The PSI Compiler v0.4 for MOAL to C
User's Guide

Psi Compiler Project
c/o Dr. Lenore R. Mullin
Department of Computer Science
University of Missouri-Rolla 65401
lenore@cs.umr.edu

May 28, 2018

Contents

1 Introduction

The PSI Compiler is an array compiler that performs algebraic reductions on array expressions. The reductions eliminate unnecessary computation and temporary storage. The PSI Compiler is independent of the source or destination language. Thus, there has been development of different versions of the compiler that support different languages. This document describes the MOAL to C version of the PSI Compiler. Work is under way to translate subsets of FORTRAN90, High Performance FORTRAN (HPF), and SETL into our input grammar for use with our compiler. The mathematical techniques used by the compiler are described in the technical report "A Reduction Semantics for Array Expressions: The PSI Compiler", which is also in this distribution and available via ftp from cs.umr.edu:tech-reports/94-05.ps. Both the C and HPF versions of the compiler are available via Mosaic at http://sunserver.cs.umr.edu/ psi/psi.html

The compiler is currently available for the SUN, and SGI architectures. The target language of the compiler is ANSI C, so you will need to have an ANSI C compiler available to you. The *bin* directory of this distribution contains the compiler executable. This directory needs to be added to the PATH environment variable or moved to a directory that already is in the PATH. In order to compile the C programs that are generated by the compiler the header files in the *include* directory and library files in the *lib* directory need to be in directories that are searched by your C compiler. The *bin* and *lib* directories have *sun* and *sgi* sub-directories containing the same files for the two different architectures.

This distribution has been installed on the Computer Science Department's file system at UMR. In order to use the compiler on your Unix account you will need to add /usr/local/psi/bin/sun to your PATH variable for the Sun workstations or /usr/local/psi/bin/sgi for SGI workstations. Also, when compiling the C output from the compiler you need to include the options -I/usr/local/psi/include and -L/usr/local/psi/lib/sun for the Sun workstations and -L/usr/local/psi/lib/sgi for the SGI workstations. There are some example programs in /usr/local/psi/examples.

2 The Language

The source language is MOAL (Mathematics of Arrays Language) since it is derived from MOA[1], The compiler's input grammar is given by the following BNF description.

```
Moa Compiler 0.3 input grammar specification, MOAL.

Uppercase words indicate a type of an object.

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 := "^" INTEGER_number "<" { INTEGER_number |
INTEGER_PARAMETER_name } ">"
block_body := "{" definition_part statement_list "}"
definition_part := { constant_definition_part | variable_definition_part |
global_definition }
```

[1] MOA is described in the technical report

```
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 "^0 <>=" number
vector_constant := "<" { number } ">"
variable_definition_part := variable_definition ";"
variable_definition := array VARIABLE_name array_definition
global_definition := "global" VARIABLE_name ";"
statement_list := { statement ";" }
statement := assignment_statement | for_statement | allocate_statement |
procedure_call
procedure_call := PROCEDURE_name "(" actual_parameter_list ");"
actual_parameter_list := variable_access { "," variable_access }
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
reduction_operator := "+" | "-" | "*" | "/"
unary_operator := "iota" | "dim" | "shp" | reduction_operator "red" | "tau" |
        "rav"
factor := term  | "(" expression ")" | unary_operator factor
term := variable_access | constant_vector
variable_access := identifier;
identifier := letter { letter | digit | '_' }
constant_vector := "<" { number } ">"
```

2.1 Procedure Definitions

An input file consists of one or more procedure definitions. The following is an example of a procedure definition with an empty statement body.

```
test(int n, array A^2 <n n>)

{
}
```

Each procedure definition consists of the procedure name, argument list and a procedure body. The argument list is a list of declarations separated by commas. Argument declarations may be either integer or array declarations. The procedure body contains a definition part that contains object declarations and a statement part.

Integer arguments are declared with the keyword "int" followed by the name of the argument. Integer arguments may only be used in inline vectors and are provided for the purpose of indicating the size of an array in an array declaration or the loop bounds of a loop statement. Integer arguments are not intended for computation. The array argument declaration consists of the keyword "array" followed by the argument name, the dimension, and the shape (or size of the array). The dimension is specified by a

"^" symbol followed by an integer. The shape is an inline vector that may include integers and previously declared integer arguments. The inline vector is a whitespace separated list of elements enclosed in angle brackets.

2.2 The Definition Part

The definition part of a procedure contains declarations and attributes of data objects. The only type of object that can be declared is an array object. The array object definition declares the name of the object and defines the structure of the array by it's dimension and shape (size). An example array declaration is

```
array A^2 <4 3>;
```

This declares the array A which has 2 dimensions with 4 rows and 3 columns. The last vector in the declaration is called the shape and is sometimes written ρ. An array can be declared with a dynamic shape in two ways, with an integer parameter or a delayed binding. If n is an integer parameter to the enclosing procedure than it can be used in the shape of a declaration giving the array a dynamic shape. The second method can be accomplished be leaving the shape part of the declaration empty. For example

```
array A^3;
```

declares an array with a delayed shape binding. The shape of the array must be bound to the array in the statement part of the procedure before it is used. The "allocate" statement described in section 2.3 is used for delayed binding.

A constant array may be declared and initialized by preceding the "array" keyword with the "const" keyword and including initial data after the shape part. In this case the declaration must contain a shape part and may not reference integer arguments. The initial data is defined by an "=" symbol followed by an inline vector containing the components of the array in **row major order**. For example

```
const array A^2 <2 2>=<3 1 2 4>;
```

declares the constant array

$$A \equiv \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}$$

If the target architecture is a multi-processor architecture (see the "arch" directive in section 3.1) then the definition part may contain property commands. Currently there is only one, the global property. This property indicates that an array is to be entirely stored on each processor (i.e. not distributed). By default an array will be distributed if the target architecture is a multi-processor architecture. The global property is indicated by the keyword "global" followed by the array name and a ";". For example

```
global A;
```

In the future property statements will allow the specification of an array's distribution (e.g. by rows or columns, block or cyclic).

2.3 The Statement Part

There are three types of statements: assignments, control flow, and allocation statements.

There is one allocation statement that is used to specify the delayed shape binding of an array. The statement consists of the keyword "allocate" followed by the name of the array to bind the shape to, the shape, and a ";". The shape can be specified by an inline vector or an array variable of dimension one. For example,

```
allocate A b;
```

provided b is an array variable of one dimension with the same number of elements as the dimensionality of A.

The two control flow statements are the procedure call and for statement. The procedure call statement allows a call to a procedure defined in the current file or any other C file that you will later link together. The heat equation implementation in the examples directory show how this is used to call a procedure written in C to print out the results of the computation. The syntax is the name of the procedure followed by the list of actual parameters in parentheses and ";". The actual parameters are restricted to named variables (i.e. no expressions). For example

```
printout(A);
```

The for statement allows the use of iteration. The syntax of this command is

```
for ( vector1 <= array_name < vector2 ) {
statement list
}
```

where vector1 and vector2 are either inline vectors or one dimensional array variables, array_name is a one dimensional array variable, and statement list is a list of statements. This construct operates by executing the statement body repeatedly. The vector denoted by array_name is initially equal to vector1 and is incremented at each iteration until the vector is no longer bounded by vector2. For example

```
for (<0 0> <= t < <3 2>) {
  A=B;
}
```

would execute $A = B$, 6 times with t successively equal to $< 0\ 0 >, < 0\ 1 >$, $< 1\ 0 >, < 1\ 1 >, < 2\ 0 >$, and $< 2\ 1 >$.

The assignment statements consist of array expressions written in MOA notion. This is discussed in the next section.

2.4 Array Expressions

A Mathematics of Arrays (MOA) is an array algebra for manipulating and proving properties about array expressions. This section is an informal operational view of some of the operators in MOA. Details about the algebra and the formal definitions of the operators can be found in the included technical report.

2.4.1 Pointwise Operations

The computational operators include +,-,*, and / (divide). These operators work with two arguments of the same shape and perform the operator pointwise. These operators also work between scalars and arrays.

2.4.2 Reduction

There is also a corresponding reduction operator for each of these computational operators. The reduction operator performs the operation between all the elements of the first dimension "reducing" it to an array of one dimension less. These operators have the form *op* red. For example,

$$+red \begin{bmatrix} 1 & 3 & 2 \\ 5 & 9 & 2 \\ 5 & 2 & 3 \end{bmatrix}$$

sums the rows of the array; the results would be

$$< 11\ 14\ 7 >$$

2.4.3 Structural Information, Partitioning and Restructuring

Operators that return information about an array's structure include dim, shp, and tau. Each of these operators takes an array argument. The dim operator returns the dimension of it's argument, shp returns the shape of the array as a vector, and tau returns the total number of the components in the array.

The remaining operators are for manipulating arrays and perform no computation with them. These operators include $psi, take, drop, cat, iota, rav$ and $omega$. The language also includes $pdrop$, and $ptake$ but these are just restricted versions of $drop$ and $take$. To aid in the discussion of these operators we first define two functions that are not part of the language but are defined in MOA. Since compilers store the components of an array in a linear address space, they need to use an indexing function that accepts an index of a multi-dimensional array with it's shape and returns the linear address of the corresponding component stored in memory. The γ function defined in MOA is such a function. For example, given $< 2\ 1 >$ an index of A where $\rho A \equiv < 3\ 4 >$ is the 9^{th} component (starting from 0) in the linear stored array, $\gamma(< 2\ 1 >, < 3\ 4 >) = 9$. The inverse function γ' is also defined and for the example $\gamma'(9, < 3\ 4 >) = < 2\ 1 >$.

In order to better understand these operators the following discussion refers to figures that pictorially explain the examples. The figures show which components of the arrays on the right hand side of an assignment are assigned to which components of the array on the left hand side. The components that are assigned as a result of the assignment are shaded and the arrow indicates the direction of the assignment. The figures depict the assignment for arguments of a particular shape, not the general case.

The symbol "=" serves as the assignment operator and takes a left and right argument. The right argument is any valid expression of the language. The left hand side can take two forms, the first is a simple name denoting an array object and the second an expression of the form

$$v \text{ pdrop } A$$

where v is an array object of dimension 1 and A is an array object. These two forms are discussed with the discussion of the $take$ and $drop$ operators.

2.4.4 Take and Drop

The $take$ and $drop$ operators take a vector expression as left argument and an array expression as right argument. They operate as indicated by the name by taking or dropping off parts of an array. The left argument of the $take$ operator indicates how many components of each dimension to take from the origin of the right argument. Part 1 of figure 1 shows the assignment

$$A = < 3\ 2 > \text{ take } B$$

Figure 1: Examples of the take operator

which takes the first three rows and the first two columns of the three rows. The length of the left argument can be less then the dimensionality of the right argument but not greater than. Part 2 of the figure shows a "short" left argument or one with a length strictly less then the dimensionality of the right argument. The assignment in part 2 of the figure is

$$A =< 2 > \text{take } B$$

which takes the first two rows.

The *drop* operator is illustrated in figure 2. Part 1 shows the assignment

$$A =< 3\ 2 > \text{drop } B$$

This drops off the first three rows and then the first two columns. The *drop* operator also takes short left arguments like *take*. Part 2 and 3 illustrate the two forms the left side of an assignment can take. The first form, with just a simple name to denote an array object, has an implied *ptake*[2] operator in front of it, and it's left argument is the shape of the array denoted by the simple name. This means that if the right hand side is a "smaller" array than the left hand side, then the assignment is to the same size sub-array of the left hand side at the origin. Part 2 of the figure illustrates this with the assignment

$$A =< 1\ 1 > \text{drop } B$$

2.4.5 Ptake and Pdrop

The second form contains an explicit *pdrop* operator that indicates assignment to a different location then the origin. Part 3 of the figure shows the assignment

$$< 1\ 1 > \text{pdrop } A =< 2\ 1 > \text{pdrop } B$$

This assignment also utilizes the implicit *ptake* so that the assignment is made to

$$< 1\ 2 > \text{ptake } (< 1\ 1 > \text{pdrop } A)$$

Both the *take* and *drop* operators can take left arguments with all positive components or all negative components. If the components are negative then instead of taking or dropping from the origin the

[2]The ptake and pdrop operators are equivalent to the take and drop operators with the restriction that the components of the left argument may not be negative. These operators are included in the language because they produce more efficient code than the take and drop operators.

Figure 2: Examples of the drop operator

Figure 3: Examples of the psi operator

Figure 4: Examples of the cat and omega operators

components are taken or dropped from far most corner. So $< -2 \ -1 >$ drop B drops of the last two rows and the last column instead of the first. The assignment

$$A =< -2 \ -1 > \text{drop } B$$

is equivalent to the assignment

$$A =< 3 \ 2 > \text{take } B$$

for the example in part 1 of figure 1.

2.4.6 Psi

The *psi* function is the index function of the language and is implemented with the use of the γ function. The left argument of *psi* is the index vector expression and the right hand side an array expression. Part 1 of figure 3 shows the indexing of a single component of an array. The assignment of part 1 is

$$A =< 1 \ 1 > \text{psi } B$$

The *psi* operator also takes short vector arguments as in the operators *take*, and *drop*. Part 2 of the figure illustrates the use of a short index vector in the assignment

$$A =< 2 > \text{psi } B$$

2.4.7 Concatenation

The *cat* operator performs the concatenation operation on two array expressions. The concatenation is done over the primary axis (first dimension) as with all the operators discussed thus far. The assignment

$$A = B \text{ cat } C$$

is depicted in part 1 of figure 4. In the figure B and C have the shape $<\ 1\ 3\ >$.

2.4.8 Omega

Since these operators all use the primary axis as a base, the *omega* operator is provided to operate with a different axis. The operator takes four arguments, a left and right array expression, an operator, and a partition vector. The partition vector must be a constant inline vector with two components. The first component specifies how to partition the left array expression and the second the right array expression. During the operation of the omega operator the operator argument is performed one or more times on sub-arrays of the left and right array expressions. The sub-arrays used are the sub-arrays indicated be the partition vector. If the left array expression is an array of shape $< 3\ 4\ 2 >$ and the first component of the partition vector is 1 then the sub-arrays considered are the sub-arrays of the last dimension. They are the 12 vectors of two components in the array. If the first component of the partition vector was 2 then the sub- arrays would be the last 2 dimensions or the 3 4x2 sub-arrays of the array.

If there are the same number of sub-arrays on both the right and left side then the operator argument is applied to each pair of corresponding sub-arrays of the left and right array expressions. If there is only one sub-array on one side and more than one on the other then the one sub-array is used repeatedly with each of the sub-arrays of the other side. If the number of sub-arrays on each side are not equal and neither have just one then the expression is undefined and invalid. The expression is also invalid if the sub-arrays do not meet the requirements of the operator argument that is being applied. The results of the operations with the sub-arrays are concatenated together to produce the final result. Part 2 of figure 4 illustrates a concrete example of the operation with the assignment

$$A =< 0 > \text{psi omega} < 1\ 1 > B$$

In the omega expression *psi* is the operator argument that is applied to sub-arrays of $< 0 >$ and B. The partition vector indicates that sub-arrays of the last dimension of $< 0 >$ are to be used, these are vectors and there is only one of them, $< 0 >$. The partition vector indicates that sub-arrays of the last dimension of B are also to be used. Since B is a 3x3 array, there are three of these and they are vectors of length three. Since there is one sub-array on the left and three on the right the one on the left is used repeatedly three times with the sub-arrays on the right. This results in indexing the first component of the rows of B, and concatenating them together. This assignment is equivalent to the assignment

$$A = (< 0 > \text{psi } (< 0 > \text{psi } B)) \text{ cat } (< 0 > \text{psi } (< 1 > \text{psi } B)) \text{ cat } (< 0 > \text{psi } (< 2 > \text{psi } B))$$

for the shapes given in the figure. *omega*'s operation and definition is complex and further details should be found in the included technical report and "Mathematics of Arrays" referenced in the report.

2.4.9 Flattening an array - rav

The *rav* operator is used to access an array as it is stored in memory, row major order, as a linear address space. This operator takes one argument, the array to be "raveled". For example, the expression

$$\text{rav } A$$

has a shape equal to $<$ tau $A>$ and

$$< i > \text{psi (rav } A) \equiv \gamma'(i, \rho A) \text{ psi } A$$

For the following array

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$< i >$ psi (rav A) $= i$ for all valid indexes i.

2.4.10 Iota

The *iota* function is used to create an array that is useful for indexing other arrays. It takes one argument that is a vector. *iota* v creates an array that contains all the valid indexes for an array with a shape v. The array is created such that

$$i \text{ psi (iota } v) \equiv i$$

where i, and v are vectors of equal length. The shape of (iota v) is (v cat $<$ tau $v >$). For example, iota $< 5 >$ is equivalent to the array

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

with the shape $<$ 5 1 $>$.

3 Using the Compiler

3.1 General Use

Compiling programs for sequential machines is simple and can be followed from the example makefiles. The command line format is

```
mc sourcefile [options] [-o | -r | -g ]
```

Where options is one or more of the following

-o filename This specifies a C code output filename.

-g filename This specifies a generic code output filename. The generic code is still in MOAL but all expressions have been reduced.

-r filename This specifies a reduction transcript filename. It provides a transcript of the reductions that are performed on each expression.

If the -o, -g or -r appears at the end of the command then it may not appear in options and it's output will be directed to the screen. For example

```
mc mm.m -o mm.c
```

Will generate C code for the MOAL program mm.m and output it to the file mm.c.

```
mc mm.m -r
```

will generate a list of the reductions performed and display them on the screen.

The C program can then be compiled and linked with other C modules with a ANSI C compiler. Since the language does not provide any I/O capability the generated C program is not very useful unless it's linked with a C driver module that gets input, calls the MOAL routine, and performs the desired actions with the output. When linking output from this compiler the *moautil* library needs to be included to provide some utility functions that the generated C programs use.

3.2 Use on Distributed Memory Architectures

To compile a program for a multi-processor architecture some compiler directives need to be included in the MOAL source file. Compiler directives appear as the first and only thing on a line. Currently available compiler directives include *procs*, *arch*, and *hostname*. They each take a single argument and appear as

```
$directive argument
```

in the source file. The *procs* directive takes an integer argument that indicates the number of processors the target program is to use. The *arch* directive takes an argument naming the architecture to compile the program for. Currently, the compiler only supports the architectures *network*, *cm*5, and *cm*5*b*.

The *picl* directive can be used to monitor the performance of programs generated for multi-processor architectures. Currently, this is only supported for the *network* architecture. The argument to the *picl* directive is a base filename used by the generated program to write a trace file. When you run your compiled program it will generate several trace files with the base filename you specified that contain time stamped events that occurred during the execution of your program. These trace files need to be combined into one file and purged of timing conflicts, since time stamps are generated with distributed clocks which can not be synchronized. The program maketrf will load these files, resolve clock conflicts and output one sorted trace file in PICL format. Run the program as

```
maketrf basename*.ntr basename.trf
```

maketrf will load all the files with basename and the extension .ntr and create the output file with base name and extension .trf. The output of this program may then be used with ParaGraph, a performance analysis tool. The maketrf program is only available for the Sun architecture. ParaGraph can be obtained from netlib.

Programs compiled for the *network* architecture utilize BSD style sockets to communicate between multiple computers on a network. If the *network* architecture is used, the *hostname* directive must also be used. The argument of this directive names the C output file name for the host program. The host program generated for network programs initializes all the "node" programs and sends the initial data to them. This means that your driver program that passes data **in** should be linked with the host program. Data that is **output** from the procedures will be distributed on the "node" programs and the node drivers need to do what is necessary with this data. As you can imagine this is a pain and not transparent at all. In future versions this will and has to be fixed. The end of the section describes the data partitioning that is used and will help you figure out where your output is. The host program should be linked with the *moahost* library and likewise the node program with the *moanode* library.

In order to run a distributed program on a network the node program must first be started on the desired computers. The host program can then be run; it looks for the file "net.conf" in the current

directory. The user should construct the *net.conf* file that contains a list of host names that can be used as nodes. There may be more names in the *net.conf* file than are needed by the program. Each line contains one name followed by the number of processes that should be used on that machine (currently this must be 1). The gethostbyname() function is used to lookup the address of that machine and the host will initiate a connection. The host attempts to connect to a node program on each named machine in the *net.conf* file. If the number of successful connections is less than the number of processors specified by the *procs* directive then the program will abort.

The *cm5* and *cm5b* architectures target Thinking Machine's CM-5. When compiling for this architecture the *hostname* directive is optional. Since the CM-5 currently supports both hostless and host/node programs, a hostless program can be compiled by not including *hostname*. The *cm5* architecture will generate a program that uses asynchronous communication where *cm5b* programs will utilize the CM-5's blocking communication primitives. Programs generated for the CM-5 utilize the CMMD v3.0 library and should be compiled appropriately as indicated in the CMMD User's Guide. There is also an example program with an appropriate makefile in the distribution.

Part of the process of generating programs for multi-processor architectures is partitioning data over the processors. Unless the *global* (See section 2.2) property has been attached to an array, the compiler will automatically partition the array over the processors. The intent of the compiler is to provide intelligent automatic partitioning but has not been included thus far. This version of the compiler always partitions arrays evenly in a block distribution over the first dimension. Future versions will allow properties to be attached to arrays. This will allow a user specified distribution. An intelligent automatic partitioner will also be included.

Please note that this version is still major number 0. The compiler is still in progress and is being distributed to familiarize the public with our research. We also hope that it will provide a useful tool for some people.