Introduction to MicroC

Letterio Galletta

MicroC

MicroC is a sublanguage of C

Many simplifications have been made compared to real C

- datatypes: only int and char variables, arrays, and pointers
- no structs, unions, doubles, function pointers, …
- no initializers in variable declarations
- Functions can return only int, char, void, bool
- No pointer arithmetic
- Pointers and arrays are not interchangeable
- No dynamic allocation of memory

An example of MicroC (1)

```
/* Function to reverse arr[] from start to end*/
void reverseArray(int arr[], int start, int end)
{
    int temp;
    while (start < end)
    {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start = start + 1;
        end = end - 1;
    }
}
```

An example of MicroC (2)

```
/* Utility that prints out an array on a line */
void printArray(int arr[], int size)
{
  int i;
  for (i=0; i < size; i = i + 1)
    print(arr[i]);
}
```

An example of MicroC (3)

```
/* Main function */
int main()
{
    int arr[6];  int i; int n; n = 6;

    for(i = 0; i < 6; i = i + 1)
      arr[i] = i + 1;
    reverseArray(arr, 0, n-1);
    printArray(arr, n);
    return 0;
}
```

MicroC compilation

MicroC can be compiled to native code using the LLVM infrastructure

The microc compiler generates LLVM bitcode that

- Can be run with the tool lli
- Can be translated to assembler with the tool llc
- Can be linked with other C code and translated to native code with clang compiler

Abstract syntax of MicroC (1)

The definition of the abstract syntax tree is defined in the file ast.ml

Roughly, there are four main syntactic categories:

1. Expression
2. Access expression, i.e., l-value expressions
3. Statements
4. Declaration, e.g., functions and global variables

A program is a list of function or global variable declarations

Abstract syntax of MicroC (2)

A node of the AST is annotated with a position and an id

```
type 'a annotated_node = {loc : position[@opaque]; node : 'a; id : int
}[@@deriving show]
```

- The loc field stores about the position in the source file
- The node field is the syntactic element
- id is not used at the moment
- The annotation [@@ deriving show] is used by the deriving ppx to automatically generate a string representation of the node

Abstract syntax of MicroC (3)

Nodes are built by instantiating the annotated_node record

```
type expr = expr_node annotated_node
and expr_node =
 | Access of access             (* x   or  *p   or  a[e]    *)
 | Assign of access * expr      (* x=e  or  *p=e  or  a[e]=e   *)
 | Addr of access               (* &x   or  &*p   or  &a[e]    *)
 | ILiteral of int              (* Integer literal          *)
 | CLiteral of char             (* Char literal            *)
 | ...
[@@deriving show]
```

MicroC Lexical elements (1)

Identifiers starts with a letter or an underscore and then can contain letters, underscore and numbers

i, _local_var, string_of_int32

Integer literal are sequence of digits (integers are 32bit values)

32, 1024, 3232

Character literals have the form ‘c’ where c is a character

‘A’, ‘b’, ‘1’

MicroC Lexical elements (2)

Boolean literals are true and false

Keywords are: if, return, else, for, while, int, char, void, NULL, bool

Operators: &, +, -, *, /, %, =, ==, !=, <, <=, >, >=, &&, ||, !

Other symbols: (,), {, }, [,], &, ;, ,

Comments:

Single line comments //

Multiple lines /* … */

Operator precedence and associativity

```
right    =            /* lowest precedence */
left     ||
left     &&
left     ==  !=
nonassoc >  <  >=  <=
left     +  -
left     *  /  %
nonassoc !  &
nonassoc [            /* highest precedence  */
```

MicroC Grammar (1)

Program ::= Topdecl* EOF

Topdecl ::= Vardecl “;” | Fundecl

Vardecl ::= Typ Vardesc

Vardesc ::= ID | “*” Vardesc | “(“ Vardesc “)” | Vardesc “[“ “]” | Vardesc “[“ INT “]”

Fundecl ::= Typ ID “(“((Vardecl “,”)* Vardecl)? “)” Block

Block ::= “{“ (Stmt | Vardecl “;”)* “}”

Typ ::= “int” | “char” | “void” | “bool”

MicroC Grammar (2)

```
Stmt ::= “return” Expr “;” | Expr “;” | Block | “while” “(“ Expr “)” Stmt
      |   “for” “(“ Expr? “;” Expr? “;” Expr? “)” Stmt
      |   “if” “(“ Expr “)” Stmt “else” Stmt  | “if” “(“ Expr “)” Stmt

Expr ::= RExpr | LExpr

LExpr ::= ID | “(“ LExpr “)” | “*” LExpr | “*” AExpr | LExpr “[“ Expr “]”

RExpr ::= AExpr | ID “(“ ((Expr “,”)* Expr)? “)” | LExpr “=” Expr | “!” Expr
        |  “-” Expr | Expr BinOp Expr
```

MicroC Grammar (3)

BinOp ::= “+” | “-” | “*” | “%” | “/” | “&&” | “||” | “<” | “>” | “<=” | “>=” | “==” | “!=”

AExpr ::= INT | CHAR | BOOL | “NULL” | “(“ RExpr “)” | “&” LExpr

Notes:

- The grammar is ambiguous
- Tokens with no semantic values are enclosed in “”
- Tokens with semantic values are capitalized, e.g., ID, NAME