μcomp-lang Compiler Report

Languages, Compilers and Interpreters
Università di Pisa

Francesco Gargiulo (627052)
f.gargiulo2@studenti.unipi.it
August 19, 2022

This report describes the incremental development of my implementation of the μcomp-lang compiler. Each chapter is dedicated to a specific assignment (Parser, Semantic Analysis, Component Linking and Code Generation, Language Extensions), explaining the corresponding design and implementation choices.

1 Parser

The μcomp-lang parser uses ocamllex for implementing the scanner and menhir for the parser. The parser takes as input the source code and, if the input is syntactically correct, produces an untyped abstract syntax tree (AST) as an intermediate representation; otherwise, it generates an error.

1.1 Scanner

Given an input stream of characters, the scanner produces a stream of tokens. The *scanner.mll* file contains the scanner implementation, which is composed by an header containing util functions, the regular expressions used to represent sets of characters, and the scanner functions used to match input characters against valid patterns and generate the corresponding tokens.

Scanner Functions

The *scanner.mll* file defines four scanner functions: `next_token`, `comment`, `line_comment`, and `character`. The main function is `next_token` which, in order to produce a token, tries to match the current characters against valid patterns. If there is no matchable valid pattern then a lexing error is raised. As the names suggest, `comment` and `line_comment` are used to consume input characters which correspond to multi-line comments and single-line comments, respectively. Lastly, `character` is used to recognize literal characters. In particular, it defines the rules needed to handle escape sequences.

Lexer Buffer Position

In order to provide precise tokens locations, every time there is a new line character the lexer buffer `lexbuf` position must be updated by using the procedure `Lexing.new_line` on it.

Reserved Keywords and Identifiers

Identifiers and μcomp-lang reserved keywords can be matched by the same pattern. For this reason an hash table is used to represent the set of language-reserved keywords in order to decide whether to generate a keyword token or an identifier token. Moreover, identifiers can have a maximum length of 64 characters: the function `check_id_length` is used to raise a lexing error when an identifier exceeds its length limit.

Negative 32bits Integers

μcomp-lang integers are 32bit values. Decimal literals cannot be used to directly represent negative values; this is because, for example, "-42" is interpreted as two tokens: unary minus and integer literal. The range for 32bit integers is [-2147483648, 2147483647], this means that -2147483648 cannot be directly represented as a decimal value because it would be interpreted as `MINUS` and `LINT(2147483648)`, generating an overflow. As a solution it is possible to use the corresponding hexadecimal representation 0x80000000.

1.2 Parser

The parser consumes tokens produced by the scanner and, using grammar rules, tries to produce a rightmost derivation. The *parser.mly* file contains the parser implementation, which is composed by an header containing util functions, token declarations, and the grammar specification.

Parsing State

The header of the *parser.mly* file defines a mutable record `compilation_unit_m` that is used to maintain the program's interfaces, components, and connections found during parsing.

```
type compilation_unit_m = {
  mutable interfaces_m : Location.code_pos interface_decl list;
  mutable components_m : Location.code_pos component_decl list;
  mutable connections_m : connection list;
}
```

At the end of a successful parsing, this record is used to create the resulting located AST.

Representing For Loops as While Loops

μcomp-lang supports for-loops but the AST doesn't. For this reason, the header of the *parser.mly* file uses the util function `convert_for_to_while` to convert a generic for loop:

```
for(expr1; expr2; expr3){
  statements
}
```

to the semantically equivalent while loop:

```
expr1
while(expr2){
  statements
  expr3
}
```

Solving the Dangling Else Problem

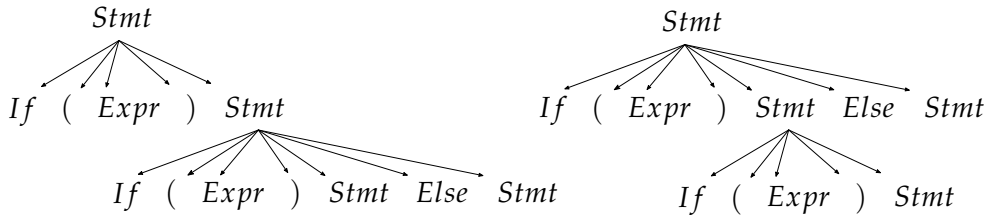
The μcomp-lang grammar specification is ambiguous because the rules for the if-then-else construct allow for multiple rightmost derivations. Consider the production rule for statements:

$$
\begin{aligned}
\mathit{Stmt} ::= {} & \text{"return"}\ \mathit{Expr}?\ \text{";"} \\
& \mid \mathit{Expr}?\ \text{";"} \\
& \mid \mathit{Block} \\
& \mid \text{"while"}\ \text{"("}\ \mathit{Expr}\ \text{")"}\ \mathit{Stmt} \\
& \mid \text{"if"}\ \text{"("}\ \mathit{Expr}\ \text{")"}\ \mathit{Stmt}\ \text{"else"}\ \mathit{Stmt} \\
& \mid \text{"if"}\ \text{"("}\ \mathit{Expr}\ \text{")"}\ \mathit{Stmt} \\
& \mid \text{"for"}\ \text{"("}\ \mathit{Expr}?\ \text{";"}\ \mathit{Expr}?\ \text{";"}\ \mathit{Expr}?\ \text{")"}\ \mathit{Stmt}
\end{aligned}
$$

It is possible to generate a string of terminal and nonterminal symbols:

$$\text{"if"}\ \text{"("}\ \mathit{Expr}\ \text{")"}\ \text{"if"}\ \text{"("}\ \mathit{Expr}\ \text{")"}\ \mathit{Stmt}\ \text{"else"}\ \mathit{Stmt}$$

that allows two distinct partial derivation trees:



This results in a shift/reduce conflict as soon as the lookahead token is `ELSE`: the automaton can either shift (leftmost derivation tree) the `ELSE` or it can reduce (rightmost derivation tree) the production corresponding to the rightmost derivation tree's last level.

Our desired behavior consists in associating each *else* to the innermost unmatched *if* (leftmost derivation tree). Thus, the *parser.mly* defines a `%prec NOELSE` annotation to assign to the if-then production a lower precedence than the if-then-else production.

2 Semantic Analysis

During the semantic analysis phase, the located AST produced by the parser is analyzed to check types and names usage. If there aren't errors, the semantic analyzer converts the input located AST to the equivalent typed AST.

2.1 Symbol Table

In order to achieve an efficient semantic analysis, a symbol table is used to store the information derived during the visit of the located AST. Moreover, a symbol table allows to manage nested scopes: a new symbol table node is created each time we enter a new lexical scope. The nodes are linked together in an order that corresponds to the lexical nesting levels.

For example, consider the following μcomp-lang component:

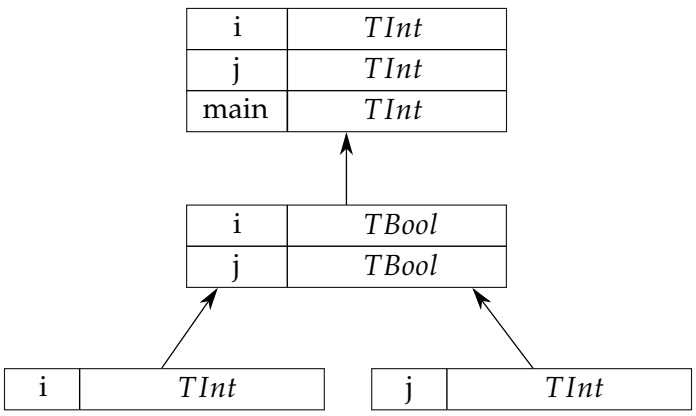
```
component EntryPoint provides App {
  var i : int;
  var j : int;

  def main() : int {
    var i : bool;
    var j : bool;

    { var i : int; }
    { var j : int; }

    return 0;
  }
}
```

The following symbol table allows us to handle nested scopes:



The search for the information associated with a given key starts at the current node and continues up to the root. The search stops if either the key is found or if it is established that the scope doesn't contain it.

Symbol Table Implementation

A symbol table node is either an empty node or a node containing an hashtable for its corresponding lexical scope and a reference to the node's parent.

```
type 'a t =
  | Empty
  | Scope of 'a t * (Ast.identifier, 'a) Hashtbl.t
```

It is possible to initialize an empty table by using the homonymous function. This function returns an empty node, which is the root of any symbol table.

```
let empty_table = Empty
```

When entering a nested scope, the `begin_block` function allows to create a new symbol table node and attaches it to its parent node.

```
let begin_block table = Scope(table, Hashtbl.create 0)
```

When exiting a nested scope, the `end_block` function allows to get the parent node back.

```
let end_block table = match table with
  | Empty -> Empty
  | Scope(parent, _) -> parent
```

It is possible to add an entry to the current symbol table node if the node is not an empty node and if the identifier has no information already associated.

```
let add_entry symbol info table = match table with
|Empty -> failwith "Error: can't add a symbol to an empty node"
|Scope(p,htbl) ->
  if Hashtbl.mem htbl symbol then raise (DuplicateEntry(symbol))
  else Hashtbl.add htbl symbol info;
  Scope(p, htbl)
```

When looking for the information associated with a certain key *id*, the search starts in the current node *t*. If *t* contains *id* then the corresponding value is returned; if *t* doesn't contain *id*, then the search continues into *t*'s parent, and so on. This allows for variables declared in inner scopes to hide variables with the same name declared in outer scopes. If the search arrives at the root table which doesn't either contain *id*, then *id* is not present in the scope. The symbol table interface was modified in order to allow `Option` results to be returned.

```
let rec lookup id t = match t with
  | Empty -> None
  | Scope(p,htbl) ->
      if Hashtbl.mem htbl id then Some(Hashtbl.find htbl id)
      else lookup id p
```

The `of_alist` function creates a new symbol table node, attached to an empty root node, containing the input (key, value) associations.

```
let of_alist (entries : (Ast.identifier * 'a) list) =
  let add_entry table (id,entry) = add_entry id entry table
  in let etable = begin_block empty_table
  in List.fold_left add_entry etable entries
```

2.2 Semantic Analysis

The Environment

The semantic analysis (contained in the *semantic_analysis.ml* file) defines a type to represent the environment, which consists of four different symbol tables:

```
type env = {
  interfaces : ((identifier * identifier * typ
                 * Location.code_pos) list * Location.code_pos)
                 Symbol_table.t;
  components : (identifier list * identifier list
                 * Location.code_pos)
                 Symbol_table.t;
  functions : (identifier option * typ * Location.code_pos)
                 Symbol_table.t;
  variables : (identifier option * typ * Location.code_pos)
                 Symbol_table.t;
}
```

- The *interfaces* symbol table maps each interface identifier to a tuple containing the interface's location, and a list of (*interface_id*, *memberdecl_id*, *memberdecl_typ*, *memberdecl_pos*) describing its members. This information is used to retrieve the members of some interfaces when visiting a component in order to setup the component's scope.
- The *components* symbol table maps each component identifier to a tuple containing its location, the list of interfaces it uses, and the list of interfaces it provides.
- The *functions* symbol table maps each function identifier to a tuple containing the (optional) interface of the function, the function `typ` (`TFun`), and its location. The function `typ` contains the list of the formal parameters `typs` and the returning `typ`.

- The *variables* symbol table maps each variable identifier to a tuple containing the (optional) interface of the variable, the variable `typ`, and its location.

During the AST traversal, a mutable record is used to retrieve and modify the environment at any moment:

```
type mutable_env = {mutable env: env}
let globalenv = ref {env=init_env}
```

The Execution Flow

There are two main functions driving the semantic analysis:

- `check_interface` performs the semantic analysis of a single interface node and returns the corresponding typed node. Moreover, it adds to the environment the information about interface members.
- `check_component` performs the semantic analysis of a single component node and returns the corresponding typed node. The function first checks the used and provided interfaces, then it sets up the component environment by beginning a new lexical scope and adding all the functions and variables declared in its used interfaces. The component's scope also contains its functions definitions to allow mutual recursion.

During the analysis there are several util functions used to check that specific semantic properties are respected. In particular three mutually recursive functions `type_of_expr`, `type_of_lval`, and `rvalue_of_ref`, are used throughout the analysis to semantically check any expression and to retrieve its `typ` in the given environment.

Removing Unreachable Instructions

After the semantical check of functions, the `remove_unreachable` function is used to remove instructions that are impossible to reach. This will simplify the code generation phase. The util function `has_return` checks if a block has a return statement that is always executed. The key intuition is that instructions are unreachable if the statement preceding them satisfies one of the following conditions:

- the statement is a return
- the statement is a block containing a return instruction that is always executed
- the statement is an if-then-else such that both branches contain an always executed return instruction

The `remove_unreachable` function is also recursively applied to blocks contained in other statements.

3 Component Linking and Code Generation

The goal of this phase is to produce the LLVM IR code that corresponds to the typed AST generated by the semantic analysis. This is accomplished by two consecutive sub-phases: component linking and code generation.

3.1 Component Linking

In the input typed AST, external names are qualified with their interface identifiers. We now want to swap each of these interface identifiers with the identifiers of the components providing such interfaces. The *linker.ml* file contains the functions used to achieve this.

First of all, we need to generate the information needed for the linking: the `build_env` function is used to create a symbol table mapping each component to a tuple containing its list of used interfaces, and its list of provided interfaces. The `create_provides_table` function creates a symbol table mapping each interface identifier to the component providing it.

Then, the `check_connections` is used to check that all the connections are well-formed. The main util function it uses is `check_connection`, which ensures that each individual connection contains no errors.

Finally, the `wire_component` function is applied to each component to change the interface associated to external names with the component implementing such interface. If a function call has no external interface, then it is associated with the component it is contained into; the reason for this is to simplify name mangling during code generation.

3.2 Code Generation

The final step consists in converting the typed (and linked) AST to LLVM IR code.

Name Mangling

The LLVM module will contain, within a unique namespace, all the global variables and functions names defined in the components. Consider a component named "MyComp": to avoid collisions between names defined in distinct components, each function and global variable name defined in MyComp will be preceded by the string *_MyComp_*.

For this reason the standard library functions defined in *rt-support.c* have been renamed `_Prelude_getint` and `_Prelude_print`.

The Execution Flow

During the code generation, an environment is used to retrieve the llvalue associated to functions and variables names.

```
type env = {
  functions : L.llvalue Symbol_table.t;
```

```
  variables : L.llvalue Symbol_table.t;
}
```

The high-level execution flow is the following: components global variables are added to the environment and to the LLVM module as global variables. Standard library functions prototypes are added to the environment and declared in the LLVM module. Components function prototypes are added to the environment and defined in the LLVM module to allow for mutual recursion. Finally, component function bodies are processed and added to the LLVM module.

Handling Short-Circuit Evaluation

When evaluating a Boolean operator (`AND`, `OR`), the left-hand side expression `e1` is always evaluated, while the right-hand side expression `e2` is evaluated only if necessary. To implement this behavior, we create three blocks and their corresponding builders:

```
let left_block = L.append_block context "SC.l" func in
let right_block = L.append_block context "SC.r" func in
let merge_block =  L.append_block context "SC.merge" func in
let left_builder = L.builder_at_end context left_block in
let right_builder = L.builder_at_end context right_block in
let merge_builder = L.builder_at_end context merge_block
```

Jump from the current block to the block where the left-hand side expression will be evaluated:

```
add_terminator builder (L.build_br left_block)
```

Evaluate the left-hand side expression:

```
let e1_eval = build_expr env func left_builder e1
```

Depending on the boolean operator, define the conditional jump:

```
let jump_instruction =
  match binop with
  | And -> L.build_cond_br e1_eval right_block merge_block
  | Or -> L.build_cond_br e1_eval merge_block right_block
in add_terminator left_builder jump_instruction;
```

`right_block` is reached only if the left-hand side expression evaluation is not enough to determine the value of the Boolean operator.

Evaluate the right-hand side expression:

```
let e2_eval = build_expr env func right_builder e2 in
add_terminator right_builder (L.build_br merge_block)
```

Finally, we use a `build_phi` instruction to get the Boolean operator value depending on which path was traversed. Given that the right- and left-hand side expressions might contain more subexpressions, we first get the blocks where their builders are:

```
let left_phi_block = L.insertion_block left_builder in
let right_phi_block = L.insertion_block right_builder in
let phi = L.build_phi ([(e1_eval, left_phi_block);
                        (e2_eval, right_phi_block)])
                      "" merge_builder
```

Finally, the original builder is positioned at the end of the merge block:

```
L.position_at_end merge_block builder
```

Compiling and Testing

Given an input file *main.mc*, it is possible to compile the source code and generate the file *main.bc* containing the corresponding LLVM IR using the instruction:

dune exec bin/mcompc.exe -- main.mc

Then, using clang it is possible to compile the standard library contained in *rt-support.c* and link it with the LLVM IR to obtain the executable program

clang main.bc bin/rt-support.c -o main

The compiler has been tested manually with all the files in *test/samples* and also automatically using the *test/testall.sh* script.

4 Language Extensions

This section contains the language extensions implemented after the core functionalities were completed and tested. Each extension has a corresponding *test-name.mc* and *test-name.out* in *test/samples* so that they are automatically executed by *test/testall.sh*.

Do-While Construct

The Do-While construct has the following syntax:

```
do{
    statement
}
while(expr);
```

and it has been implemented by simply appending (at parsing time), inside a block, a while after the always-executed statement:

```
{
    statement
    while(expr){
        statement
    }
}
```

Prefix/Postfix Increment/Decrement

The AST `expr_node` type has now a new constructor to handle prefix/postfix increment/decrement:

```
Increment of inctype * 'a lvalue * int32
```

The `inctype` allows to indicate if an increment is of prefix or suffix type.

Code generation implements this new functionality by updating the memory location and returning the old or new value depending on the operator.

Because the language now supports prefix decrements, expressions like `--42` contained in *test-ops2.mc* are no longer allowed because prefix decrement expects a lvalue. For this reason, the expression `--42` contained in *test-ops2.mc* was replaced by `-(-42)`.

Compound Assignment

The `Assign` constructor of the `expr_node` type has been modified to specify the type of assignment:

```
Assign of assigntype * 'a lvalue * 'a expr
```

where

```
type assigntype = Eq | Plus | Minus | Times | Div | Mod
```

Compound assignments can only applied to `TInt` types. Code generation implements this functionality by evaluating the `lvalue` (which can contain an expression), evaluating the `expr`, applying the given operation and storing its result.

Variable Declaration with Initialization

Both the constructors for local and global variables have been extended to accommodate an optional initial expression:

```
LocalDecl of vdecl * 'a expr option
VarDecl of vdecl * 'a expr option
```

The syntax for declaring a variable with an initial variable is the following:

```
var n=42 : int;
```

Code generation handles a local declaration by first defining the variable and then generating an assignment expression that is passed to the function `build_expr`, which handles such operations.

Global variables can be initialized only with constant values: initialization expressions cannot contain variables or function calls. Complex expressions such as `4*10+2` are handled using the functions for constant expressions defined in the `Llvm` Ocaml module.

Inheritance Among Interfaces

The interface builder has been modified to support the list of extended interfaces:

```
InterfaceDecl of {
  iname : identifier;
  declarations : 'a member_decl list;
  extends : identifier list;
}
```

It is possible to create multiple levels of inheritance between interfaces using the following syntax:

```
interface A extends B,C {...}
interface B {...}
interface C extends D {...}
interface D {...}
```

This functionality has been implemented during the semantic analysis phase: the `solve_interfaces_inheritance` function creates a graph where interfaces are nodes and edges (u, v) indicate that u extends v. The function uses a depth-first search on such graph to detect eventual cycles, which are prohibited. Then each interface u is modified to add all the inherited members contained in the subtree rooted in u.