PSI Compiler
Programmer's Guide

May 28, 2018

Contents

Figure 1: Module call structure of the compiler

1 Overview

This section describes the structure and operation of the compiler in a very general sense. The compiler consists of many modules that may be grouped into several groups. Like any compiler, there is a module that drives the operation of the compiler, a lexical analyzer that converts the input into a stream of tokens, a parser that checks the syntax of the input and creates an internal representation of the input program, and code generation modules that convert the internal representation into an output program in the output language. In addition to that, the PSI Compiler contains modules that perform reductions on array expressions. Essentially this can be viewed as a source to source transformation of the internal representation and thus is invoked between parsing and code generation.

Figure **??** shows the different sections that make up the compiler and how they relate to each other. The arrows indicate that the section at the tail invokes the section at the head. The sections to the right of the parser section are invoked in order from top to bottom. Each section in the figure represents one or more C modules. The following table lists the C modules that belong to each section.

driver	**main.c** – invokes initialization routines and the parser
lexical analyzer	**lex.c** – reads and tokenizes the input file
input parser	**parse.c** – parses the input and invokes many things
shape analyzer	**moa_func.c** – computes the shape of expressions
reduction engine	**psi.c** – uses rewrite rules to reduce expressions
code generation	**code.c** – generates output code
distributed code gen.	**part.c** – partitions arrays
	dist.c – generates code for assignments of dist. arrays
auxiliary	**vect.c** – symbolic manipulation of vector expressions
	ident.c – symbolic manipulation of scalar expressions
	poly.c – symbolic manipulation of normalized polynomials
	scalar.c – converts to and from normalized polynomial forms
	values.c – identifies constants (used by **scalar.c**)
	output.c – prints expressions, in internal form, in MOAL
system (not shown)	**sys.c** – used by all for system related activity and maintaining memory management with garbage collection

The overall flow of execution of the compiler is as follows. The main module scans the command line arguments and opens appropriate input and output files. It then calls the initialization routines of all modules that require one. Then the parser is called. The MOAL parser is a recursive descent parser. The parser parses the input of the program by repetitively calling the lexical analyzer to retrieve the next token from the input stream. When a subprogram declaration has been parsed the arguments of the subprogram are defined in a symbol table and the declaration is translated to the output. When variable declarations are parsed they are defined in the parsers symbol table. When directives are parsed some global variables are set that control the behavior of the compiler. A subroutine body may contain assignments, **for** loops, or allocate statements. When assignment statements are parsed the shape analysis module is called to build an expression tree, for the right hand side of the assignment, that has a defined shape at each node. The expression is reduced by the reduction engine and the assignment statement is stored in a list of statements. When **for** loops or allocate statements are parsed they create entries in the list of statements. After the statement body has been completely parsed, the list of statements is passed to code generation. If the target architecture is a multi-processor architecture then the list of statements is used by the partitioner (part.c) to create a list of distributions for each array that is used by the distributed code generation module.

2 Programming Style

Of course it is difficult to dictate style but some issues are of importance. The man consideration is that the compiler remain a modular system. That is, procedures that conceptually belong together should be kept together. As demonstrated by the previous section, if modularity is maintained then the structure of the code can be easily explained and understood. If a module's contents can not be described in one or two sentences, it probably contains too much.

In C modules are represented by separate C files, since it has no language construct for modules. To maintain separate files in C, each file should have a corresponding file with the same name only with a .e extension. This is an export file and should contain prototypes for procedures that may be used outside the module and extern declarations for variables that may be used outside the module.

Each file contains a list of changes that have been made at the top of the file; this should be maintained. The file VERSION should have an update list of general changes made to the compiler indicating date and version.

Keep in mind that other people will have to read, understand and possible change your code. Thus it should be readable and easy to understand. Efficiency is not a major concern in this project and should not conflict with the readability of the algorithms.

3 Data Structures

Nearly all data structures are defined in the **psi.h** header file. The following sections describe the semantics of these data types. Some of the data types are used with more then one set of semantics during different stages of the compiler.

Since garbage collection is used for most of these data types, the sys.c module should be consulted before allocating objects of these types. If an object that is being maintained by garbage collection is allocated using `get_mem` (malloc), obscure side effects will occur that are difficult to trace back to the problem.

3.1 Tokens (`symbol_t`,`emit_t`)

Tokens are used to represent the input string of characters as a stream of numeric tokens which are easier to manipulate and take less space. Tokens are passed to the parser through the global variable emit which is of the type `emit_t`. `emit_t` contains two fields *index* and *arg*. *index* represents the token as specified by the `#define`'s in **parse.h**. If *index* is `NAME1` or `NUMBER1` then more information is required to identify the token. If *index* is `NAME1` then *arg*1 is an arbitrary number that will represent that name throughout the execution of the compiler. If *index* is `NUMBER1` then *arg*1 is the number that was read.

Since each name is given a unique number that remains constant every time it is read, a table of these values must be maintained. The table of values is maintained in a hash table of linked lists. The linked lists are lists of the type `symbol_t`. It contains a *name* field that identifies the string represented by the particular element of the list. The *index* field identifies the unique number that has been associated with that string. If a string is parsed that is not found in the table, a unused number is assigned to it and and a new entry in the table is created.

During initialization, the keywords of the input language are entered into the table. Since these are not identifiers, there needs to be a distinction between an entry in the table for a keyword and one for an identifier name. The distinction is made with the *isname* field of the `symbol_t` structure.

3.2 The Symbol Table

As in the lexical analyzer, the parser associates certain information with identifiers that must be retained and retrieved every time they are encountered. So the parser also maintains a symbol table. The information that is required is the information that is specified by a variable declaration. For example, the variable's type. This table is also maintained as a hash table of linked lists. The nodes of the list are of the `ident_t` type. This type is described in the next section.

3.3 Identifiers (`ident_t`)

The `ident_t` type is used in many places in the compiler and represents all known information about a particular data object. It is defined in the **psi.h** header file. The *string* field contains the actual string name of the data object. The *index* is the number that was assigned to this string by the lexical analyzer. The *real* field indicates the base type of the object. It will be `TRUE` for real numbers and `FALSE` for integers.

The *type* field unfortunately does not really indicated type but kind of data objects. Its value may take on those defined immediately preceding it the **psi.h** file. They are:

`CONSTANT` This is a constant. This may be either an integer or a float depending on the value of *real* An object of this kind will use the *value* field to represent the constant value, regardless of its type.

`VAR_FLOAT` This is a variable scalar value. No special fields are used for objects of this kind.

`CONST_ARRAY` This is a constant array. This kind uses the *array.dim* field to indiciate the dimension of the array, the *array.shp* field to indicate the shape of the array, and the *array.cnst* field to store the values of the constant array.

`EMBEDED_ARRAY` This is a constant array that has been previously assigned to some temporary variable. The *string* field will hold the name of the temporary variable not the constants name. The other field described for the `CONST_ARRAY` kind the same here. Sometimes it is necessary, during code generation, access a constant array dynamically. This means it needs to be assigned to a temporary variable. If this is done its kind is changed the `EMBEDED_ARRAY` kind so that if it needs to be accessed dynamically again, the same temporary can be used rather then create a new one and do the same assignment.

`ARRAY` This is an array variable. This kind uses the *array.dim* and *array.shp* fields the same way as the `CONST_ARRAY` kind.

`RAV` This is an array that has been built from scalar expressions. This kind uses the *array.dim* and *array.shp* fileds the same way as the `CONST_ARRAY` kind. The *array.rav* field is also used to store the elements of the array. Since these elements may be constants, scalar variables, or even expressions they are represented by the `s_expr_t` type. The `s_expr_t` represents scalar expressions.

`IOTA` Is a special kind to represent the iota constructor. Iota builds a particular array but there is no need to actually build it unless it is used somewhere. This kind represents that array and if it is encountered during code generation then the module has built in knowledge of the contents of the array constructed by iota. The *array.dim* and *array.shp* fields are still used as with the other array kinds.

The *array.rule* field can be set for any array kind and indicates a users explicit distribution for the array on a distributed system.

Currently the compiler requires the dimension of every array be known so the dimension of an array is a constant. However, in the hope that in the future a dimension may be a variable, the *array.dim* field is a pointer to an identifier. The shape of an array may be a constant or a variable so each element of the shape is a pointer to an identifier. Since the shape is a vector of such, the *array.shp* field is a pointer to an array of pointers to identifiers. The length of the array of pointers is the dimension of the array variable.

The $flags$ field is used to attach different attributes to objects. Attributes are represented by a group of bits of the $flags$ field. The different attributes represented by $flags$ are:

`GLOBAL` This object is duplicated on all processors on a distributed memory architecture. This can be set by the user with the global directive statement.

`DYNAMIC` This object has an unknown shape and must be allocated dynamically.

`USED` This object has been used in a previous assignment statement. If this is not the case and it is also not `DYNAMIC` or a `PARAMETER` then it needs to be allocated.

`HASSHAPE` This indicates that a dynamic array has been allocated somehow and can now be used in an assignment (i.e. its shape is known).

PARAMETER This object is a parameter to the subprogram currently being parsed.

CODED This object has been assigned to a variable. This is used to recognize a RAV kind object that can be accessed through a variable rather then its element wise scalar expressions.

All of these attributes can be set, reset, or checked by macros defined in the **psi.h** file.

The *next* field of `ident_t` points to the next node in a symbol table list. Since the same objects of `ident_t` type are used in expressions and in the symbol table, this field may not be used anywhere except by the symbol table procedures.

3.4 Scalar Expressions (s_expr_t)

Scalar expressions are expressions involving only scalar objects. The `s_expr_t` represents these expressions as expression trees. The *op* field represents the operation performed at that node. The possible values of *op* are `#define`'d immediately after the definition of `s_expr_t` in the **psi.h** file. If *op* is `NOP` then the *ident* field is a pointer to a variable or constant represented by this node. If *op* is a binary operator then the *left* and *right* fields point to the left and right arguments to the operator. If *op* is a unary operator then most often the *left* field will point to the argument to the operator. Some special unary operators might allow either *left* or *right* be the argument and require the other to be `NULL` in order to determine which is the correct one. *flags* represents attributes as with `ident_t`. The only attribute associated with objects of type `s_expr_t` is `CODED`. It has a similar meaning to that described for `ident_t`.

The `#define`'d values for *op* are used for scalar, vector, and array expressions, so not all of them can be used with any given kind of expression. The valid operators for expressions represented by `s_expr_t` are `NOP`, `ABS`, `IF_NEG`, `IF_POS`, `PLUS`, `MINUS`, `SKIP`, `TIMES`, `DIVIDE`, `MIN`, `MAX`, and `MOD`.

3.5 Vector Expressions (vect_t)

Vector expressions are expressions that involve only one dimensional array objects. These expressions are used to represent internal vectors described in *Array Expressions* and shape vectors. Since many of the reduction rules are described in terms of MOA operations on vectors, the MOA operations on vectors are built into the compiler and are contained in the **vect.c** module. Like `s_expr_t` these expressions are represented as expression trees. The *op* field represents the operation performed at that node. The possible values of *op* are `#define`'d immediately above the definition of `vect_t` in the **psi.h** file. If *op* is `NOP` then the *ident* field is a pointer to a variable or constant represented by this node. If *op* is a binary operator then the *left*, and *right* fields point to the left and right arguments to the operator. If *op* is a unary operator then most often the *left* field will point to the argument to the operator. Some special unary operators might allow either *left*, or *right* be the argument and require the other be `NULL` to determine which is the correct one. *flags* represents attributes as with `ident_t`. The only flag associated with objects of type `vect_t` is `CODED`. It as a similar meaning to that described for `ident_t`.

The *shp* field is used to represent the shape of the vector; this is a scalar expression. Also scalar expressions, the fields *index* and *loc* are used to indicate the index of the first element and the location of the vector in the result. The expression (2 drop some vector) will be represented with a `vect_t` that has an index of 2 and points to the vector argument. The expression (a cat b) would be represented by a tree containing a left of the root and b on the right. The location of b in the result is the shape of a, in accordance with the definition of cat. Thus the *loc* field of the b node would be equal to the shape of a.

The `#define`'d values for *op* are used for scalar, vector, and array expressions so not all of them can be used with any given kind of expression. The valid operators for expressions represented by `vect_t` are `NOP`, `ABS`, `IF_NEG`, `IF_POS`, `PLUS`, `MINUS`, `SKIP`, `TIMES`, `DIVIDE`, `MIN`, `MAX`, `CAT`, `STORE`, `RSCAN`, `SCAN`, and `MOD`.

3.6 Array Expressions (`expr_t`,`parser_t`,`forall_t`)

Array expressions are stored in expression trees using the `expr_t` type. The *op* field specifies the operator represented by the node or `NOP`, if it is a leaf of the expression tree. The `expr_t` is also used for propagating information through the expression during shape analysis and while reducing expressions. For the purpose of shape analysis, the *shp* and *dim* fields of `expr_t` represents the shape and dimension of the sub-expression represented by the sub-tree whose root is that `expr_t`.

The `EXT_OP` operator and *ext_op* and *ext_str* fields have been included to support external operations. The nodes must be created and maintained by external routines. When they are encountered during the reduction process an external routine `reduce_external` is called. Likewise, during code generation if a `reduced_t` (discussed later) is encountered with the `EXT_OP` operator then the external routine `code_external` is called.

Several `vect_t` fields are used during the reduction process. The fields are *index*, *bound*, and *loc*. They are undefined until the reduction process begins. The function `psi_reduce` performs the forward reduction. The reduction process is described in the UMR technical report "A Reduction Semantics for Array Expressions: The PSI Compiler". The first thing done is to convert an expression

$$A = \xi$$

into the equivalent form

$$\vec{b} \bigtriangleup (\vec{l} \bigtriangledown A) = \vec{b} \bigtriangleup (\vec{i} \bigtriangledown \xi)$$

where $\vec{b} = \rho\xi$, $\vec{l} = (\delta A)\widehat{\rho}0$, and $\vec{i} = (\delta\xi)\widehat{\rho}0$. This second expression is represented in the expression tree by setting $bound = \vec{b}, loc = \vec{l}$ and $index = \vec{i}$ in the top node of the tree. The forward reduction is then performed by using the reduction rules in the technical report to derive new expressions of the left and right arguments of the outermost operator, eliminating the outermost operator. The outermost operator in the tree is, of course, the top node. So the reduction rules tell us how to compute $index, bound$, and *loc* for the two arguments (left and right branches) of the top node. Then, to eliminated that operator we just move down the tree, first to the left then to the right, recursively calling `psi_reduce` to reduce the expressions that resulted from the last reduction. The *rot* field is also included for possible future use when rotate is implemented, but is currently unused.

The $left$ and *right* fields are the arguments to the operator, if any. The *ident* field is used for the NOP operator, which is used for the leaves of the expression tree. The $flags$ field marks the node with certain attributes of leaf nodes. The valid attributes are

`ALLOC` Indicates that this array has been allocated.

`SCANNED` Indicates that the *shp* field of this node is now equal to $_*\mathrm{scan}\ (\rho\xi)$ if the node represents ξ. This is used to make the computation of γ easier.

There are macros defined in **psi.h** to set,reset, and check these flags.

The $func$ field is a string representation of the operator represented by the node. This is used for convenience during code generation.

During shape analysis, omega operations are eliminated using the definition of omega. The definition involves a forall expression that must also be represented in the tree. The $left$ argument of an omega node is the expression that results from applying the definition of omega. The $forall$ field of `expr_t` is used to describe the forall expression that results from applying the definition. $forall$ is a pointer to a `forall_t` type. The fa field of `forall_t` is a new vector variable that is created for the forall expression. The *bound* field represents the bounds of the forall expression.

For historical reasons the parser does not deal directly with expressions as `expr_t` objects. Instead, there is a top level object that points to `the expr_t` object. The top level object is a `parse_t` type. The *psi* field is a pointer to the expression and the *ident* field is no longer used.

The `#define`'d values for *op* are used for scalar, vector, and array expressions, so not all of them can be used with any given kind of expression. The valid operators for expressions represented by `vect_t` are `NOP`, `PLUS`, `MINUS`, `SKIP`, `TIMES`, `DIVIDE`, `MIN`, `MAX`, `TAKE`, `DROP`, `CAT`, `PSI`, `FORALL`, `SCALAR_EXTEND`, `PTAKE`, `PDROP`, `PLUS_RED`, `MINUS_RED`, `TIMES_RED`, `DIVIDE_RED`, `RAVEL`, `ALLOCATE`, `REDUCTION`, `RED_OP`, `RESHAPE`, `ROTATE`, `REVERSE` and `MOD`.

3.7 Compound Operators (`op_t`)

The omega operator allows the user to create compound operators by allowing omega to take omega as an operator argument. The compiler internally represents operators with the `op_t` type. If the operator is not an omega operator then the *omega* field is `FALSE`, *func* is a pointer to the function in **moa_func.c** that processes the operator and *part* and *next* are `NULL`. If the operator is an omega and its operator argument is not an omega then the *omega* field is `FALSE`, *func* is a pointer to the function in **moa_func.c** that processes the operator argument of the omega, *part* is a pointer the vector partition argument of the omega and *next* is `NULL`. If the operator is an omega and its operator argument is an omega then the *omega* field is `TRUE`, *func* is `NULL`, *part* is a pointer to the vector partition argument of the omega and *next* is a pointer to the `op_t` that results from recursively applying these rules to the operator argument of the omega.

3.8 Statements (`assign_t`,`reduced_t`,`loop_t`,`statement_t`)

The `statement_t` type is used to represent any statement of the MOAL input. The *kind* field indicates the kind of statement it is. *kind* may be any of the following (which are defined in **psi.h**)

`LOOP` A loop statement represented by the *d.loop* field.

`ASSIGN` An assignment statement that has not be reduced and is represented by the *d.assign* field.

`REDUCED` A reduced assignment statement represented by the *d.reduced* field.

`DYNAMIC` An array allocation statement represented by *d.dynamic*.

`CALL` A call to a procedure whose arguments are represented in *d.dynamic* and *str* is the actual procedure call statement.

The *next* field is used to create a list of statements representing the code body of a procedure.

The *d.assign* field uses the `assign_t` type. This type has a *result* field that points to the result array and a *expr* field that points to the expression of the right hand side. The *d.loop* field is represented by the `loop_t` type. The `loop_t` type contains the fields *lower* and *upper* which are pointers to the lower and upper bound expressions. The *var* field is a pointer to the loop variable. The loop body of the loop is stored in a statement list pointed to by the *body* field.

The *d.reduced* field uses the type `reduced_t`. This type also represents different things depending on the value of the *type* field. The *type* field may have the following values defined in **psi.h**

`SKIP` This is a dummy node and does nothing.

NOP This is a reduced assignment statement consisting of the operations in the list pointed to by the *d.list* field.

`FORALL` This represents a forall loop that results from an omega expression. The first element of the list in the *d.list* field is the forall expression. The remaining elements of the list are the reduced expressions resulting from the application of the omega definition.

`ALLOCATE` This is used to allocate temporary arrays. The $d.frag$ field points to the array to be allocated.

`EXT_OP` This represents an external operator that should be coded with the external routine `code_external`. This is used for input languages that support array operations not supported by the compiler.

3.9 Distributed Arrays (`rule_t`,`dist_t`)

The `rule_t` type is used to indicate an array's desired distribution. If an explicit distribution directive is in the input program then that distribution is specified by a `rule_t` pointed to by the variable's `ident_t` object. During the partitioning process these rules are put into a list of all available rules. If no rule exists for a particular array then a default rule is created by the partitioner that distributes the array evenly over the first dimension. The *ident* field is a pointer to the identifier begin distributed. The *dist* field is either `BLOCK` or `CYCLIC` (defined in **psi.h**). The *shp* field is the shape of the array being distributed. The *part* field is the shape of the partitions. The *proc* field is the shape of the processor array that the array should be mapped to. Once all the rules are in a list, a list of final distributions is created with the `dist_t` type. Eventually the partitioner should be able to resolve any conflicting rules that appear in the initial list.

The `dist_t` type is used to create a list of distributions used by code generation to generate distributed assignments. The fields of `dist_t` have the following meaning

ident The array being distributed.

shp The shape of the array.

c The shape of the cyclic portion of the dist. mem.

g The shape of the processor array.

e The shape of the processors local sub-array.

pl The location of the processor sub-array used.

pb The shape of the processor sub-array used.

next The next distribution in the list.

3.10 Normalized Polynomials (`poly_t`)

Normalized polynomials of k variables are represented as k dimensional arrays of coefficients. These are stored in objects of `poly_t` type. The d field is the dimension of the array and the *shp* field is the shape of the array. The *rav* field is a pointer to the components of the array. Each variable x_i is represented by dimension i. The coefficient of the term $x_1^{e1} x_2^{e2} \cdots x_d^{ed}$ is the element $< e1\ e2\ \cdots ed\ >$ of the array. These are used to simplify scalar expressions and put them in a normal form to determine if two scalar expressions are equivalent.

3.11 Garbage Collection (save_t)

The save_t type is used to store a list of statements. When garbage collection is invoked every object not referenced somewhere in the list of statements, given by the save_list global variable, is returned to the heap.

3.12 Code Generation (name_t,steps_t)

During code generation many variable scopes may be created as a result of user **for** loops. Any array dynamically allocated in a particular scope needs to be deallocated at the end of that scope. For this purpose two global variables, allocs and dynamics, of the name_t type maintain a list of variables that have been allocated during the current scope. This list is used generate code for deallocations. The name_t type contains only a string name of the variable and a pointer to the next name_t in the list.

The steps_t type is used to maintain a list of steps for each variable in a given assignment. Each element of the list gives the step values as an s_expr_t for each dimension of the variable it represents. The steps for each dimension are stored as an array of pointers to s_expr_t's in the *steps* field. The *num* field is a number representing which variable that node represents.

4 Module Descriptions

4.1 lex.c

The **lex.c** module is the lexical analyzer. This module must be initialized when the compiler begins by calling lex_init and then get_symbol. At all times, the current look ahead symbol is stored in the emit global variable. The next symbol is read by calling get_symbol.

4.2 parser.c

The **parser.c** module contains a standard recursive descent parser. Any parser used with the MOA compiler must do several things. Handling distributed programs is dependent on the input language. The parser must set the global variable n_procs to be the target number of processors. If a host/node program is to be generated, then the global variable host must be set to TRUE. Whenever a procedure body is parsed the appropriate procedure header must be produced by the parser in the output language in the output file. The following variables, max_dim,max_combine,forall_num,red_num and const_array_num should be set to zero. These variables will keep track of the number of temporary variables used of different kinds. The files **tfile**,**vfile** and **hfile** (if one is needed) should be opened. Any variable declarations will be output to the **vfile**. Generated code will be output to the **tfile**. If a host program is being generated then it will be written to **hfile**. Each statement that is parsed should be stored in an appropriate statement_t object. At the end of the procedure the parser should have a complete list of the statement body. Then the partition routine is called with the statement list as input and it will return a distribution list. The statement list and distribution list is then passed to the code_c routine that generates code for the list of statements. Upon return from this procedure the parser should call declare_utils, close the mentioned files and copy them to the output as appropriate.

In order for the parser to successfully build the assignment statement it must follow the following procedure. When a variable access is encountered, psi_access should be called with the ident_t representing the variable. The routine will return a parser_t representing the expression consisting of only the variable access. When an operator is encountered, the function in **moa_func.c** for that operator is called with the arguments of the operator represented as a `parser_t` object. The procedure will return a new `parser_t` that represents the expression of the operator and its arguments. The `parser_t` for each argument is obtained by recursively using this procedure. Each time a procedure is called from **moa_func.c** the shape values of the result of the operation is computed as a function of the arguments' shapes, as indicated by their definition in MOA. This process is shape analysis. This is done for all operators except the assignment. When an assignment has been completely parsed, the parser will have a `parser_t` for the result and for the right hand side expression. These are stored appropriately in a `statement_t` object to represent the assignment. The `statement_t` object can be passed to the `psi_reduce` procedure in **psi.c** which will reduce the assignment statement.

4.3 moa_func.c

The moa_func module performs shape analysis as an expression is parsed, as described in the last subsection. There is one procedure for each MOA operator currently supported with the name `psi_` plus the name of the operator. They each take one `parser_t` argument for each argument to the operator and return a new `parser_t`.

4.4 psi.c

This module contains the main routines that perform the reduction of an array assignment. The module contains two initialization routines, `reduction_init` and `psi_init`, that must be called when the compiler is invoked. The only other procedure in this module called from outside is `psi_reduce` which does the reduction. The *array expressions* section describes how the forward reduction takes place. The expression is converted to its normal form in `psi_reduce` and then passed to the `assign` procedure. The `assign` procedure looks at the operator of the top node. If the operator is a `NOP` then there is nothing more to do and that `expr_t` is added to the reduced list by the `addto_reduced` procedure. If it is an operator then a procedure is called to apply the reduction rule for that operator by propagating new values of *bound*, *index*, and *loc* to the operator's arguments. There is one such function for each operator with the name `red_` plus the name of the operator. Finally, `assign` is recursively called with the arguments of the operators to reduce them.

When `assign` returns in `psi_reduce`, the final list of reduced expressions has been created. This list may contain many assignments, if the expression contains cat or algebraic operators. The list is created in reverse order so `reverse` is called to put them back in the proper order. Finally, a procedure not related to the reductions is called, `combine_reduced`. The `combine_reduced` procedure provides an important optimization. It searches through the list of reduced expressions and identifies the ones that have the same bound and thus can use the same **for** loops. The result of this procedure is a list of lists. Each set of reduced expressions that can be combined into one set of loops is stored in a list. Each of these sets are then put in a list. The return type is `reduced_t`. The *d.list* field of each node in the `reduced_t` list points to a list of reduced expressions that can be combined.

4.5 part.c

This is the partitioning module. The top level procedure is `partition` and accepts a statement list as an argument. The partitioner calls `rec_partition` and `resolve_rules`. `rec_partition` searches the assignment statements for all arrays that are used. If an array is used and has an explicit rule then that rule is added to a global list. If the array does not have an explicit rule, a default rule is created that partitions the array evenly over the first dimension.

When the `rec_partition` returns, a global list of all the rules for all the arrays is stored in the *rules* variable. `resolve_rules` is then called. This procedure will eventually resolve any conflicting rules and create a final distribution list. The procedure currently assumes there are no conflicts (and in fact there can't be yet) and just converts them to the final distribution form (the `dist_t` type). The list of final distributions is returned from the `partion` procedure and will be passed to the code generation module.

4.6 code.c

The code module contains the procedures required to generate code for sequential execution. `code_c` is the top level procedure in the code module which accepts a statement list and a distribution as arguments. This procedure calls `init_dist_arrays`, if the target architecture has more then one processor. `init_dist_arrays` is described in the **dist.c** section. Then `code_c_rec` is called to generate code for the statement list. After code generation is complete, if the target number of processors is more than one then `allocate_utils`, from **dist.c**, is called to allocate the utility variables used for distributed assignments.

`code_c_rec` searches through the list and calls `code_reduced` if it finds a reduced assignment. Each reduced assignment contains a list of reduced expressions that are searched by `code_reduced`. Each node in the list of reduced expressions may have the `NOP`,`FORALL`, or `ALLOCATE` values in the *type* field with the following meaning

`NOP` This type is an assignment list and the *d.list* field points to the list of expressions that can be combined. This list is combined and the expressions are coded by `code_expr`.

`FORALL` This type is a forall statement. The $d.list->d.frag$ field points to the `FORALL` `expr_t` node that contains the bounds of the forall. This node is passed to the `open_forall` and `close_forall` procedures to generated the **for** loops. The $d.list->next$ is another list of reduced expressions that belong within the scope of the forall expression. This list is coded by recursively calling `code_reduced`.

`ALLOCATE` This is an allocation for a temporary array. The $d.frag$ field is a pointer to the temporary array.

The `code_expr` procedure accepts a list of expressions that can be combined into one expression. The first thing done is to separated out any expressions that are distributed because these can not be combined even though that have the same bounds. The `seperate_reduced` procedure is called to perform the separation. The result is a list of lists similar to the one produced by `combine_reduced`. Each node of the list points to a list of expressions that still can be combined (i.e. are not distributed). All distributed expressions will become a list with one element, since they can not be combined with anything. Next, the list is search and any arrays that are not distributed, have not be allocated, and have not been used are allocated by `allocate_array`. All arrays are also marked `USED`. Finally, the list of lists is search and for each list it is passed to `global_assign` if it is not distributed or `code_dist` if it is.

`global_assign` takes a list of expressions to be coded with combined loops. The procedure uses any of the expressions to generate the **for** loops for the expression, since they must all have the same bounds in order to be combined. Then code is generated for each expression inside the same set of nested loops. The start offset for each array is computed before the **for** loops using γ (generated by the `do_shapes` and `do_location` procedures). The steps (strides) for each variable are computed by the `add_steps` procedure to create the structure described in the Code Generation section.

`code_dist` is described in the next section.

4.7 dist.c

The distribution module contains an initialization procedure *dist_init* that must be called when the compiler is invoked. The two main procedures in this module are *init_dist_arrays*, and *code_dist*. *init_dist_arrays* is called by *code_c* before any code generation is performed. This procedure will generate code to distribute initial data over the processors. The second procedure, *code_dist* generates code for a list of expressions, involving distributed arrays, that can be combined. Currently, no expressions involving distributed arrays may be combined so the list must be a list of one element. The procedure works in two phases. First *dist_lhs* is called. This procedure creates a loop that loops through all processors, say p, and at each iteration determines what part, say part1, of the array on the left hand side is owned by the processor p. Then the procedure computes what part of the right hand side is required for the assignment to part1. Call this part2. Now *dist_rhs* is called. This procedure loops through all the processor numbers, say sp and determines what part, say part3, of part2 belongs to processor sp. The expression can now be coded. If $p = sp$ =my_processor then I own part1 of the left hand side and part3 of the right hand side, so I can compute the expression (between part1 and part3). Otherwise if p =my_processor then I own part1 but not part3 so I should wait to receive something from processor sp and then use that to compute the expression. Otherwise if sp =my_processor then I own part3 but not part1 so I should send part3 to processor p. Otherwise I do not own part1 or part3 and can continue without doing anything.

4.8 vect.c

Both the shape analysis and reduction process involves many manipulations of vectors. To aid with this task, **vect.c** contains procedures that manipulate symbolic vectors represented by `vect_t` data structures. The following is an overview of the procedures in **vect.c** and a brief description of their purpose.

`make_vect` `vect_t *make_vect(expr_t *expr)`

This procedure takes an array expression in an `expr_t` and converts it to the `vect_t` representation. This is used mostly for operators that take vector arguments. If the array expression is a simple array access it can just be converted to the `vect_t` form. If it is not a simple array access then the expression is assigned to a temporary variable, so that it can be accessed as an array. This temporary variable is named `tmp_vect`n , where n is the current unused temporary vector numer. Function `assign` in **psi.c** is called to reduce this assignment and add it to the global reduced statement list, `reduced`.

`simplify_vect` `vect_t *simplify_vect(vect_t *vect)`

This simplifies a vector expression by recursively calling itself on the `left` and `right` nodes of a `vect_t` type, collapsing any constant sub-expressions that the vector expression contains, calling `const_op`, also located in **vect.c**, in order to perform any constant operations between subexpressions.

`vect_op` `vect_t *vect_op(int op, vect_t *left, vect_t *right)`

This procedure takes two vector arguments and an operator and creates a `vect_t` node with that operator and those arguments. The arguments become the left and right subtrees of the node. The result is passed to `simplify_vect` before returning it.

`vect_take` `vect_t *vect_take(vect_t *left, vect_t *right)`

This procedure performs a vector take with a positive left argument. `propogate_take` in **vect.c** is called in order to enforce the new bounds on each node of the `right` vector expression tree. Negative take is not implemented at the time of this writing, but could be implemented in the future by converting it into a drop expression.

`vect_drop` `vect_t *vect_drop(vect_t *left, vect_t *right)`

This procedure performs a vector drop with a positive left argument. `propogate_drop` and `update_locs` in **vect.c** are called in order to enforce the new bounds on each node of the `right` vector expression tree. Negative drop is not implemented at the time of this writing, but could be implemented in the future by converting it into a take expression.

`vect_cat` `vect_t *vect_cat(vect_t *left, vect_t *right)`

This procedure performs the concatenation of two vectors. A new root node is formed with CAT as the operator and the two vectors `left` and `right` as the left and right subtrees. `propogate_cat` in **vect.c** is called in order to add the shape of the left subtree to the location of the right subtree. This is in accordance with the Psi Calculus definition of concatenation, 4.10, p. 14.

`vect_unop` `vect_t *vect_unop(int op, vect_t *vect)`

This procedure takes a vector argument and an operator and creates a `vect_t` node with that operator and argument. The result is passed to `simplify_vect` before returning it.

`red_rav` `s_expr_t *red_rav(vect_t *vect, int i)`

This procedure accepts a vector `vect` and an index `i` and returns an s_expr_t that is the `i`*th* element of the vector. `red_rav` recursively calls itself on the left and right subtrees of the vector in order to build up the `s_expr_t` tree representing the `i`*th* element.

`print_scalar` `void print_scalar(vect_t *expr)`

This procedure prints the scalar value of an vector expression. This procedure can be used only when the vector expression has one element. `print_scalar` recursively calls itself in order to print out the complete scalar expression that `expr` represents.

`vect_assign` `void vect_assign(vect_t *vect, vect_t *res, char *op)`

This procedure can be used to assign a vector expression to a temporary variable so that it may be accessed as an array in the output C code. It first sets up a variable tt tmp of tt vect_t type to store the information about the temporary variable, and then calls `recursive_assign` to assign `res` to the temporary variable. `op` is normally tt =, but can also be the C operators `+=`, `-=`, `*=`, and `/=`.

`purify_vect` `vect_t *purify_vect(vect_t *vect)`

This function "purifies" a vector by insuring that it can be accessed as an array. If the input vector is an expression, it must be assigned to a temporary array in the output C code through a call to `vect_assign`. Otherwise we can just return `vect`. This function also checks to see if `vect` is the empty vector.

`vect2array` `vect_t *vect2array(char *name,vect_t *vect)`

This function converts a vector to an array. This function calls `vect_assign` to assign `vect` to the variable `name` in the output C code. If `name` is null, then a variable of the form `tmp_vect`*n* is used. This function is similary to `force_vect` except that a name can be specified and the vector is assigned to an array even if `vect` is of type `EMBEDDED_ARRAY`.

`rav_value` `double rav_value(vect_t *vect, int i, int *simple)`

This function takes a vector `vect`, an index `i`, and an integer pointer `simple` as arguments. If the $\texttt{i}^{th}$ element of the vector is a constant (i.e. can be determined at compile time), then its value is returned and `simple` is set to TRUE. Otherwise `simple` is set to FALSE and the return value is undefined.

`vect_len` `vect_t *vect_len(vect_t *vect)`

This function returns the length of vector `vect` (i.e. the shape of the vector) as a vector expression.

`tau` `void tau(FILE *fd, expr_t *expr)`

This procedure computes the product of the expression `expr` and prints the result to the specified file `fd`. It is used to compute tau of an array by passing it the shape of the array.

`force_vect`   `vect_t *force_vect(vect_t *vect)`

This function is like purify_vect except that, if a temporary assignment is made, the generated code is output directly to the code file. In purfiy_vect the code would be added to the reduced expression list.

`static_shps`   `int static_shps(vect_t *vect)`

This function returns a true value if the vector `vect` has constant shape, loc, and index values. It recursively calls itself in order to check all components of the vector expression.

`vect_comp`   `int vect_compare(vect_t *vect1, vect_t *vect2)`

This function compares two vectors to determine if they have equivalent components. The two vectors `vect1`, and `vect2` are purified (via `purify_vect`) and the expanded (via `expand_vect`), and `ident_compare` in **ident.c** is called in order to compare them.

4.9 ident.c

Like the vector module this module contains utility procedures for manipulating and generating code for scalar expressions (`s_expr_t`). The following is a list of the procedures available in **ident.c** and a brief description of each.

`make_ident`   `ident_t *make_ident(s_expr_t *s)`

This function converts a scalar expression to an `ident_t` data type. the scalar expression `s` is a simple variable access then that variable is returned. Otherwise the expression is assigned to a temporary in the output C file, and an `ident_t` for the temporary is returned.

`print_s_expr`   `void print_s_expr(FILE *outfile, s_expr_t *s)`

This procedure prints a scalar expression to a file. Opening and closing parenthesis are added unless it is a simple variable access, and the procedure `print_s` in **code.c** is called to print the actual expression. The `print_op` procedure in **code.c** is used to print operator symbols, since these are dependent on the output language.

`make_s_expr`   `s_expr_t *make_s_expr(vect_t *vect)`

This function forces a vector `vect` to be a scalar expression using a temporary variable if necessary. This function is meant for one element vectors since it only converts the first element of `vect` into a scalar expression.

`red_s_expr`   `s_expr_t *red_s_expr(s_expr_t *s)`

This function forces a scalar expression `s` to be a variable access. If the scalar expression is not a simple scalar, the expression is assignmed to a temporary variable. This function also checks to see if the scalar expression has already been assigned to a temporary variable.

`simplify_s_expr`   `s_expr_t *simplify_s_expr(s_expr_t *s)`

This function simplifies scalar expressions by collapsing constants. Also, normalizes them by converting them to normallized polynomial form and back.

`s_op`   `s_expr_t *s_op(int op, s_expr_t *left, s_expr_t *right)`

This function accepts a scalar operator `op` and two `s_expr_t` arguments `left` and `right`. It returns a scalar expression that represents `left op right`.

`s_vect`   `vect_t *s_vect(s_expr_t *s)`

This function converts a scalar expression `s` into a one element vector.

`s_compare` `int ident_compare(ident_t *ident1, ident_t *ident2)`

This function compares two `ident_t` variables `ident1` and `ident2`. How they are compared depends upon `ident1->type` and `ident2->type`. This function has case statements for all possible combinations of the two types.

`s_compare` `int s_compare(s_expr_t *s1, s_expr_t *s2)`

This function compares two scalar expressions `s1` and `s2` to determine if they are equivalent. This function recursively traverses the scalar expression tree, calling `ident_compare` to compare the leaves of the tree. The two scalar expressions must be in the same form (the purpose of using normalized polynomials).