Introduction to µComp-lang

Letterio Galletta

µComp-lang

µComp- lang a simple component-based imperative language

Main features

- Components are the basic units, which are linked/wired together to form whole programs
- Components provides and uses Interfaces
- Interfaces specify a set of functions and global variables to be provided by the interface's provider
- Components are statically linked to each other via their interfaces

µComp-lang

µComp- lang a simple component-based imperative language

Many simplifications have been made

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

An example of µComp-lang (1)

```
interface Sorter {

  def sort(arr : int[], size : int) : void;

}

interface Comparator {

  def compare(n1 : int, n2 : int) : int;

}

interface Printer {

  def printArray(arr : int[], n : int);

}
```

```
component CPrinter provides Printer {
  def printArray(arr : int[], n : int){
    var i : int;
    for (i = 0; i < n; i = i + 1)
        print(arr[i]);
  }
}
```

An example of µComp-lang (2)

```
component InsertionSort provides Sorter uses Comparator {
  def insertionSort(arr : int[], n : int){
      var i : int; var key : int; var j : int;

      for (i = 1; i < n; i = i + 1) {
          key = arr[i];
          j = i - 1;

          /* Move elements of arr[0..i-1], that are
            greater than key, to one position ahead
            of their current position */
          while (j >= 0 && compare(arr[j], key) > 0) {
              arr[j + 1] = arr[j];
              j = j - 1;
          }
          arr[j + 1] = key;
      }
  }

  def sort(arr : int[], n : int){
    insertionSort(arr, n);
  }
}
```

An example of µComp-lang (3)

```
component EntryPoint provides App,Comparator uses Printer, Sorter {

  def compare(n1 : int, n2 : int) : int {
      return n1 - n2;
  }

  def main() : int {
      var arr : int[5];

      arr[0] = 12;
      arr[1] = 11;
      arr[2] = 13;
      arr[3] = 5;
      arr[4] = 6;

      sort(arr, 5);
      printArray(arr, 5);

      return 0;
  }
}
```

An example of µComp-lang (4)

```
connect {
    EntryPoint.Sorter <- InsertionSort.Sorter;
    EntryPoint.Printer <- CPrinter.Printer;
    InsertionSort.Comparator <- EntryPoint.Comparator;
}
```

µComp-lang compilation (1)

µComp-lang can be compiled to native code using the LLVM infrastructure

The 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 C code and translated to native code with clang compiler

μComp-lang compilation (2)

The compilation in organized in the following steps:

- Parsing: transform a source program into an untyped (but located) AST
- Semantic analysis: transform an untyped AST into a typed AST where the non local names are qualified with the interface
- Linking: the non local names are resolved with the component specified by the programmer (qualified with a component name)
- Codegen: a typed AST is translated into a LLVM module (name mangling)

Abstract syntax of μComp-lang (1)

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

Roughly, there are seven main syntactic categories:

1. Expression
2. l-value expressions
3. Statements
4. Declaration, e.g., functions and variables
5. Interfaces
6. Components
7. Connections

A program is a list of interfaces, components and connections declarations

Abstract syntax of μComp-lang (2)

A node of the AST is annotated with an annotation

```
type ('a, 'b) annotated_node = { node : 'a; annot : 'b }
[@@deriving show, ord, eq]
```

- Node field is the underlying syntactic element, e.g., expr
- Annot field represents a generic annotation, e.g., position, type, ...
- The annotation [@@ deriving show, ord, eq] is used by the deriving ppx to automatically generate a useful function for the type, i.e., comparison and string conversion

Abstract syntax of µComp-lang (3)

Nodes are built by instantiating the annotated_node record

```
type 'a expr = ('a expr_node, 'a) annotated_node

and 'a expr_node =
  | LV of 'a lvalue (* x or a[e] *)
  | Assign of 'a lvalue * 'a expr (* x=e or a[e]=e *)
  | ILiteral of int (* Integer literal *)
  | CLiteral of char (* Char literal *)
  | BLiteral of bool (* Bool literal *)
  | UnaryOp of uop * 'a expr (* Unary primitive operator *)
  | Address of 'a lvalue (* Address of a variable *)
  | BinaryOp of binop * 'a expr * 'a expr (* Binary primitive operator *)
  | Call of identifier option * identifier * 'a expr list (* Function call f(...) *)
[@@deriving show, ord, eq]
```

µComp-lang 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 in base 10 or digits in base 16 prefixed with `0x` (integers are 32bit values)

32, 1024, 3232, 0xFF

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: var, if, return, else, while, int, char, void, bool, interface, uses, provides, component, connect, def, for

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  */
```

Grammar (1)

```
CompilationUnit ::= TopDecl* EOF

TopDecl ::= "interface" ID "{" IMemberDecl+ "}"

          |   "component" ID ProvideClause? UseClause? "{" CMemberDecl+ "}"

          | "connect" Link ';'

          | "connect" "{" (Link ";")* "}"

Link ::=  ID "." ID "<-" ID "." ID

IMemberDecl ::= "var" VarSign ";" | FunProto ";"

ProvideClause ::= "provides" (ID ",")* ID
```

Grammar (2)

UseClause ::= "uses" (ID ",")* ID

VarSign ::= ID ":" Type

FunProto ::= "def" ID "("((VarSign ",")* VarSign)? ")" (":" BasicType)?

CMemberDecl ::= "var" VarSign ";" | FunDecl

Fundecl ::= FunProto Block

Block ::= "{" (Stmt | "var" VarSign ";")* "}"

Type ::= BasicType | Type "[" "]" | Type "[" INT "]" | "&" BasicType

Grammar (3)

BasicType ::= "int" | "char" | "void" | "bool"

Stmt ::= "return" Expr? ";" | Expr ";" | Block | "while" "(" Expr ")" Stmt

| "if" "(" Expr ")" Stmt "else" Stmt | "if" "(" Expr ")" Stmt

| “for” “(“ Expr? “;” Expr? “;” Expr? “)” Stmt

Expr ::= INT | CHAR | BOOL | "(" Expr ")" | "&" LValue | LValue "=" Expr | "!" Expr

| ID "(" ((Expr ",")* Expr)? ")" | LValue | "-" Expr | Expr BinOp Expr

LValue ::= ID | ID "[" Expr "]"

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

Grammar (4)

- The grammar is ambiguous
- Tokens with no semantic values are enclosed in “”
- Tokens with semantic values are capitalized, e.g., ID
- The parsing phase produces an located AST where names are not qualified, if the program is syntactically correct, an error otherwise.