Notes on the Specification of the `M` language

Robin Cockett*
Programming Languages group
Department of Computer Science
University of Calgary

April 1, 2008

1 Introduction

The programming language `M` is one of an array of languages which has various features which started with a language which I called "Minisculus" which was enlarged to provide more programming features (and a more challenging exercise) for students in the Compiler Construction courses at the University of Calgary.

"Minisculus" was a bare bones language with variable declarations and assignments, and some basic control constructs (such as conditional statements, while loops, or for loops). It was compiled down to a very simple stack machine written in about 20 lines of shell script. This language was subsequently extended to a language which I called `M+` which was a simply typed block structured language with some ad hoc polymorphism. The language allowed functions and variables to be defined locally and students, therefore had to resolve not only the type of expressions but also the scope issues underlying a block structured language.

To support `M+` I decided to develop a small assembly language which I called `AM` and I wrote an assembler for this which did extensive type and bounds checking. This assembler gradually was enlarged to include a heap, with instructions to support heap manipulations, and an enlarged set of instructions to support array manipulations. These last two enlargements were prompted by the desire to support two extensions of `M+` which I called `M++` and `M+-`. `M++` allowed the declaration of simple (inductive) datatypes with computations driven by case statements for these structures. `M+-` introduced arrays into the language: these arrays could be defined locally but could not be returned as values of functions. This meant that they were passed by reference and updated through that reference locally. As arrays could be of arbitrary dimension and size, their bound information had to be recorded within their body and type checking simply had to keep track of their dimension (these mechanisms are explained in detail later).

The language `M` is the amalgam of `M+-` and `M++`. The structure of the type system is three tiered: at the base there are the built-in types (`int`, `real`, `bool`, `char`), built on these are the simple datatypes which are constructor based, and finally there are arrays of simple datatypes. This does mean that one is not allowed arrays of structures which themselves contain arrays as all

*Thanks to Allan Lyons and Seulkiro Park for a number of corrections March, 2008

array structure is required to be at the top level. Semantically this is quite natural as the array structure is handled by reference while all the simple datatypes are handled by value.

The purpose of this document is to describe the language `M` and to discuss how it may be implemented over the `AM` assembler. The discussion of the language starts with the grammar and then goes on to describe the informal semantics and the basic implementation issues.

2 The Syntax of `M`

`M` is a block structured language with functions (as opposed to procedures) which has arrays and simple inductive datatypes (such as lists and trees). Below is the grammar of `M` accompanied by a brief commentary.

2.1 Commentary on `M` grammar

An `M` program is a block. This is a list of declarations followed by the program body which is a list of statements:

```
prog -> block.

block -> declarations program_body.
```

2.1.1 Declarations

A declaration can either be a function declaration, a variable declaration, or a simple inductive (polynomial) datatype declaration. Each declaration is terminated by a semi-colon.

```
declarations -> declaration SEMICOLON declarations
             |.

declaration -> var_declaration
             | fun_declaration
             | data_declaration.
```

A variable declaration is preceded by the reserved word `var` and declares a list of identifiers or arrays whose type is attached by a colon followed by the simple type. Arrays sizes may be given as expressions in terms of variables in whose scope the declaration lies. This allows one to declare a local array of a size dependent on an argument of the function or of a variable declared in an outer scope.

All types in `M` are constructed from the four ground types: reals, integers, booleans, and characters. These can be used to build inductive datatypes such as lists and trees which are the basic types of `M`. A variable can, in addition, have an array type, which is given by a single positive number which signifies the number of dimensions. Simple variables have dimension zero.

```
var_declaration
          -> VAR var_specs COLON type.
```

```
var_specs -> var_spec more_var_specs.

more_var_specs
          -> COMMA var_spec more_var_specs
           |.

var_spec
     -> ID array_dimensions.

array_dimensions
     -> SLPAR expr SRPAR array_dimensions
      |.

type -> INT
      | REAL
      | BOOL
      | CHAR
      | ID.
```

Examples of variable declarations include:

```
var x:int;
var b:bool;
var a[100]:int;
var A[3][3]:real;
var c[20]:char;
var L:intlist;
```

where `A` is 2-dimensional array and `L` is a variable of type `intlist`.

Also allowed is an extended syntax for multiple declarations on the same line:

```
var i,j,k,scores[N]:int;
```

which is convenient for writing the declarations more succinctly.

2.1.2 Functions

A function declaration is preceded by the reserved word `fun` and consists of an identifier followed by an argument list with a type followed by the function block. This consist of a declaration list followed by the function body enclosed in curly parentheses. The argument list consist of a (possibly empty) list of variable declarations separated by commas. Arrays are declared in argument lists without their size indicated but with the number of dimensions indicated by the number of square parentheses required. Arrays are passed by reference, thus they are passed as a pointer to the location at which they are stored.

A function can call any function which has already been declared or is declared in the same block. Thus, (mutually) recursive functions are permissible. Functions are also allowed to use variables defined in the same block. A variable, array, or function in a minisculus program can

only be legally used if it is been declared in an enclosing block or function: the usual rules of scope apply.

```
fun_declaration
     -> FUN ID param_list COLON type CLPAR fun_block CRPAR.

fun_block -> declarations fun_body.

param_list -> LPAR parameters RPAR.

parameters -> basic_declaration more_parameters
            |.

more_parameters -> COMMA  basic_declaration more_parameters
            |.

basic_declaration -> ID basic_array_dimensions COLON type.

basic_array_dimensions -> SLPAR SRPAR basic_array_dimensions
             |.
```

Here is a simple example of a function:

```
fun MAX(n:int,m:int):int
{ var ans:int;
  begin
    if n=<m then ans:=m
    else ans:=n;
    return ans;
  end };
```

One may also omit the `begin` and `end` statements.

Also allowed is an extended syntax (whose definition has not been given, but is similar to the variable declaration above) for multiple parameters of the same type:

```
fun MAX(n,m:int):int;
```

which is convenient for writing the function declarations more succinctly.

A function can also take array arguments. Arrays are passed by reference and cannot be the result of a function. Below is an example: a function to sum the elements of an integer array. Note that one does not have to know the size of the array: this is obtained dynamically using the built-in function `size`. This means that the type system only needs to keep track of the simple type and the number of dimensions of a variable.

```
fun SUM(a[]:int):int
{ var i,sum:int;
    sum:= 0; i:= 0;
```

```
        while i<size(a) do
        { sum:= sum + a[i];
          i:= i+1;
        };
        return sum;
    };
```

Note also how the dimension of an argument to a function is specified: each dimension is represented by a empty pair of square braces. Note also that array bounds start at 0 and run to one less than the size of the dimension.

To further illustrate how the array dimensions work, here is an example function which computes the size of a three dimensional array:

```
fun SIZE(x[][][]:real):int
{ return size(x)*size(x[])*size(x[][]);
};
```

2.1.3 Polynomial inductive datatypes

The `M` language supports simply typed, mutually recursive, polynomial inductive datatypes. These are defined by naming the new type and providing the constructors of that type and its arguments. These datatypes can be passed both as arguments and as the result of a function: for this reason their storage is allocated on the heap.

```
data_declaration -> DATA ID EQUAL cons_declarations.

cons_declarations -> cons_decl more_cons_decl.

more_cons_decl -> SLASH cons_decl more_cons_decl
                |.

cons_decl -> CID OF type_list
           | CID.

type_list -> type more_type.

more_type -> MUL type more_type
           |.
```

Here is how to declare the type of integer lists:

```
data intlist = #NIL
             | #CONS of int * intlist;

var L:intlist;

L:= #CONS(1,#CONS(2,#CONS(3,#NIL)));
```

Here the constructors are indicated by prefixing the identifier with a `#`. The constructor `#NIL` has no arguments. The constructor `#CONS` has two arguments: an integer and an `intlist`.

Here is the datatype of integer trees and a function to collect the integers of the tree into a list:

```
data inttree = #LEAF of int
             | #BRAN of inttree * inttree;

fun coll(t:intree):intlist
{ var L:intlist;
    case t of { #LEAF n => L:= #CONS(n,#NIL)
                #BRAN(t1,t2) => L:= append(coll(t1),coll(t2)) }
    return L;
 };
```

2.1.4 `M` statements

The difference between a program body and a function body is that the function body *must* end with a return statement. Otherwise both consist of a series of program statements separated by semi-colons. Program statements include assignments, conditional statements, while loops, read statements, print statements, case statements and blocks. Notice that a block permits the declaration of local variables, arrays, and functions and is delimited by curly braces.

```
program_body -> BEGIN prog_stmts END
              | prog_stmts.


fun_body -> BEGIN prog_stmts RETURN expr SEMICOLON END
          | prog_stmts RETURN expr SEMICOLON.

prog_stmts -> prog_stmt SEMICOLON prog_stmts
            |.

prog_stmt -> IF expr THEN prog_stmt ELSE prog_stmt
           | WHILE expr DO prog_stmt
           | READ location
           | location ASSIGN expr
           | PRINT expr
           | CLPAR block CRPAR
           | CASE expr OF CLPAR case_list CRPAR.

location -> ID array_dimensions.
```

The case statements provide the basic mechanism for programming with the datatypes. These consist of matching an expression whose type is the given datatype and branching according to which constructor is detected. When a constructor is detected the arguments of the given constructor are made available in the program as new named variables.

```
case_list -> case more_case.

more_case -> SLASH case more_case
          |.

case -> CID var_list ARROW prog_stmt.

var_list -> LPAR var_list1 RPAR
          |.

var_list1 -> ID more_var_list1.

more_var_list1 -> COMMA ID more_var_list1
                |.
```

Here is an example of how to use a case statement to obtain the tail of a list:

```
case L of  #NIL => L:= #NIL
         | #CONS(x,xs) => L:= xs;
```

The read and print statement can only handle basic types (`real`, `int`, `bool`, and `char`). However, one can read such a value into an entry of an array.

2.1.5 Expressions

There are three kinds of expression in `M`: integer, real, and boolean. The syntax cannot distinguish these different sorts of expressions and thus some type checking is necessary. Basic `M` does not include any type coercions (e.g. from `int` to `real`).

Boolean expressions are used in conditional and while statements. Boolean expressions include comparisons of integer and real expressions.

```
expr ->  expr OR bint_term
       | bint_term.

bint_term -> bint_term AND bint_factor
           | bint_factor.

bint_factor -> NOT bint_factor
             | int_expr compare_op int_expr
             | int_expr.

compare_op -> EQUAL | LT | GT | LE |GE.

int_expr -> int_expr addop int_term
          | int_term.

addop -> ADD | SUB.
```

```
int_term -> int_term mulop int_factor
          | int_factor.

mulop -> MUL | DIV.

int_factor -> LPAR expr RPAR
            | SIZE LPAR ID basic_array_dimensions RPAR
            | FLOAT LPAR expr RPAR
            | FLOOR LPAR expr RPAR
            | CEIL LPAR expr RPAR
            | ID modifier_list
            | CID cons-argument_list
            | IVAL
            | RVAL
            | BVAL
            | CVAL
            | SUB int_factor.
```

A modifier list is either the arguments of a function or the address list of an array. Clearly these must be correctly typed.

```
modifier_list  -> fun_argument_list
           | array_dimensions.

fun_argument_list  ->  LPAR arguments RPAR

cons_argument_list ->  fun_argument_list
           |.

arguments -> expr more_arguments
           |.

more_arguments -> COMMA expr more_arguments
           |.
```

2.2 Lexical analysis for `M`

The lexical tokens for the `M` grammar are defined as follows:

```
"+"  => ADD
"-"  => SUB
"*"  => MUL
"/"  => DIV
"=>" => ARROW
```

```
"&&" => AND
"||" => OR
"not" => NOT

"="  => EQUAL
"<"  => LT
">"  => GT
"=<"  => LE
">="  => GE

":=" => ASSIGN

"("   => LPAR
")"   => RPAR
"{"   => CLPAR
"}"   => CRPAR
"["   => SLPAR
"]"   => SRPAR
"|"   => SLASH

":"  => COLON
";"  => SEMICLON
","  => COMMA

"if" => IF
"then" => THEN
"while" => WHILE
"do" => DO
"read" => READ
"else" => ELSE
"begin" => BEGIN
"end" => END
"case" => CASE
"of" => OF
"print" => PRINT
"int"   => INT
"bool"  => BOOL
"char"  => CHAR
"real"  => REAL
"var"   => VAR
"data"  => DATA
"size" => SIZE
"float" => FLOAT
"floor" => FLOOR
"ceil"  => CEIL
```

```
"fun"   => FUN
"return" => RETURN

"#"[_{digit}{alpha}]*     => CID         (constructor)
{alpha}[_{digit}{alpha}]* => ID          (identifier)
{digit}+"."{digit}+ => RVAL            (real/float)
{digit}+ => IVAL                         (integer)
"false" => BVAL                          (booleans)
"true" => BVAL
{quote}{char}{quote} => CVAL             (character)
{quote}"\n" {quote} => CVAL
{quote}"\t" {quote} => CVAL

where

alpha = [a-zA-Z]
digit = [0-9]
quote = ["]
char = [^_A-Za-z0-9]
```

2.3 Comments

M has two types of comments: multi-line comments

```
/*  comment  */
```

and one line comments

```
%   comment
```

The multi-line comments allow nesting of comments so that commenting out a section of code which already has comments has the correct effect.

3 Abstract syntax and intermediate representation tree for M

The abstract syntax tree for M is the following Haskell datatype:

```
data M_prog = M_prog ([M_decl],[M_stmt])
data M_decl = M_var (String,[M_expr],M_type)
            | M_fun (String,[(String,Int,M_type)],M_type,[M_decl],[M_stmt])
            | M_data (String,[(String,[M_type])])
data M_stmt = M_ass (String,[M_expr],M_expr)
            | M_while (M_expr,M_stmt)
            | M_cond (M_expr,M_stmt,M_stmt)
            | M_read (String,[M_expr])
            | M_print M_expr
            | M_return M_expr
```

```
            | M_block ([M_decl],[M_stmt])
            | M_case (M_expr,[(String,[String],M_stmt)])
data M_type = M_int | M_bool | M_real | M_char | M_type of string
data M_expr = M_ival Int
            | M_rval Float
            | M_bval Bool
            | M_cval Char
            | M_size (String,Int)
            | M_id (String,[M_expr])
            | M_app (M_operation,[M_expr])
data M_operation
                = M_fn String
                | M_cid String
                | M_add | M_mul | M_sub | M_div | M_neg
                | M_lt | M_le | M_gt | M_ge | M_eq | M_not | M_and | M_or
                | M_float | M_floor | M_ceil;
```

Notice that all operations (including function calls) are handled uniformly by `M_app` in this tree. Further that expressions need not make semantic sense as it is possible to add a boolean to a real, for example.

The intermediate representation has the following effects:

- it removes all the basic declarations;
- it collects the function declarations at each level;
- it collects the local array declarations;
- in expressions it replaces identifiers by level and offset indexes with a possible offset calculation for an array;
- function calls are replaced by a label and level index;
- constructors are replaced by a number.

```
data I_prog = I_PROG ([I_fbody],Int,[(Int,[I_expr])],[I_stmt])
          -- functions, number of local variables, array descriptions, body.
data I_fbody = I_FUN (String,[I_fbody],Int,Int,[(Int,[I_expr])],[I_stmt])
          -- functions, number of local variables, number of arguments
          -- array descriptions, body
data I_stmt = I_ASS (Int,Int,[I_expr],I_expr)
          -- level, offset, array indexes, expressions
            | I_WHILE (I_expr,I_stmt)
            | I_COND (I_expr,I_stmt,I_stmt)
            | I_CASE (I_expr,[(Int,Int,I_stmt)])
          -- each case branch has a constructor number, a number of arguments,
          -- and the code statements
            | I_READ_B (Int,Int,[I_expr])
```

```
          -- level, offset, array indexes
            | I_PRINT_B I_expr
            | I_READ_I (Int,Int,[I_expr])
            | I_PRINT_I I_expr
            | I_READ_F (Int,Int,[I_expr])
            | I_PRINT_F I_expr
            | I_READ_C (Int,Int,[I_expr])
            | I_PRINT_C I_expr
            | I_RETURN I_expr
            | I_BLOCK ([I_fbody],Int,[(Int,[I_expr]))],[I_stmt])
          -- functions, number of local variables, array descriptions, body.

data I_expr = I_IVAL Int
            | I_RVAL Float
            | I_BVAL Bool
            | I_CVAL Char
            | I_ID (Int,Int,[I_expr])
          -- level, offset, array indices
            | I_APP (I_opn,[I_expr])
            | I_REF (Int,Int)
          -- for passing an array reference as an argument
          -- of a function: level, offset
            | I_SIZE (Int,Int,Int)
          -- for retrieving the dimension of an array: level,offset,dimension
data I_opn = I_CALL (String,Int)
          -- label and level
           | I_CONS (Int,Int)
          -- constructor number and number of arguments
           | I_ADD_I | I_MUL_I | I_SUB_I | I_DIV_I | I_NEG_I
           | I_ADD_F | I_MUL_F | I_SUB_F | I_DIV_F | I_NEG_F
           | I_LT_I  | I_LE_I  | I_GT_I  | I_GE_I  | I_EQ_I
           | I_LT_F  | I_LE_F  | I_GT_F  | I_GE_F  | I_EQ_F
           | I_LT_C  | I_LE_C  | I_GT_C  | I_GE_C  | I_EQ_C
           | I_NOT | I_AND | I_OR | I_FLOAT | I_FLOOR | I_CEIL
```

3.1 Example translations

Here are two examples of translations:

3.1.1 A basic translation

Here is a small M+ program with no arrays or datatypes:

```
          var x,y:int;
          fun exp(b:int):int
          { var z:int;
```

```
          if b=0 then z:= 1
                 else z:= x * exp(b-1);
          return z;
        };
        read x;
        read y;
        print exp(y);
```

Here is its syntax tree:

```
M_prog
  ([M_var ("x",[],M_int),
    M_var ("x",[],M_int),
    M_fun
      ("exp",[("b",0,M_int)],M_int
         ,[M_var ("z",[],M_int)]
         ,[M_cond
             (M_app (M_eq,[M_id("b",[]),M_ival 0]),M_ass ("z",[],M_ival 1),
              M_ass
                ("z",[]
                 M_app
                   (M_mul,
                    [M_id("x",[]),
                     M_app
                       (M_fn "exp",[M_app (M_sub,[M_id("b",[])
                                    ,M_ival 1])])])))),
          M_return (M_id("z",[]))))],
   [M_read("x",[]),M_read("y",[])
   ,M_print (M_app (M_fn "exp",[M_id("y",[])]))])
```

In the intermediate representation the declarations all disappear: they are only "used" in the construction of the symbol table. Function declaration are replaced by the representation of their bodies attached to a code label to which a caller jumps. The body of a function is then same as a program block: the fact that there are arguments is now embedded in the code. A program block indicates how many cells it is necessary to allocate for local storage (and if there are arrays a list of descriptions of these). This information is needed in the generation of the entry and return code sequence of the function/block code. It also, of course, contains the representation of the statements in the block.

The invocation a function requires not only the label but also the static distance of the call from the actual function declaration so that the access link can be set correctly. All variables are replaced by the offset and level calculated from the symbol table.

Notice also that all operations are distinguished by the data that they act on. Thus, reading or printing integers is distinct from reading or printing booleans. We can do this because type checking has resolved the type of each argument!

The basic intermediate representation of the above program is:

```
I_PROG
```

```
([I_FUN
    ("fn1"
    ,[]                         -- no local function
    ,1                          -- one local variable
    ,1                          -- one argument
    ,[I_COND(IAPP (IEQ,[I_ID(0,-4,[]),I_IVAL 0])
     ,I_ASS(0,1,I_IVAL 1)
     ,I_ASS(0,1,I_APP(I_MUL_I,[I_ID(1,1,[])
                         ,I_APP(I_CALL ("fn1",1),[I_APP(I_SUB,[I_ID(0,-4,[])
                                                               ,I_IVAL 1])])])))
     ,I_RETURN (I_ID(0,1 []))
     ])]
,2           -- two local variables
,[I_READ_I (0,1,[])
 ,I_READ_I (0,2,[])
 ,I_PRINT_I(I_APP (I_CALL ("fn1",0),[I_ID(0,2,[])]))
 ])
```

3.1.2 An example with arrays

Here is a small M program with arrays:

```
var x[2]:int;
fun exp(b:int):int
{ var z:int;
  if b=0 then z:= 1
         else z:= x[1] * exp(b-1);
 return z;
};
read x[0];
read x[1];
print exp(x[0]);
```

Here is its syntax tree:

```
M_prog
  ([M_var ("x",[M_ival 2],M_int),
    M_fun
      ("exp",[("b",0,M_int)],M_int
         ,[M_var ("z",[],M_int)]
         ,[M_cond
             (M_app (M_eq,[M_id("b",[]),M_ival 0]),M_ass ("z",[],M_ival 1),
              M_ass
                ("z",[],
                 M_app
                   (M_mul,
```

```
                  [M_id("x",[M_ival 1]),
                   M_app
                    (M_fn "exp",[M_app (M_sub,[M_id("b",[]),M_ival 1])])])))),
          M_return (M_id("z",[]))])],
   [M_read("x",[M_ival 1]),M_read("y",[M_ival 2])
   ,M_print (M_app (M_fn "exp",[M_id("y",[M_ival 2])]))])])
```

Notice in this syntax tree every variable is treated as a potential array: those which are not arrays simply have the empty lists of array indexes.

The intermediate representation is:

```
I_PROG
  ([I_FUN
      ("fn1"
      ,[]                                    -- no local functions
      ,1                                     -- one local variable
      ,1                                     -- one argument
      ,[]                                    -- no arrays
      ,[I_COND(I_APP (I_EQ_I,[I_ID(0,-4,[]),I_INT 0])
       ,I_ASS(0,1,[],I_INT 1)
       ,I_ASS(0,1,[],I_APP(I_MUL,[I_ID(1,1,[I_INT 2])
                          ,I_APP(I_CALL ("fn1",1),[I_APP(I_SUB,[I_ID(0,-4,[])
                                                               ,I_INT 1])])])))
       ,I_RETURN (I_ID(0,1,[]))
       ])]
  ,1           -- one local variable which is an array
  ,[(1,[I_INT 2])]  --  array spec
  ,[I_READ_I (0,1,[I_INT 0])
   ,I_READ_I (0,1,[I_INT 1])
   ,I_PRINT_I (I_APP (I_CALL ("fn1",0),[I_ID(0,1,[I_INT 0])]))
   ])
```

3.1.3 Blocks and arrays

Here is a program with blocks and arrays:

```
var n:int;
read n;
{ var a[n]:real;
  n:=0;
  while n<size(a) do {read a[n]; n:=n+1;};
  n:=0;
  while n<size(a) do {print a[n]; n:=n+1;};
};
```

The intermediate representation for this program should look like:

```
I_PROG([]
      ,1
      ,[]
      ,[I_READ_I (0,1,[])
       ,I_BLOCK([]
               ,1
               ,[(1,[I_ID (1,1,[])])]
               ,[I_ASS (1,1,[],I_IVAL 0)
               ,I_WHILE(I_APP (I_LT_I,[I_ID (1,1,[]),I_SIZE (0,1,1)])
                      ,I_BLOCK([],0,[]
                              ,[I_READ_F (1,1,[I_ID (2,1,[])])
                               ,I_ASS (2,1,[],I_APP (I_ADD_I,[I_ID (2,1,[])
                                                             ,I_IVAL 1]))
                              ]))
               ,I_ASS (1,1,[],I_IVAL 0)
               ,I_WHILE(I_APP (I_LT_I,[I_ID (1,1,[]),I_SIZE (0,1,1)])
                      ,I_BLOCK([],0,[]
                              ,[I_PRINT_F (I_ID (1,1,[I_ID (2,1,[])]))
               ,I_ASS (2,1,[],I_APP (I_ADD_I,[I_ID (2,1,[]),I_IVAL 1]))])))])
       ])
```

4 Operational semantics

There is a lot to say here ... but for the moment I shall just comment that the semantics of `M` is basically very standard with the possible exception of how its datatypes are handled. To illustrate one subtlety consider the following code:

```
data rosetree = #bud of rosetreelist;
data rosetreelist = #rnil
                  | #rcons of rosetree * rosetreelist;
var x,y:rosetree;

begin
  x:= #rcons(y,x);
  y:= #bud(x);
end
```

This code is legal and to the unwary may lead one to believe that one can build circular structures in `M`. However, this is definitely not the case! In the first assignment neither `x` nor `y` are defined (they are `void`) so that `x` is assigned `#rcons(void,void)` while `y` in the next step is assigned `#bud(#rcons(void,void))`.