Design Goals

Arrays in moto should work just like arrays in Java, that is arrays are objects. They should be instantiated on the heap (not the stack) and bounds checked at runtime. We should be able to create arrays of Objects or elementary types.

Design Questions

Q. Should unbalanced (jagged) arrays be allowed or even created by default ?
A. *yes, this should be the only type of multi-dimensional array we support* **-** This is the way java works. More importantly for a pure multidimensional array I believe programmers should have no problem doing the math themselves. This also means that one definition of an array with multiple dimensions results in multiple object allocations

Q. Should we bounds check at runtime ?
A. *yes*

Q. Where does the array metadata (*dimensions*, *bounds*, and *subtype*) need to get stored ?
A. *In the array object itself as meta data AND in the RefVal MotoVal* - The reasoning works like this. We know arrays need meta data because we want to bounds check at runtime. The primary problem with storing meta data with the arrays (that of byte alignment) needed to be solved for storing the length of the array anyway.

The subtype and dimension are only needed currently by the verifier and only when dealing with Array Objects (REF_TYPE). Creating a new MotoVal type would deny the fact that arrays are supposed to be Objects. Setting the MotoVal reference to some intermediate storage would make the compiler and interpreter work differently.

Q. How do we deal with functions that return arrays ?
A. Well, functions will need to be able to specify an array return type. For the time being we will simply prepend the dimensions to the String types inside the mxfn record.

Q. Should array index evaluation be short-circuited if bounds are wrong ?

Design Decisions

The string name of values and variables declared as int[] or Object[] should always be Array

One should not be allowed to declare or pass generic Array Objects

Lexer and Parser Changes

New Tokens

OPENBRACKET - “[“
CLOSEBRACKET - “]”
OPENCLOSEBRACKET - “[“([\n\r\t]*)”]”

New (in italics) or Modified Rules

```
declaration
```

```
     : NAME NAME array_declaration_list
     | type_qualifier NAME NAME array_declaration_list

function_declaration
     : NAME NAME
     | type_qualifier NAME NAME
     | NAME array_declaration_list NAME
     | type_qualifier NAME array_declaration_list NAME

embedded_function_definition_statement
     : function_declaration OPENPAREN declaration_list CLOSEPAREN
     | function_declaration OPENPAREN CLOSEPAREN

function_definition_statement
     : DEFINE function_declaration OPENPAREN declaration_list
CLOSEPAREN CLOSEPAREN statement_list ENDDEF
     | DEFINE function_declaration OPENPAREN CLOSEPAREN CLOSEPAREN
statement_list ENDDEF
     | DEFINE function_declaration OPENPAREN declaration_list
CLOSEPAREN CLOSEPAREN ENDDEF
     | DEFINE function_declaration OPENPAREN CLOSEPAREN CLOSEPAREN
CLOSEPAREN ENDDEF
     | DEFINE error CLOSEPAREN

array_declaration_list
     : array_declaration
     | array_declaration_list array_declaration

array_declaration
     : OPENCLOSEBRACKET

postfix_expression
     : subscript_expression

allocation_expression
     : NEW NAME array_index_list
     | NEW NAME array_index_list array_declaration_list

assignment_expression
     : subscript_expression
 assignment_operator assignment_expression

subscript_expression
     : postfix_expression array_index

array_index_list
     : array_index
     | array_index_list array_index

array_index
     : OPENBRACKET expression CLOSEBRACKET
```

New Nonterminals

array_declaration_list
array_declaration
array_index_list
array_index
array_subscript_expression

New Opcodes

ARRAY_DECLARATION - Specifies that a particular declaration rule parsed for an Array declaration

ARRAY_INDEX(expression) - Specifies an index of an array to be either evaluated or assigned
ARRAY_NEW - Instantiates an array
ARRAY_LVAL - Specifies that an index of an array should be assigned to
ARRAY_RVAL - Specifies that an index of an array should be evaluated

Structure of an MXArrayReference

int dim // dimensions of this array
int[10] bounds // the actual bounds of each dimension
void* type // the moto type (for now) which this array stores
void* data // a pointer to the storage of the array

MXArray types

The general idea is to create one struct for each subtype of array. Each struct will start with some meta data.

```
typedef struct arrMeta {
   int length;
   int dim;
    void* subtype;
} ArrMeta;

typedef struct booleanArr {
   struct arrMeta meta;
   char data[1];
} BooleanArray;

typedef struct charArr {
   struct arrMeta meta;
   char data[1];
} CharArray;

typedef struct intArr {
   struct arrMeta meta;
   int32_t data[1];
} IntArray;

typedef struct longArr {
   struct arrMeta meta;
   int64_t data[1];
} LongArray;
```

```
typedef struct floatArr {
   struct arrMeta meta;
   float data[1];
} FloatArray;

typedef struct doubleArr {
   struct arrMeta meta;
   double data[1];
} DoubleArray;

typedef struct refArr {
   struct arrMeta meta;
   void* data[1];
} RefArray;

typedef struct arrArr {
   struct arrMeta meta;
   union unArr* data[1];
} ArrArray;

typedef union unArr {
   struct booleanArr ba;
   struct charArr ca;
   struct intArr ia;
   struct longArr la;
   struct floatArr fa;
   struct doubleArr da;
   struct refArr ra;
   struct arrArr aa;
} UnArray;
```

MXArray functions

MXArray *arr_create (int dim, int* bounds,void* type, void* data); - Creates the mxarray

void arr_free(MXArray *arr); - Frees the mxarray

Environment Changes

Need to add a new built in subtype Array. It is a reference type

Need to add extra fields to the RefVal MotoVal to allow for Array verification

```
typedef struct refVal {
   MotoType *type;
   void* value;
   MotoType *subtype; /* Used iff this refval is an Array */
   char dim;          /* Used iff this refval is an Array */
} RefVal;
```

Modified functions

```
moto_createEnv
- Need to addBuiltInType(env, "Array");

moto_valToCType
-  else if (!strcmp(val->type->name,"Array"))
          return "mxarray*";

void*
moto_defaultCreateArray(MotoVal* val, int dim) {
  switch (val->type->kind) {
    case INT32_TYPE:
      return malloc(sizeof(int32_t)*dim);
    case INT64_TYPE:
      return malloc(sizeof(int64_t)*dim);
    case FLOAT_TYPE:
      return malloc(sizeof(float)*dim);
    case DOUBLE_TYPE:
      return malloc(sizeof(double)*dim);
    case BOOLEAN_TYPE:
      return malloc(sizeof(char)*dim);
    case CHAR_TYPE:
      return malloc(sizeof(char)*dim);
    case REF_TYPE:
      return malloc(sizeof(void*)*dim);
    default: /* FIXME: I sure hope we don't get here */
      return "NULL";
  }
}

    case INT32_TYPE:
      return ((int32_t*)data)[idx];
    case INT64_TYPE:
      return ((int64_t*)data)[idx];
    case FLOAT_TYPE:
      return ((float*)data)[idx];
    case DOUBLE_TYPE:
      return ((double*)data)[idx];
    case BOOLEAN_TYPE:
      return ((char*)data)[idx];
    case CHAR_TYPE:
      return ((char*)data)[idx];
    case REF_TYPE:
      return ((void*)data)[idx];
    default: /* FIXME: I sure hope we don't get here */
      return "NULL";
char*
moto_CArrayAllocation(MotoVal* val, int dim) {
  switch (val->type->kind) {
    case INT32_TYPE:
      return “malloc(sizeof(int32_t)*dim)”;
    case INT64_TYPE:
      return “malloc(sizeof(int64_t)*dim)”;
    case FLOAT_TYPE:
      return “malloc(sizeof(float)*dim)”;
```

```
      case DOUBLE_TYPE:
        return “malloc(sizeof(double)*dim)”;
      case BOOLEAN_TYPE:
        return “malloc(sizeof(char)*dim)”;
      case CHAR_TYPE:
        return “malloc(sizeof(char)*dim)”;
      case REF_TYPE:
        return “malloc(sizeof(void*)*dim)”;
      default: /* FIXME: I sure hope we don't get here */
        return "NULL";
    }
}
```

New functions

```
MotoVal *
moto_createArrayVal(MotoEnv *env) {
   MotoVal *val = moto_createVal(env, REF_TYPE);
   val->type = moto_getType(env, "Array");
   return val;
}

inline int
isArray(MotoVal *val);
```

psuedocode for modified function moto_fnToCPrototype

Dynamic Loader Changes

Need to change loadlibrary to initialize MXFunction dimensions for the return type and arguments

Need to modify the mxFunction structure to include a dimesion for the return type and argument types

Motod Changes

Need to prepend the dimensions of array typed arguments and return types

Verifier Changes

New functions

psuedocode for motov_array_index

1) Evaluate the expression
2) Verify the type of the expression is an int
        2.1) if not, call moto_illegalTypeForArrayIndex
3) Push an int value onto the stack

psuedocode for motov_array_lval

Same as motov_array_rval

psuedocode for motov_array_rval

1) Evaluate the first operand
2) Verify that it was declared as an Array
    2.1) If not err with moto_illegalTypeForSubscriptOp
3) Record the current position on the stack
4) Evaluate the LIST of ARRAY_INDEX elements
5) Switch on the relationship between the # of subscript operations and the arrays declared dimensions
    5.1) If the dimensions subscripted exceeds the dimensions of the array
        5.1.1) moto_illegalSubscriptDimension
        5.1.2) recover by setting the return value equal to the subtype of the specified array
    5.2) if The dimensions subscripted equals the dimensions of the array
        5.2.1) set the return value equal to the subtype of the specified array
    5.3) The dimensions subscripted are less than the dimensions of the array
        5.3.1) Create a new Array types return value with the same subtype but dimensions = original dimensions - the number of dimensions subscripted
6) Pop off and free the subscript values on the stack
7) Free the array value
8) Return the return value

psuedocode for motov_array_new

1) Grab the array subtype from the first operand
2) Make sure this type is registered
    2.1) If the type is not registered then throw a typeNotDefined error and set the subtype to object
3) Discover the array dimension by looking at the length of operands 2 and potentially 3
    3.1) Record the current position on the stack
    3.2) Evaluate the LIST of ARRAY_INDEX elements
    3.3) dim += stack_size(frame->opstack) - stacksize;
    3.4) Pop off and free the elements pushed onto the stack
    3.5) If the dimensions are not fully specified (i.e. 3 operands) then add the number of unspecified dimensions (opcount for operand 3)
4) Construct a new Array type MotoVal and push it onto the stack

Modified functions

psuedocode for modified motov_declaration

1) Extract the declaration operand (operand 0) and from that extract the variable name, type, dimension, and any qualifiers
    1.1) discover if this is an ARRAY_DECLARATION by checking the declarations operator
    1.2) get the type or subtype from operand 0 of the declaration
    1.3) get the variable name from operand 1 of the declaration
    1.4) if this is an *array declaration* get the dimensions by counting the number of elements in the LIST in operand 3 of the declaration
    1.5) get any type qualifiers for this declaration (currently only the only supported qualifier is global)
        1.5.1) For non-array declarations the type qualifier will be in position 2

1.5.2) For array declarations the type qualifier will be in position 3
2) Check the variable type
2.1) if the type is not defined set a typeNotDefinedError
2.2) if the type specified is 'void' set a vars cannot be decared as void err
3) Check to see if it's already been declared in this frame
3.1) if it hasn't then declare it
3.1.1) For non-array declarations set the type = to the specified type
3.1.2) For array declarations declare an "Array" typed variable. Set the subtype = to the specified type.
3.2) if it has throw a multiply declared var err

psuedocode for modified motov_define

1) *extract the type and function name and whether the return type is an array or not*
2) if the type is not defined generate an error
3) extract the arguments
4) for each type that's not defined generate an error
- 5) if the prototype is already declared generate an error
6) create a private frame
7) For each argument
7.1) Discover whether the argument is an array by checking uc_opcode(argdec) == ARRAY_DECLARATION
7.2) Get the argument type or subtype
7.3) Get the argument name
7.4) If this argument defines an array then
7.4.1) get the dimensions by counting the number of elements in the LIST in operand 2 of the declaration
7.5) if the type is not found recover by setting it to object
7.6) declare the variable in the function frame
7.6.1) if the variable is an array then declare it with type array and set the dimensions and subtype
7.6.2) otherwise just decdlare it normally
8) Declare the special rvalue argument
8.1) If the function declaration returned an array set the dimensions and subtype
9) call motov on the statement list
10) pop off the private frame

psuedocode for modified motov_checkVarAssign

1. add and fill a new dimensions array

```
if (strcmp(var->type->name, val->type->name) == 0) {
        if(isArray(val) &&
           (val->refval.dim != var->val->refval.dim ll
            val->refval.subtype != var->val->refval.subtype)
        ) result = 0; break;
```

psuedocode for modified motov_method

1. add and fill a new dimensions array

```
      if (f->dim > 0) { /* function returns an Array */
         r = moto_createValFromName(env, "Array");
         r->refval.subtype = type;
```

```
        r->refval.dim = r->dim;
      } else {
        r = moto_createValFromName(env, f->rtype);
      }
```

psuedocode for modified motov_fn

1. add and fill a new dimensions array

```
2.      if (f->dim > 0) { /* function returns an Array */
          r = moto_createValFromName(env, "Array");
          r->refval.subtype = type;
          r->refval.dim = r->dim;
        } else {
          r = moto_createValFromName(env, f->rtype);
        }
```

psuedocode for modified motov_id

```
if(isArray(val)){
      val->refval.dim = var.dim;
       val->refval.subtype = var.subtype;
   }
```

New Verifier Errors

moto_illegalTypeForArrayIndex - Illegal type <%s> used for array index. Only integr
al types may be used for array indexes
moto_illegalTypeForSubscriptOp - The subscript operator [] may only be used on Arra
ys, not variables of type %s
moto_illegalSubscriptDimension - The dimensions subscripted %d exceeds the dimensio
ns of the array

IllegalTypeForArrayIndex
IllegalTypeForSubscriptOp
IllegalSubscriptDimension

<u>Compiler Changes</u>

Q. how do we know whether to subscript aa, ai, af etc in array_index ?
A. perhaps we shouldn't print anything there ... but instead just pass through and print in array rval

Q. how should we change the way function names are generated when they take arrays as arguments ?
A. we should probably migrate the whole convention to C++ style

psuedocode for new function motoc_array_index

1) call motoc on the first operand

psuedocode for new function motoc_array_lval

1) call motoc_array_rval

psuedocode for new function motoc_array_rval

psuedocode for new function motoc_array_new

1) Grab the array subtype from the first operand
2) Discover the array dimensions by looking at the length of operands 2 and potentially 3
    2.1) Record the current position on the stack
    2.2) Evaluate the LIST of ARRAY_INDEX elements
3) Construct a new Array type MotoVal
4) Generate the code for calling arr_create

psuedocode for modifications to motoc_declare

1) get dimensions if need be

```
  declaration = uc_operand(p,0);

  isArrayDeclaration = (uc_opcode(declaration) == ARRAY_DECLARATION);

  typen = uc_str(declaration, 0);
  varn = moto_strdup(env, uc_str(declaration, 1));
   type = moto_getType(env, typen);

  if (isArrayDeclaration)
    dim = uc_opcount(uc_operand(declaration, 2));

  if ((!isArrayDeclaration && uc_opcount(declaration) == 3 ) ||
    (isArrayDeclaration && uc_opcount(declaration) == 4)) {
    global = '\1'; /* FIXME: I know it's the only qualifier I have right now
                  but this will need to get changed in the future */
  }
```

2) declare the var differently if its an array

```
  /* Declare the variable */
  if(!isArrayDeclaration)
    var = moto_declare(env, type, varn, global);
  else {
    var = moto_declare(env, moto_getType(env, "Array"), varn, global);
    var->dim = dim;
     var->subtype = type;
  }
```

psuedocode for modifications to motoc_assign

psuedocode for modified motoc_id

```
if(isArray(val)){
       val->refval.dim = var.dim;
        val->refval.subtype = var.subtype;
    }
```

psuedocode for modifier motoc_fn

psuedocode for modifier motoc_method

psuedocode for modifier motoc_define

psuedocode for modifier motoc_new

Interpreter Changes

psuedocode for new function motoi_array_index

1) Evaluate the expression operand (0)
2) Push the result ontothe stack (where does bounds checking get done in java?)

psuedocode for new function motoi_array_lval

1) call motoi_array_rval

psuedocode for new function motoi_array_rval

1) Grab the array by evaluating the first operand
2) Record the current position on the stack
3) Evaluate the LIST of ARRAY_INDEX elements
4) if the dimensions subscripted equals the dimensions of the array
    4.1) create a new motoval of the subtype of the array
    4.2) set its value to the specified indexes into the array
5) else (The dimensions subscripted are less than the dimensions of the array)
    5.1) create a new array type motoval
    5.2) set its value to the specified indexes into the array

psuedocode for new function motoi_array_new

1) Grab the array subtype from the first operand
2) Discover the array dimensions by looking at the length of operands 2 and potentially 3
    2.1) Record the current position on the stack
    2.2) Evaluate the LIST of ARRAY_INDEX elements
3) Construct a new Array type MotoVal
4) Build the array by calling arr_create

psuedocode for modifications to motoi_declare

1) get dimensions if need be

```
declaration = uc_operand(p,0);

isArrayDeclaration = (uc_opcode(declaration) == ARRAY_DECLARATION);

typen = uc_str(declaration, 0);
varn = moto_strdup(env, uc_str(declaration, 1));
 type = moto_getType(env, typen);

if (isArrayDeclaration)
```

```
    dim = uc_opcount(uc_operand(declaration, 2));

  if ((!isArrayDeclaration && uc_opcount(declaration) == 3 ) ||
    (isArrayDeclaration && uc_opcount(declaration) == 4)) {
    global = '\1'; /* FIXME: I know it's the only qualifier I have right now
                but this will need to get changed in the future */
  }

2) declare the var differently if its an array
  /* Declare the variable */
  if(!isArrayDeclaration)
    var = moto_declare(env, type, varn, global);
  else {
    var = moto_declare(env, moto_getType(env, "Array"), varn, global);
    var->dim = dim;
     var->subtype = type;
  }
```

psuedocode for modifications to motoi_assign

psuedocode for modifications to motoi_new

1) add and fill a new dimensions array

psuedocode for modifications to motoi_method

1. add and fill a new dimensions array

```
1) Change when allocating the function return value
  if (f->dim > 0) { /* function returns an Array */
    r = moto_createValFromName(env, "Array");
    r->refval.subtype = type;
    r->refval.dim = f->dim;
  } else {
    r = moto_createValFromName(env, f->rtype);
  }
```

psuedocode for modifications to motoi_fn

1. add and fill a new dimensions array

```
1) Change when allocating the function return value
  if (f->dim > 0) { /* function returns an Array */
    r = moto_createValFromName(env, "Array");
    r->refval.subtype = type;
    r->refval.dim = f->dim;
  } else {
    r = moto_createValFromName(env, f->rtype);
  }
2) Change when setting the values of function parameters from moto defined functions

      int isArrayDeclaration;

       /* For each type that is not defined generate an error */
```

```
 isArrayDeclaration = (uc_opcode(argdec) == ARRAY_DECLARATION);
atypen = uc_str(argdec,0);
 atype = moto_getType(env, atypen);
aname = uc_str(argdec,1);
if (isArrayDeclaration)
   dim = uc_opcount(uc_operand(argdec, 2));

if(!isArrayDeclaration)
   avar = moto_declare(env, atype, aname, '\0');
else {
   avar = moto_declare(env, moto_getType(env,"Array"), aname, '\0');
   avar->dim = dim;
    avar->subtype = atype;
}

 moto_setVar(env, avar, val);
```

Changes to Moto Functions

Since arrays may now be passed as arguments into moto functions, changes must be made to the moto resolves which function to call. This resolution occurs in the function **ftab_get** in ftable.c . It takes an array of String 'types' (char**) to do the resolution.

Each type is compared via a string equality operation to mxf->argtypes[i] ... this is problematic in many ways since argtypes[i] for arrays is currently just 'int' ...

A functions argtypes are instantiated in dl.c (336) but only the argtypes array is allocated. The individual elemens are not. They are instead strtok'd

Possible options:
1) Retain the argtype as a String and for arrays store entries like int[] or String[][]
- ftab.get will not need to change
2) Store moto vals for argtype representation
3) Continue down the path of storing a seperate argdimensions array
- If the argdimensions array for the current index is > 0 then the argtype stored is an array
- ftab.get would need to be passed an argdimensions array OR the argtypes would need to be modified

Changes in the ftable

ftab_get

needs to take a new dimensions array
needs to take into account dimensions when building the string 'key' to the cache

ftab_cacheGet

needs to take a new dimensions array

Examples

Notes

Declaring arrays merely says what kind of values the array will hold. It does not create them. Java arrays are objects, and like any other object you use the `new` keyword to create them. When you create an array, you must tell the compiler how many components will be stored in it. Here's how you'd create the variables declared on the previous page:

```
k = new int[3];
yt = new float[7];
names = new String[50];
```

The numbers in the brackets specify the length of the array; that is, how many slots it has to hold values. With the lengths above `k` can hold three ints, `yt` can hold seven floats and `names` can hold fifty Strings. This step is sometimes called allocating the array since it sets aside the memory the array requires.

Unbalanced Arrays

Like C Java does not have true multidimensional arrays. Java fakes multidimensional arrays using arrays of arrays. This means that it is possible to have unbalanced arrays. An unbalanced array is a multidimensional array where the dimension isn't the same for all rows. In most applications this is a horrible idea and should be avoided.

examples

float[] foo
int[] bar = new int[12]
String[][] fuzzy = new String[3][4]

```
double[][] ID3 = {
  {1.0, 0.0, 0.0},
  {0.0, 1.0, 0.0},
  {0.0, 0.0, 1.0}
};
```

int[][][] foobar = new String[4][][]
foobar[2] = new String[3][2]
foobar[3] = foobar[2]

maka()[3]=5
bane = simple[4][3][2]
bra = fuzzy(foo,bar)[5][2]