global” VARIABL_name ”;”
statement_list := { statement ”;” }
statement := assignment_statement | for_statement | allocate_statement
allocate_statement := ”allocate” identifier variable_access
for_statement := for ”(” term ”<=” variable_access ”<” term ”)” ”{” statement_list ”}”
assignment_statement := variable_access ”=” expression
variable_access := VARIABLE_name | PARAMETER_name
expression := factor { operator expression }
operator := ”+” | ”-” | ”*” | ”/” | ”psi” | ”take” | ”drop” | ”cat” | ”pdrop” | ”ptake” | operator
”omega” constant_vector
unary_operator := ”iota” | ”dim” | ”shp” | ”+ red” | ”- red” | ”* red” | ”/ red” | ”tau” | ”rav”
factor := term | ”(” expression ”)” | unary_operator factor
term := variable_access | constant_vector
variable_access := identifier;
identifier := letter { letter | digit | ’_’ }
constant_vector := ”<” { number } ”>”
```

Figure 1: The MOA compiler input language specification

- Psi

$$\forall 0 \le^* \vec{v} <^* \vec{b},\ \ (\vec{v}+\vec{l})\psi(\vec{m}\psi A) = (\vec{v}+\vec{i})\psi(\vec{n}\psi(\vec{j}\psi\xi)) \Rightarrow$$
$$\forall 0 \le^* \vec{v} <^* \vec{b}',\ \ (\vec{v}+\vec{l}')\psi(\vec{m}'\psi A) = (\vec{v}+\vec{i}')\psi(\vec{n}'\psi\xi)$$

where $\vec{b}', \vec{l}', \vec{m}', \vec{i}'$, and $\vec{n}'$ are defined by

$$\begin{array}{rcl} \vec{b}' & \equiv & \vec{b} \\ \vec{l}' & \equiv & \vec{l} \\ \vec{m}' & \equiv & \vec{m} \\ \vec{i}' & \equiv & \vec{i} \\ \vec{n}' & \equiv & \vec{j} +\!\!+ \vec{n} \end{array}$$

- Take

$$\forall 0 \le^* \vec{v} <^* \vec{b},\ \ (\vec{v}+\vec{l})\psi(\vec{m}\psi A) = (\vec{v}+\vec{i})\psi(\vec{n}\psi(\vec{j} \bigtriangleup \xi)) \Rightarrow$$
$$\forall 0 \le^* \vec{v} <^* \vec{b}',\ \ (\vec{v}+\vec{l}')\psi(\vec{m}'\psi A) = (\vec{v}+\vec{i}')\psi(\vec{n}'\psi\xi)$$

where $\vec{b}', \vec{l}', \vec{m}', \vec{i}'$, and $\vec{n}'$ are defined by

$$\begin{array}{rcl} \vec{b}' & \equiv & \vec{b} \\ \vec{l}' & \equiv & \vec{l} \\ \vec{m}' & \equiv & \vec{m} \\ \vec{x} & \equiv & ((\vec{j}<0)(\vec{j}+((\tau\vec{j}) \bigtriangleup \rho\xi)) + (\tau\vec{j} \bigtriangleup \vec{i})) +\!\!+ ((\tau\vec{j}) \bigtriangledown \vec{i}) \\ \vec{i}' & \equiv & (\tau n) \bigtriangledown \vec{x} \\ \vec{n}' & \equiv & (\tau n) \bigtriangleup \vec{x} \end{array}$$

- Drop

$$\forall 0 \le^* \vec{v} <^* \vec{b},\ \ (\vec{v}+\vec{l})\psi(\vec{m}\psi A) = (\vec{v}+\vec{i})\psi(\vec{n}\psi(\vec{j} \bigtriangledown \xi)) \Rightarrow$$
$$\forall 0 \le^* \vec{v} <^* \vec{b}',\ \ (\vec{v}+\vec{l}')\psi(\vec{m}'\psi A) = (\vec{v}+\vec{i}')\psi(\vec{n}'\psi\xi)$$

where $\vec{b}', \vec{l}', \vec{m}', \vec{i}'$, and $\vec{n}'$ are defined by

$$\begin{array}{rcl} \vec{b}' & \equiv & \vec{b} \\ \vec{l}' & \equiv & \vec{l} \\ \vec{m}' & \equiv & \vec{m} \\ \vec{x} & \equiv & ((\vec{j}>0)(\vec{j}) + ((\tau\vec{j}) \bigtriangleup \vec{i})) +\!\!+ ((\tau\vec{j}) \bigtriangledown \vec{i}) \\ \vec{i}' & \equiv & (\tau n) \bigtriangledown \vec{x} \\ \vec{n}' & \equiv & (\tau n) \bigtriangleup \vec{x} \end{array}$$

- Cat

$$\forall 0 \le^* \vec{v} <^* \vec{b},\ \ (\vec{v}+\vec{l})\psi(\vec{m}\psi A) = (\vec{v}+\vec{i})\psi(\vec{n}\psi(\xi_l +\!\!+ \xi_r)) \Rightarrow$$
$$\forall 0 \le^* \vec{v_l} <^* \vec{b_l},\ \ (\vec{v_l}+\vec{l_l})\psi(\vec{m_l}\psi A) = (\vec{v_l}+\vec{i_l})\psi(\vec{n_l}\psi\xi_l),$$
$$\forall 0 \le^* \vec{v_r} <^* \vec{b_r},\ \ (\vec{v_r}+\vec{l_r})\psi(\vec{m_r}\psi A) = (\vec{v_r}+\vec{i_r})\psi(\vec{n_r}\psi\xi_r)$$

where $\vec{b_l}, \vec{l_l}, \vec{m_l}', \vec{i_l}, \vec{n_l}, \vec{b_r}, \vec{l_r}\vec{m_r}, \vec{i_r}$, and $\vec{n_r}$ are defined by

$$\begin{array}{rcl}
\vec{b'} & \equiv & [(1 \bigtriangleup (((\rho\vec{n})\widehat{\rho}1) +\!\!+ \vec{b}))\lfloor((1 \bigtriangleup \rho\xi_l) - (1 \bigtriangleup (\vec{n} +\!\!+ \vec{i})))] +\!\!+ (1 \bigtriangledown (((\rho\vec{n})\widehat{\rho}1) +\!\!+ \vec{b})) \\
\vec{b_l} & \equiv & ((\vec{b}'[0] > 0)(((\tau\vec{n}) \bigtriangledown \vec{b}')[0])) +\!\!+ ((\tau\vec{n} + 1) \bigtriangledown \vec{b}') \\
\vec{l_l} & \equiv & \vec{l} \\
\vec{m}_l & \equiv & \vec{m} \\
\vec{x_l} & \equiv & ((1 \bigtriangleup \rho\xi_l)\lfloor(1 \bigtriangleup (\vec{n} +\!\!+ \vec{i}))) +\!\!+ (1 \bigtriangledown (\vec{n} +\!\!+ \vec{i})) \\
\vec{i_l} & \equiv & (\tau n) \bigtriangledown \vec{x_l} \\
\vec{n_l} & \equiv & (\tau n) \bigtriangleup \vec{x_l} \\
\vec{b_r} & \equiv & ((1 \bigtriangleup \vec{b}) - (1 \bigtriangleup \vec{b_l})) +\!\!+ (1 \bigtriangledown \vec{b}) \\
\vec{l_r} & \equiv & ((1 \bigtriangleup \vec{l}) + (1 \bigtriangleup \vec{b_l})) +\!\!+ (1 \bigtriangledown \vec{l}) \\
\vec{m_r} & \equiv & \vec{m} \\
\vec{x_r} & \equiv & ((1 \bigtriangleup (\vec{n} +\!\!+ \vec{i})) - (1 \bigtriangleup \vec{x_l})) +\!\!+ (1 \bigtriangledown (\vec{n} +\!\!+ \vec{i})) \\
\vec{i_r} & \equiv & (\tau n) \bigtriangledown \vec{x_r} \\
\vec{n_r} & \equiv & (\tau n) \bigtriangleup \vec{x_r}
\end{array}$$

- Omega (binary)

$$\forall 0 \leq^* \vec{v} <^* \vec{b},\ \ (\vec{v} + \vec{l})\psi(\vec{m}\psi A) = (\vec{v} + \vec{i})\psi(\vec{n}\psi(\xi_l \,\mathrm{op}\Omega_{<\,\sigma_l\sigma_r\,>}\xi_r)) \Rightarrow$$

Forall $(h \bigtriangleup (\vec{n} +\!\!+ \vec{i})) \leq^* \vec{j} <^* (h \bigtriangleup (\vec{b_0} + (\vec{n} +\!\!+ \vec{i})))$

$\forall 0 \leq^* \vec{v} <^* \vec{b}',\ \ (\vec{v} + \vec{l}')\psi(\vec{m}'\psi A) = (\vec{v} - \vec{i}')\psi(\vec{n}'\psi[(((((\delta\xi_l) - \sigma_l) \bigtriangleup \vec{j})\psi\xi_l)\mathrm{op}((((\delta\xi_r) - \sigma_r) \bigtriangleup \vec{j})\psi\xi_r)])$

where $m, h, d_1, \vec{b_0}, \vec{b}', \vec{l}', \vec{m}', \vec{i}'$, and $\vec{n}'$ are defined by

$$\begin{array}{rcl}
m & = & ((\delta\xi_l) - \sigma_l)\lfloor((\delta\xi_r) - \sigma_r) \\
h & \equiv & ((\delta\xi_r) - \sigma_r + (\delta\xi_l) - \sigma_l - m \\
d_1 & = & \delta[\xi_l \,\mathrm{op}\Omega_{<\,\sigma_l\sigma_r\,>}\xi_r] \\
\vec{b_0} & \equiv & ((d1 - (\tau\vec{b}))\widehat{\rho}1) +\!\!+ \vec{b} \\
\vec{b}' & \equiv & h \bigtriangledown \vec{b_0} \\
\vec{l}' & \equiv & h \bigtriangledown (\vec{m} +\!\!+ \vec{l}) \\
\vec{m}' & \equiv & (h \bigtriangleup (\vec{m} +\!\!+ \vec{l})) - (h \bigtriangleup (\vec{n} +\!\!+ \vec{i})) \\
\vec{i}' & \equiv & h \bigtriangledown (\vec{n} +\!\!+ \vec{i}) \\
\vec{n}' & \equiv & \Theta
\end{array}$$

The Compiler Implementation

A preliminary compiler has been implemented to realize these reduction techniques. The compiler was written in the C language and can be used on several platforms. These platforms include networks of Sun, Next, and SGI workstations, and the Connection Machine CM-5. The compiler excepts an input program of array expressions written in MOA notation. The compiler compiles the input program directly into C output. The input language is described in figure 1. Notice that all that remains after the psi reductions is the manipulation of address pointers for B and C.

```
test(array A^2 <5 4>, array C^2 <4 6>)

{
  const array B^2 <2 4>=<1 2 3 4 1 2 3 4>;

  A=B cat (<1 2> drop C);
}
```

Figure 2: An example input file for the MOA compiler

The procedure definition provides an interface to C programs by allowing integers and pointers to arrays to be passed as parameters to the MOA procedure.

Now we can write a procedure *test* to implement the expressions we used in the reduction section. The input program for our example is shown in figure 2. The resulting C output from compiling the *test* procedure is shown in figure 3.

Conclusion

Our decomposition and mapping technologies to one or more processors also employs the Psi Calculus [2, 5, 6]. The techniques employed in [5] were extended and scaled to a CM-5, a topic of our most recent paper[6]. Our recent efforts are to interface scientific languages such as Fortran 90 and to benchmark our performance with other Fortran 90 compilers. We are also decomposing problems using PVM. We will determine if our automatic techniques compare to PVM's performance and versatility. Future work is to support heterogeneous processing and load balancing.

```
#include <stdlib.h>
#include ”moalib.e”
test(float _A[], float _C[])
{
  int i0;
  int i1;
  int step1[2];
  int step2[2];
  int shift,offset;
  float _B[]={1.000000, 2.000000, 3.000000, 4.000000, 1.000000, 2.000000, 3.000000, 4.000000};
  shift=0*4+0;
  offset=0*4+0;
  for (i0=0; i0¡2; i0++) {
      for (i1=0; i1¡4; i1++) {
          *(_A+shift)=*(_B+offset);
          shift++;
          offset++;
      }
  }
  shift=2*4+0;
  offset=1*6+2;
  step1[0]=2;
  for (i0=0; i0¡3; i0++) {
      for (i1=0; i1¡4; i1++) {
          *(_A+shift)=*(_C+offset);
          shift++;
          offset++;
      }
      offset+=step1[0];
  }
}
```

Figure 3: The output of the MOA compiler for the example

References

[1] Gaétan Hains and Lenore M.R. Mullin. An algebra of multidimensional arrays. Technical Report 782, Université de Montréal, 1991. (0) Submitted to the 12th Conference on the Foundations of Software Technology and Theoretical Computer Science, 18-20 December 1992, New Delhi, India (1) Presented at the 2nd Montreal Workshop on Programming Language Theory - Algebraic and Logical Approaches in Programming Languages, December 1991.

[2] L. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for the scheduling, routing and communications protocol of a logarithmic scan on a message passing distributed operating system: Pram algorithms revisited. Technical Report CSEE/92/07-10, University of Vermont, Dept of CSEE, 1992. Under Review, IEEE Transactions on Parallel and Distributed Computing.

[3] L. Mullin and S. Thibault. The psi compiler project: Backend to massively parallel scientific programming languages. In *Fourth International Workshop on Compilers for Parallel Computers*, 1993.

[4] Lenore M. Restifo Mullin. *A Mathematics of Arrays.* Ph.D. dissertation, Syracuse University, December 1988.

[5] L.R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for scheduling and communication protocol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, july 1993.

[6] L.R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for portable, scalable, scheduling and communication protocol. Technical Report CSC 94-04, University of Missouri-Rolla, 1994. submitted toProceedings of the Third International Symposium on High Performance Distributed Computing.

Appendix A: Introduction to the Psi Calculus

The definitions that follow are all that is needed to follow the proofs within this paper. For full definitions and typing see[4, 1]. Angle brackets $< >$ and square brackets [ ] are used to help visualize the dimensionality of an array. For example, 3 is a scalar hence no brackets are used. If we wanted to denote a one component vector we'd say $< 3 >$, a one row one column matrix by [3], etc. Multiple brackets show dimensionality, i.e. [[2 3 4]] illustrates a 1 by 1 by 3 array. We call this information shape or form and denote it by $< 1\ 1\ 3 >$. Scalars are denoted by lower case alphabetics or σ, vectors by lower case alphabetics with an arrow above and higher dimensional arrays by capital letters or ξ. $< >$ and Θ denote the empty vector. A superscript denotes the dimensionality. All objects in the ψ calculus are arrays, e.g. scalars are 0-dimensional arrays, vectors are 1-dimensional arrays, matrices are 2-dimensional arrays, etc..

Definition 1 *The unary prefix operation δ returns the dimensionality of an array.*

$$\delta A^n \equiv n$$

.

Definition 2 *An array, A has a shape, denoted by ρA, which is a vector whose entries are the lengths of each of A's dimensions.*

$$\rho A \equiv < s_0, \ldots, s_{(\delta A)-1} >$$

where the s_j's are non-negative integers, $0 \leq j < (\delta A) - 1$

Definition 3 *The total number of components of an array, τA, is defined whenever*

$$\tau A \equiv \pi(\rho A) \equiv \prod_{i=0}^{(\delta A)-1} (\rho A)_i$$

Definition 4 *Concatenation of two vectors $\vec{x} +\!\!+ \vec{y}$ is defined whenever*

$$\rho \vec{x} +\!\!+ \vec{y} \equiv < (\tau \vec{x}) + (\tau \vec{y}) >$$

and $\forall i$, s.t. $0 \leq i < ((\tau \vec{x}) + (\tau \vec{y}))$

$$(\vec{x} +\!\!+ \vec{y})[i] \equiv \begin{cases} \vec{x}[i] & \text{if } 0 \leq i < (\tau \vec{x}) \\ \vec{y}[i - (\tau \vec{x})] & \text{if } (\tau \vec{x}) \leq i < ((\tau \vec{x}) + (\tau \vec{y})) \end{cases}$$

Definition 5 *Mapping valid index vectors to lexicographic coordinates is done using γ. If we assume $\tau \vec{i} \equiv \tau \vec{s} \equiv j$. γ is defined whenever*

$$\begin{aligned} \gamma(< > ; < >) &\equiv 0 \\ \gamma(< i > ; < s >) &\equiv i \\ \gamma(< i_0 \cdots i_{j-1} > ; < s_0 \cdots s_{j-1} >) &\equiv i_{j-1} + (s_{j-1} \times \gamma(< i_0 \cdots i_{j-2} > ; < s_0 \cdots s_{j-2} >)) \end{aligned}$$

noting that $0 \leq^* \vec{i} <^* (\rho A) \Rightarrow 0 \leq \gamma(\vec{i} ; (\rho A) < (\tau A)$.

Definition 6 *The reshape operator $\widehat{\rho}$ restructures an array having the same contents to a new shape or* form. *Given a vector $\vec{v}$ containing non-negative integers, $\vec{v}\,\widehat{\rho}\, A$ is well defined providing $\pi \vec{v} \equiv \pi < \rho A >$.*

Definition 7 *The* `rav` *operator flattens an array into a vector having the same content s.t. $\rho\, \texttt{rav}\, A \equiv < \pi(\rho A) >$.*

Definition 8 *Take $\triangle$ and drop $\bigtriangledown$ are needed to define the indexing function ψ. Given n a non negative integer s.t. $0 \le n < (\tau\vec{v})$*

$$\rho(n \triangle \vec{v}) \equiv < n >$$
$$\rho n(\bigtriangledown\vec{v}) \equiv < (\tau\vec{v}) - n >$$

and $\forall i, j$ s.t. $0 \le i < n$ and $0 \le j < (\tau\vec{v}) - n$

$$(n \triangle \vec{v})[i] \equiv \vec{v}[i]$$
$$(n \bigtriangledown \vec{v})[i] \equiv \vec{v}[n+1]$$

We define the ψ function to take any valid index of an array and return a scalar or subsection of an array, e.g. a row or plane. We showed in [1] how ψ was defined inductively on its first argument as follows:

Definition 9 *The indexing function ψ is defined whenever*

- $\Theta\psi A \equiv A$
- *Given: $\vec{i}$ is a valid index vector, $\vec{s}$ is an array shape and $\vec{c}$ denotes the components of an array with shape $\vec{s}$.*

$$\psi \equiv \lambda\vec{i}\lambda\vec{s}\lambda\vec{c}.\ \textbf{if}\ [\tau(\vec{i}) \ = 0,\ \vec{s}\,\widehat{\rho}\,\vec{c}\,, \psi(1 \bigtriangledown \vec{i}, 1 \bigtriangledown \vec{s}, (\pi 1 \bigtriangledown \vec{s}) \triangle (\vec{i}[0] \times (\pi 1 \bigtriangledown \vec{s})) \bigtriangledown \vec{c})]$$

- *The shape or* form *of $\vec{i}\psi A$, is defined whenever*

$$\rho(\vec{i}\psi A) \equiv (\tau\vec{i}) \bigtriangledown (\rho A)$$

Definition 10 *Index generation on scalars or vectors is defined whenever*

$$\rho(\iota n) \equiv < n > \qquad \rho(\iota\vec{v}) \equiv \vec{v} +\!\!+ (\rho\vec{v})$$

and $\forall i$ s.t. $0 \le i < n$ and $\forall\vec{i}$ s.t. $0 \le^* \vec{i} <^* \vec{v}$

$$(\iota n)[i] \equiv i \qquad \vec{i}\psi(\iota\vec{v}) \equiv \vec{i}$$

Definition 11 *$\xi_l R^* \xi_r$ is defined whenever R is a scalar relation, and $\rho\xi_l \equiv \rho\xi_r$. $\xi_l R^* \xi_r$ is true iff*

$$\forall\vec{i}\ s.t.\ 0 \le^* \vec{i} <^* \rho\xi_l,\ \ (\vec{i}\psi\xi_l)R(\vec{i}\psi\xi_r)$$

is true. Furthermore if σ is a scalar, we write $\sigma R^* \xi$ to mean, for all valid $\vec{i}$, $\sigma R(\vec{i}\psi\xi)$. This is also equivalent to $\xi R^* \sigma$.

Definition 12 $a\lfloor b \equiv min(a, b)$