$\mu Comp\text{-}lang$

Alessandro Puccia
Matricola 547462
Department of Computer Science

June 2022

Index

Chapter 1

Introduction

μComp-lang is a simple imperative language where programs are built out from components and interfaces.

Interfaces represent a sort of types for components and specifies the signatures of functions and variables that the components providing that interface need to implement. Interfaces are also fundamental to compose components since interactions between two components can happen only using the functions of its interfaces. A component that provides interfaces **must** implement its functions with the same signature and also must define the same variables. There are also two special predefined interfaces: *Prelude* and *App*. In particular *Prelude* cannot be provided by any component and *App* can be provided by only one component.

Components are the main units of code in *μComp-lang*. A component defines its own local scope and has its state. For each component, there is only one instance at runtime, that is, components are not similar to classes but to singleton objects. Typically, components *provides* and *uses* an arbitrary number of interfaces. The provided interfaces represent the functionalities that a component implements and makes available to others. Whereas, the used interfaces represent a sort of dependencies, that is, the functionalities required by the component to achieve its goals. Other functions/variables that are defined by a component are only local to its scope. Any component also uses the interface *Prelude* in an implicit way (the programmers can still explicitly use it).

Once defined our components they need to be wired together in order to make a program. It is possible to link components by using multiple `connect` keywords or a `connect` block.

Besides components and interfaces, the base *μComp-lang* has standard constructs like `if`, `while` and `for`. It supports integers, characters and booleans as primitive data. Arrays are the only form of compound data. Functions can only returns values of primitive data or nothing.

References in *μComp-lang* are similar to the ones of other programming languages. They store addresses of primitives variables and are automatically dereferenced depending on their usage. Array references can only be defined as argument of functions.

Chapter 2

Scanning & Parsing

The **goal** of this phase is to use *OCamllex* and *Menhir* to produce an annotated *Abstract Syntax Tree* where the annotation is a structure of type `code_pos` (with members *start* and *end line*, *start* and *end column*) that will be used in the following phases.

µComp-lang supports only 32 bits integers that can be expressed either in a decimal or hexadecimal way. To check that the programmer only uses 32 bits integer literals, I used the function `of_string` provided by module `Int32`. In this way if there is an overflow/underflow it is enough to catch an exception.

During scanning other checks are done:

- Identifiers starts with a letter or an underscore and then can contain letters, underscore and numbers.
- The programmer has to specify a single character between `'`.
- Only special characters allowed are: `\'`, `\b`, `\t`, `\f`, `\\`, `\r` and `\n`.
- The programmer cannot declare identifier with names that are more than 64 characters long and that are equal to reserved keywords.
- The programmer cannot declare nested comment blocks.
- The programmer has to terminate comment blocks (if the scanner reach an `EOF` before closing a comment block then it raises an error).

The grammar for the base language (showed in appendix A, generated with *Obelisk*) was slightly modified by adding a new rule called `⟨no_multidim⟩` because the original one allowed the possibility to declare multi-dimensional arrays and to declare functions that accept them in input. Some language extensions also required to add new rules to the grammar.

The grammar had problems related to *shift-reduce* conflicts. To solve them I did the following:

```
for (init; cond; step)
    body
```

(a) How an user uses the `for` construct.

```
init;
while (cond) {
    body;
    step;
}
```

(b) How it is transformed after parsing.

- To address the problem of dangling `else` I added a new precedence rule called `then_prec` with less precedence than `else` token.
- To address the problem of the missing precedence level for production ⟨`expr`⟩ → ⟨`expr`⟩ ⟨`bin_op`⟩ ⟨`expr`⟩ I used the `%inline`[1] keyword. With this solution, *Menhir* replaces each occurrence of ⟨`bin_op`⟩ with its definition without manually transforming the grammar.

The parser implements only one check: variable names must respect the *lower camel case* notation and function/component/interface names must respect the *upper camel case* notation.

2.1 `for` loops implementation

To implement the `for` construct I decided to not extend the abstract syntax tree but instead to represent it in terms of the `while` construct. This gives the advantage that I do not need to implement an additional pattern in the following sections.

To do this, I used two additional symbol tables since a generic `for` (showed in fig. 2.1a) construct will be transformed into a block that is formed by the variable initialization followed by a `while` (showed in fig. 2.1b). I need to take also into account the variable modification and so I introduce another block (and hence another symbol table) that will contain the body and the step instruction part.

2.2 Incremental parsing & error handling

Menhir offers the possibility to use an *incremental API* by using the flag `--table`. In this case, the parser does not have access to the lexer but whenever it needs a token it stops and returns its current state to the user. The user then is responsible to invoke the lexer to get the next token and resume the parser.

The parser is started by calling the function `compilation_unit` provided by the module `Incremental` passing the current position of the lexer. The function constructs a starting

[1] Note that this keyword can be used also to gain some performance when there is a rule that can be seen as an alias for another one (for example the rule ⟨`block`⟩ is an alias of *Menhir* internal rule `delimited`).

```
compilation_unit: INTERFACE WHILE
##
## Ends in an error in state: 1.
##
## top_declaration ->
##   INTERFACE . ID LCBRACK nonempty_list(i_member_declaration)
##   RCBRACK [ INTERFACE EOF CONNECT COMPONENT ]
##
## The known suffix of the stack is as follows:
## INTERFACE
##

This is a reserved keyword and cannot be used
as a name of an interface.
```

Figure 2.2: Example of an entry inside `.messages` file.

point that has type `checkpoint` that can be used to drive the parser with the function `offer` and `resume`.

The current state of the parser is used only to provide error handling. In the traditional handling system, *Menhir* provide a special `error` token that can be used within a production. This token is used to replace the current lookahead token when an error is detected. The behaviour of the parser in this case changes if in the current state there is a shift or reduce action (or if no action, simply the parser reject the input).

Exploiting the *Incremental API* when an error is detected instead, it is possible to take much different actions (for example modifying the input stream of tokens). One possibility is to select a diagnostic message based purely on the current state of the automaton. The parser can determines, ahead of time, which are the error states by using the flag `--list-errors`. This produces (if outputted into a file) a `.messages` file that can be used to define an error message for each error state (in fig. 2.2 an example of an entry). Then, compiling this file with the flag `--compile-errors`, generates a module. Normally *Menhir* writes the code of the module to `stdout` but exploiting the `with-stdout-to` macro[1] provided by `dune` it is possible to redirect the code into a module that will be used for error handling.

The general limit of this approach is that each diagnostic message is associated with a state and not a sentence. Each entry in the `.messages` file contains a sentence that lead to an error in a given state. This is just an example that reach the error state but there may exist many other sentences that can cause an error. So, when writing these diagnostic messages I tried to make them more generic to the state ignoring the example sentence.

[1] To do this I modified the `dune` file inside `/lib` adding a new rule. This file was also modified to include a new library called `easy_logging` for debugging purposes.

Chapter 3

Semantic analysis

The **goal** of this part was to create an *AST* that is annotated with values of type `typ` and where the member names of a component are annotated with the name of the interfaces that declare them.

3.1 Symbol table

The symbol table (in fig. 3.1) is defined in the module `Symbol_table`. It defines a new type `'a t` that can be:

- either a dummy type (`Dummy`).
- or a block that is constructed with `Table` passing as arguments the parent block and an hash table that will contain the names of the current scope.

This implementation allows to represent a stack of blocks so that newly defined names will be searched first in the top block allowing to shadow the previously defined ones.

The original interface that defines the values and functions of the symbol table was slightly modified:

- `empty_table` is implemented as a function with zero arguments (in *OCaml* this means that will take `unit` as single parameter) and returns a new hash table.
- function `of_alist` receives an additional argument that represent another symbol table where the entries of the list passed as second argument will be inserted.

3.2 Semantic checks

Given an annotated `compilation_unit` first I create a global scope that will contain interfaces and components then I start checking both of them. In particular, for each component the following checks are needed:

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

Figure 3.1: Implementation of the symbol table

- Interface `Prelude` is not provided and `App` is provided by only one and not used.
- No interface repetitions inside `use` and `provide` clause. Only interface names are allowed inside these clauses.
- If a component provides two or more interfaces that declare members with the same name then these need to have also the same types.
- There must not be definitions with the same name between used interfaces and between used and provided interfaces.
- All functions and variables declarations of provided interfaces must be implemented.

It is not possible to declare `void` variables or pass them to functions. When declaring an array it is expected to have a *size* that is at least one. Each function accepts basic, compound and reference types (in particular arrays are always passed by reference without using an explicit `&`) and can return only basic type arguments. Function overloading is not allowed, if a component defines a function that has the same name of one declared in a provided interface then it *must* have the same type. In each function a new symbol table (with parent the symbol table of the corresponding component) is defined. Each following block will define a new symbol table.

For other *AST* constructs, the following checks are performed:

- `If` and `While` must have a boolean condition.
- Binary operations must have the same types for both the addends. `And` and `Or` allow only boolean types. `Add`, `Sub`, `Mult`, `Div`, `Mod`, `Greater`, `Less`, `Geq` and `Leq` allows only numerical types.
- `Neg` allows only numerical types, `Not` allows only boolean type.
- In `Assign`, references are automatically dereferenced if there is as left value a *basic type*. If there is as left value a *reference type* then:
  - if the right expression is a *reference type* then the left value will point to the address resulting from the evaluation of the right expression.
  - if the right expression is a *basic type* then the assignment changes the memory location pointed by the left value with the result of the evaluation of the right expression.

- In `Call`, passed arguments type must match formal arguments type (also in this case considering auto-dereferencing). Also, the identifier passed must be a function.

Chapter 4

Linking

The **goals** of the linking phase are:

- Check that the connections specified by the programmer are *meaningful*. A general connection `ID1.ID2` ← `ID3.ID4` is *meaningful* when: `ID1` and `ID3` are names of two different components, `ID2` is the name of an interface *used* by `ID1`, `ID4` is the name of an interface *provided* by `ID3`, `ID2` and `ID4` are the same interface.
- Check that there aren't overwriting connections.
- Check that for each component there is a connection for *all* interfaces it uses, if any.
- Rewrite the typed *AST* in such a way that the external names are qualified with the name of the component specified in the connections.

During the checks on the links, I store each one inside an hash table of hash table, in particular the first hash table is indexed by the component name while the second one is indexed by the name of the interface that provide that definition. This structure allowed me to recover easily the component name that provides a particular function or variable.

Then, I simply recurred over each function to qualify the variables and function calls with the components names that provides them.

Chapter 5

Code generation

The **goal** of code generation phase is to exploit the *LLVM API* and transform the *typed AST* into an *LLVM module*. This module is made of global variables and function declarations thus, to represent the possibility that different components could have members with the same name, I used *name mangling*. In this way, a generic global variable (or function) with name `id` defined inside a component `Cname` will have its name mangled into `__Cname_id` (following `C++` mangling style). Since I used this technique I modified the runtime support offered in file `rt-support.c` and the file `mcomp_stdlib.ml`.

I took care of dead code after a `return` statement by deleting it after the generation. To do so I iterated on all blocks (by using the function `iter_blocks`) and then, for each block, I created a list of instruction (by using the function `fold_left_instrs`) that will contain all the instructions that follows a type `Ret` instruction.

Since there is not a `skip` instruction in the *LLVM API* I need to take care of the situation where there are empty simple blocks (for example there is an `if` without an `else`). I decided to delete these blocks: first I identify them by using the function `fold_left_blocks` and then iteratively call the function `delete_block`.

5.1 Variable initialization

In the base language variables cannot be initialized but, just to be more safe during execution, when the user declare a variable (so I need to generate one of the `alloca` instructions) I also generate a `store` instruction that will store a *default value*.

This value depends on the type of the variable but in general I can exploit the function `const_null` that when passed a `lltype` returns its *null* value. This means that basic types will be initialized always with 0 (that for `char` type means the *null* terminator). For compound types I consider the type of the elements while for reference types I use the function `const_pointer_null`.

5.2 Shortcircuiting `&&`, `||`

To handle the problem of shortcircuiting I first generate the instructions that are related to the first expression then I create two blocks `t` and `f` that are used in the following way:

- Considering `&&`, if the first expression is evaluated `true` then jump to `t` block and start generating instructions related to the second expression (and at the end jump to `f` block) otherwise jump to `f` block.
- Considering `||`, if the second expression is evaluated `false` then jump to `f` block and start generating instructions related to the second expression (and at the end jump to `t` block) otherwise jump to `t` block.

Since now there are two flows, I exploited the φ node to determine the final expression value and to continue generating instructions.

Chapter 6

Language extensions

In this phase, it was requested to enhance *μComp-lang* with at least two extensions. I choose to implement `do-while` loops, pre/post increment/decrement operators, abbreviation for assignment operators and floating point arithmetic. Other tests where added to assess that these extensions will be translated correctly.

6.1 Floating point arithmetic

To implement this enhancement, I added a new token `T_FLOAT` (that takes as argument a `Float` number of 64 bits) and extended the `Ast` module by adding a new constructor `TFloat` for type `typ` and `FLiteral` for type `expr`. Also in this case it will be possible to express a floating point number with an hexadecimal form.

In semantic analysis I implemented other checks for binary, unary, pre/post increment/decrement operators and in the code generation phase I added other pattern matching cases that allowed to build floating point instructions.

6.2 `do ... while` loops

For this enhancement, I decided to extend the *AST* by adding a new construct called `DoWhile` that takes a `stmt` value (the body) and an `expr` value (the condition).

The parser is modified by adding a new token called `DO`. Both scanner and parser are modified by adding new rules accordingly.

During semantic analysis I performed, like for the `While` construct, the check to assess that there is a condition of boolean type.

The main difference is in the code generation phase because, differently from the `While` construct, it is requested to execute at least one iteration. So first jump to the basic block that represents the body and only after executing its instruction jump to the basic block where there is the condition testing instructions.

```
Assign(l, BinaryOp(Add, LV(l) <@> pos, e) <@> pos)
```

Figure 6.1: Construction of a `code_pos` annotated `Assign`, `l` is a `lvalue` type and `e` is a `expr` type.

6.3 Pre/post increment/decrement operators

Also in this case, I added multiple definitions inside the `Ast` module:

- A new type called `dop` with constructors `PreIncr`, `PreDecr`, `PostIncr` and `PostDecr`.
- A new construct for type `expr` called `DoubleOp`. It takes in input a value of type `dop` and a value of type `lvalue`.

In semantic analysis I checked that these new operations can only be used with numerical types and then in code generation handled the following situations:

- When using `PreIncrement`, `PreDecrement` then build the instruction `add` or `sub` (`fadd` or `fsub`), store the resulting value and return the *updated value*.
- When using `PostIncrement`, `PostDecrement` then I need to build the instruction `add` or `sub` (`fadd` or `fsub`), store the resulting value and return the *old value*.

6.4 Abbreviation for assignment operators

This is a straightforward enhancement since each abbreviation can be rewritten by using an assignment. In this case I do not need to add new constructs into the AST, modify the semantic analysis or the code generation but only to modify the scanner and the parser by adding new tokens called `PASSIGN` (`+=`), `MINASSIGN` (`-=`), `TASSIGN` (`*=`), `DASSIGN` (`/=`), `MODASSIGN` (`%=`) and the corresponding rules.

Then during the parsing phase, whenever I find one of these tokens I construct an `Assign` passing the left value and a `BinaryOp` that depends on the token reduced from rule `⟨abbr_assign⟩`. In fig. 6.1 I report an example for the `+=` operation.

Appendix A

Language grammar

⟨*compilation_unit*⟩ ::= ⟨*top_declaration*⟩* 'EOF'

⟨*top_declaration*⟩ ::= interface 'ID' '{' ⟨*i_member_declaration*⟩+ '}'
| component 'ID' [⟨*provide_clause*⟩]
[⟨*use_clause*⟩] '{' ⟨*c_member_declaration*⟩+ '}'
| connect ⟨*link*⟩
| connect '{' ⟨*link*⟩* '}'

⟨*link*⟩ ::= 'ID' '.' 'ID' '<-' 'ID' '.' 'ID' ';'

⟨*i_member_declaration*⟩ ::= var ⟨*var_sign*⟩ ';'
| ⟨*fun_prototype*⟩ ';'

⟨*provide_clause*⟩ ::= provides 'ID'$^{+}_{,}$

⟨*use_clause*⟩ ::= uses 'ID'$^{+}_{,}$

⟨*var_sign*⟩ ::= 'ID' ':' ⟨*type_*⟩

⟨*fun_prototype*⟩ ::= def 'ID' '(' ⟨*var_sign*⟩$^{*}_{,}$ ')' [':' ⟨*basic_type*⟩]

⟨*c_member_declaration*⟩ ::= var ⟨*var_sign*⟩ ';'
| ⟨*fun_declaration*⟩

⟨*fun_declaration*⟩ ::= ⟨*fun_prototype*⟩ ⟨*block*⟩

Grammar generated with command: `obelisk latex -i -o grammar.tex -syntax lib/parser.mly`

⟨*block*⟩ ::= '{' ⟨*block_content*⟩* '}'

⟨*block_content*⟩ ::= ⟨*stmt*⟩
| var ⟨*var_sign*⟩ ';'

⟨*type_*⟩ ::= ⟨*basic_type*⟩
| ⟨*no_multidim*⟩ '[' ']'
| ⟨*no_multidim*⟩ '[' 'T_INT' ']'
| '&' ⟨*basic_type*⟩

⟨*no_multidim*⟩ ::= ⟨*basic_type*⟩
| '&' ⟨*basic_type*⟩

⟨*basic_type*⟩ ::= int
| float
| char
| void
| bool

⟨*stmt*⟩ ::= return [⟨*expr*⟩] ';'
| [⟨*expr*⟩] ';'
| ⟨*block*⟩
| while '(' ⟨*expr*⟩ ')' ⟨*stmt*⟩
| if '(' ⟨*expr*⟩ ')' ⟨*stmt*⟩ else ⟨*stmt*⟩
| if '(' ⟨*expr*⟩ ')' ⟨*stmt*⟩
| for '(' [⟨*expr*⟩] ';' [⟨*expr*⟩] ';' [⟨*expr*⟩] ')' ⟨*stmt*⟩
| do ⟨*stmt*⟩ while '(' ⟨*expr*⟩ ')' ';'

⟨*expr*⟩ ::= 'T_INT'
| 'T_FLOAT'
| 'T_CHAR'
| 'T_BOOL'
| '(' ⟨*expr*⟩ ')'
| '&' ⟨*l_value*⟩
| ⟨*l_value*⟩ '=' ⟨*expr*⟩
| ⟨*l_value*⟩ ⟨*abbr_assign*⟩ ⟨*expr*⟩
| '!' ⟨*expr*⟩
| 'ID' '(' ⟨*expr*⟩$^{*}_{,}$ ')'
| ⟨*l_value*⟩
| '-' ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*bin_op*⟩ ⟨*expr*⟩
| ⟨*l_value*⟩ '++'

Grammar generated with command: `obelisk latex -i -o grammar.tex -syntax lib/parser.mly`

| ⟨*l_value*⟩ '`--`'
| '`++`' ⟨*l_value*⟩
| '`--`' ⟨*expr*⟩

⟨*l_value*⟩ ::= '`ID`'
| '`ID`' '`[`' ⟨*expr*⟩ '`]`'

⟨*bin_op*⟩ ::= '`+`'
| '`-`'
| '`*`'
| '`%`'
| '`/`'
| '`&&`'
| '`||`'
| '`<`'
| '`>`'
| '`<=`'
| '`>=`'
| '`==`'
| '`!=`'

⟨*abbr_assign*⟩ ::= '`+=`'
| '`-=`'
| '`*=`'
| '`/=`'
| '`%=`'

Grammar generated with command: `obelisk latex -i -o grammar.tex -syntax lib/parser.mly`