μcomp-lang compiler
Project report

Innocenzo Fulginiti
ID: 594051
i.fulginiti@studenti.unipi.it

a.a. 2021/2022

The project consists of implementing a compiler for the didactic programming language *μcomp-lang*. In this report are discussed the main decisions made for the implementation of the compiler and it's given a brief introduction of the language.

This report is structured in the following sections:

1. Presentation of the programming language μcomp-lang: in this section is given a description of the grammar of the language and its main features.
2. **Parsing phase**: in this section is discussed the implementation of the scanner and the parser.
3. **Semantic analisys phase**: in this section are explained the main decisions made to perform the semantic analisys check, according to the semantic and typing rules of the language.
4. A discussion about the **linking phase**.
5. A discussion about the implementation of **code generation phase**, where it has been used the **LLVM** infrastructure.

1 μcomp-lang language

μcomp-lang is a *component-based* imperative language. A program, in μcomp-lang, is written linking together different *components*, that represent the basic unit of a μcomp-lang program. Components use and provide *interfaces*. In an interface are declared global variables and functions to be implemented by the component that provides it. Furthermore, components are statically linked each other via their interfaces.
In each μcomp-lang there must be *one and only one* component that provides the standard library interface `App` where it's defined the method `main():int`. The function `main` is the entry point of every program.

In order to better understand the role of interfaces and components, in Listing 1 it's given an example of a program. We can point out four parts of the code of the sample:

- (*Lines 1-4*) Definition of the interface `IAddSub` which defines the two functions `add` and `sub`.
- (*Lines 6-13*) Definition of the component `AddSub` that provides the interface `IAddSub`. In this component is provided the implementation for the functions `add` and `sub` declared in the interface `IAddSub`.
- (*Lines 15-24*) The component `Sample` represents the entry point of the program, since it provides the interface `App` and so, implements the method `main`. This interface also uses `IAddSub`, indeed in lines 21-22 are called the functions declared in `IAddSub`.
- (*Line 27*) This instructions *links* `Sample.IAddSub` with `AddSub.IAddSub`, this means that for `Sample` the implementations of the functions declared in the used interface `IAddSub`, is given by the component `AddSub`.

Listing 1: Example of a μcomp-lang program

```
1  interface IAddSub {
2      def add(x : int, y : int) : int;
3      def sub(x : int, y : int) : int;
4  }
5
6  component AddSub provides IAddSub {
7      def add(x : int, y : int) : int {
8          return x + y;
9      }
```

```
10      def sub(x : int, y : int) : int {
11          return x - y;
12      }
13  }
14
15  component Sample provides App uses IAddSub {
16      def main(): int {
17          var a : int;
18          var b : int;
19          a = 5;
20          b = 3;
21          print(add(a, b));
22          print(sub(a, b));
23          return 0;
24      }
25  }
26
27  connect Sample.IAddSub <- AddSub.IAddSub
```

1.1 Syntax of the language

The whole syntax of μcomp-lang has been given to us in the project specification, but we have been asked to extend the language with at least two more constructs, the extensions made in this project are:

- the *do-while* loop;
- the pre/post increment/decrement operators (‘++’, ‘−−’);
- the assignment operators: ‘+=’, ‘-=’, ‘*=’, ‘/=’, ‘%=’.

In Listing 2, it’s shown the extended formal grammar, expressed in EBNF notation.

Listing 2: Extended grammar for μcomp-lang

```
1  CompilationUnit ::= TopDecl* EOF
2
3  TopDecl ::= "interface" ID "{" IMemberDecl+ "}"
4      | "component" ID ProvideClause? UseClause? "{"
           CMemberDecl+ "}"
5      | "connect" Link ';'
6      | "connect" "{" (Link ";")* "}"
7
8  Link ::=  ID "." ID "<-" ID "." ID
```

```
9
10 IMemberDecl ::= "var" VarSign ";" | FunProto ";"
11
12 ProvideClause ::= "provides" (ID ",")* ID
13
14 UseClause ::= "uses" (ID ",")* ID
15
16 VarSign ::= ID ":" Type
17
18 FunProto ::= "def" ID "("((VarSign ",")* VarSign)? ")" (":"
       BasicType)?
19
20 CMemberDecl ::= "var" VarSign ";"  | FunDecl
21
22 Fundecl ::= FunProto Block
23
24 Block ::= "{" (Stmt | "var" VarSign ";")* "}"
25
26 Type ::= BasicType | Type "[" "]" | Type "[" INT "]" | "&"
      BasicType
27
28 BasicType ::= "int" | "char" | "void" | "bool"
29
30 Stmt ::= "return" Expr? ";" | Expr? ";" | Block
31      | "while" "(" Expr ")" Stmt | "do" Stmt "while" "("
            Expr ")" ";"
32      | "if" "(" Expr ")" Stmt "else" Stmt  | "if" "(" Expr
            ")" Stmt
33      | "for" "(" Expr? ";" Expr? ";" Expr? ")" Stmt
34
35 Expr ::= INT | CHAR | BOOL | "(" Expr ")" | "&" LValue |
      LValue "=" Expr | "!" Expr
36      | ID "(" ((Expr ",")* Expr)? ")" | LValue | "-" Expr |
            Expr BinOp Expr
37      | "++" Expr | Expr "++" | "--" Expr | Expr "--"
38      | LValue BinOpAss Expr
39
40 LValue ::= ID | ID "[" Expr "]"
41
42 BinOp ::= "+" | "-" | "*" | "%" | "/" | "&&" | "||" | "<" |
       ">" | "<=" | ">=" | "==" | "!="
43
44 BinOpAss ::= "+=" | "-=" | "*=" | "/=" | "%="
```

In order to add the extensions in the language, the file `ast.ml`, contain-

ing the definition of the abstract syntax tree of mcomp-lang, has been updated with the constructor `DoWhile of 'a expr * 'a stmt` for the type `'a stmt_node` and with the constructor `IncDec of 'a lvalue * incdecop` for the type `'a expr_node`, where `incdecop` is a type defined as

```
1 type incdecop = PreInc | PostInc | PreDec | PostDec
```

2 Parsing phase

This phase represents the first step for the realization of the compiler and it consists, basically, in implementing the *scanner* (or *lexer*) and the *parser*. The parsing phase produces a located *abstract syntax tree* (AST) where the names are not qualified.

2.1 Scanner

The scanner takes in input the source program to be compiled and produces a stream of tokens that will be analysed from the parser. The scanner has been implemented using **Ocamllex** and the implementation is realized in the file `scanner.mll`.
In the *header* have been defined two hash tables: `keyword_table` that contains all the *keywords* of the language and `special_char_table` that contains the *escape* characters of μcomp-lang. Furthermore, in the header, there are two constants for the maximum length of identifiers and for the maximum integer value.
In the *main section* of the scanner have been defined the regular expressions and three *rules*:

- `next_token` that is the entrypoint rule that produces all the tokens;
- `single_line_comment` to recognize the single line comments;
- `multi_line_comment` that is the rule that recognizes the multi line comments.

2.2 Parser

The parser takes the stream of tokens produced by the scanner and produces the located AST. The parser is implemented using **Menhir** and the implementation can be found in the file `parsing.mly`.

At the beginning of the file `parsing.mly` are specified all the tokens of the language with their precedences. Then, all the rules are defined.

In order to produce the correct AST, in this phase, there are particular ambiguous situations that must be managed.
We may have ambiguous interpretations with the construct **if-then-else**. For example, the statement `IF e_1 THEN IF e_2 THEN s_1 ELSE s_2`, may be interpreted in two ways as showed in Listing 3.

Listing 3: Ambiguous if-then-else interpretation

```
1  /*
2  Ambiguous statement: IF e_1 THEN IF e_2 THEN s_1 ELSE s_2
3  */
4
5  // First possible interpretation
6  if (e_1) {
7      if (e_2) {
8          s_1;
9      }
10 }
11 else {
12     s_2;
13 }
14
15 // Second possible interpretation
16 if (e_1) {
17     if (e_2) {
18         s_1;
19     }
20     else {
21         s_2;
22     }
23 }
```

We would like to obtain the second interpretation. In order to disambiguate cases like that, the rule for statements in the parser, is defined as

```
1  stmt:
2      .
3      .
4      .
5      | IF LR_BRACKET e = expr RR_BRACKET s1 = stmt ELSE s2 =
           stmt
6        { make_node (If(e, s1, s2)) (to_code_position $loc) }
```

```
7       | IF LR_BRACKET e = expr RR_BRACKET s1 = stmt %prec
          THEN
8       { make_node (If(e, s1, (make_node (Skip) (
            to_code_position $loc)))) (to_code_position $loc)
            }
9   ;
```

exploiting the precedence order given to the tokens `THEN` and `ELSE`. In Listing 4 are reported all the precedence rules specified in the parser for the tokens.

Listing 4: Precedence and associativity specification

```
1  %nonassoc THEN       /* Lowest precedence */
2  %nonassoc ELSE
3  %right ASSIGN
4  %left OR
5  %left AND
6  %left EQ NEQ
7  %nonassoc GT LT GEQ LEQ
8  %left PLUS MINUS
9  %left TIMES DIV MOD
10 %right MINUSMINUS
11 %nonassoc UMINUS
12 %nonassoc NOT        /* Highest precedence */
```

Other situations of ambiguty worth discussing, are the ones caused by the unitary operator minus ('−'). Problems with the operator minus, could arise with both the subtraction binary operator and with the pre-decrement operator ('−−').
The ambiguity with the subtraction operator is solved with the rule

```
1  | MINUS e = expr %prec UMINUS
2      { make_node (UnaryOp(Neg, e)) (to_code_position $loc) }
```

giving to the operator minus a higher precedence.
While, the case with the pre-decrement operator is solved considering '−−' as the pre-decrement operator if it is followed by a variable, as two minus operators otherwise.

After the parsing phase, if syntactic errors are not found, is produced a AST where nodes are annotated with the corresponding code position inside the source program.

3 Semantic analysis

In this phase, the compiler performs semantic checks on the abstract syntax tree obtained as a result from the parsing phase, according to the semantic and typing rules given in the project specification.
The result produced after this phase is an AST whose nodes are annotated with their type if the program is semantically correct, otherwise is raised an error.

The implementation is contained in the file `semantic_analysis.ml`. Furthermore, for this phase a *symbol table* was realized, whose implementation can be found in the file `symbol_table.ml`.

3.1 Symbol table

This symbol table is used to map correctly names with the corresponding declaration among the different scopes in a program. μcomp-lang is a *statically scoped* language.

The symbol table is defined in OCaml as

```
1 type 'a t =
2     | Empty
3     | Node of 'a t * (Ast.identifier, 'a) Hashtbl.t
```

and is implemented as a stack of scopes (`Nodes`) where the scope represented by the top node is nested in the previous one and so on.
Each node of the symbol table consists of a pair where the first element is the parent node, while the second element is a *hash table* that maps an identifier into its declaration.
For the symbol table are implemented the functions specified in the given signature file `symbol_table.mli` (Listing 5).

Listing 5: Signature for the symbol table module

```
1 type 'a t
2
3 val empty_table : 'a t
4 val begin_block : 'a t -> 'a t
5 val end_block : 'a t -> 'a t
6 val add_entry : Ast.identifier -> 'a -> 'a t -> 'a t
7 val lookup : Ast.identifier -> 'a t -> 'a
8 val of_alist : (Ast.identifier * 'a) list -> 'a t
```

The function `lookup` search a symbol in a symbol table. It starts searching in the top node and recursively checks in the parent nodes until the symbol is found. If the node `Empty` is reached, it means that the symbol wasn't present in the symbol table and so a `NotFound` exception is raised.

3.2 Semantic analysis

This phase consists of visiting the AST checking if the semantic and typing rules are respected and recreating the nodes of the AST annotated with the correspondent type.

The analysis starts initializing the symbol table with the global declarations. For the symbol table, a new type has been defined, in order to keep information about the symbols. This new type is called `symbols` and has four constructors:

- `Interface_sym`. Used for interface symbols and keeps as information: the name of the interface, the type, its own symbol table and the code position.
- `Component_sym`. Used for component symbols. It maintains the same information also saved for interfaces, with two more symbol tables: one for the used interfaces and one for the provided.
- `Fun_sym`. It represents a function symbol. It keeps as information: the function name, the type, the code position and its symbol table.
- `Var_sym`. It represents a variable and keeps as information the name, the type and its code position.

The global table is created with the function `create_global_table` performing the following steps:

1. Gets the symbols of the μcomp-lang standard library interfaces `App` and `Prelude` (with their local declarations), using the function `get_mcomp_stdlib`.
2. Invokes the function `get_interfaces` to take the symbols of the interfaces declared in the program, the local member declarations are saved too.
3. Gets the symbols of the components of the program, with their local declarations, calling the function `get_components`.

After the creation of the global symbol table, the function `check_components` is called in order to ensure that, for each component, are respected the semantic rules for the used and provided interfaces as requested in the semantic rules. It is also checked that there is exactly one component that provides the interface `App`, that the interface `App` is never used and that the interface `Prelude` is never provided.

At this point is performed a visit in the entire AST in order to annotate all the nodes with their type. For this purpose the functions `annotate_interfaces` and `annotate_components` have been defined.
For the functions defined inside the components is performed a visit on the statements in their body, using the function `check_stmt`. For the expressions is ensured that they are well typed according to the typing rules of the language. The check for expression is executed by the function `check_expr`.

4 Linking phase

In this phase the compiler links the components of a program. After the semantic analysis, the AST obtained has the external names of variables and functions qualified with the used interface. In the linked phase the external names will be qualified with the name of the component specified in the connections.
Furthermore, in this phase is performed a check on the correctness of the connections as requested in the project specification.
The implementation of the linking phase can be found in the file `linker.ml`.

5 Code generation phase

This phase represents the last step of the compilation process. The code generation phase consists in emitting the LLVM code using the LLVM API for OCaml. The implementation of this step is contained in the file `codegen.ml`.

In order to generate the LLVM code a visit on all the component nodes on the AST resulting from the linking phase is performed.

It has been adopted the suggested idea to add the definitions of all the components in the global namespace, using the *name mangling* technique to ensure that each component memeber has a different name. After the global module has been initialized, the following operations are performed

for each functions defined in the program:

1. The LLVM bitcode for the instructions in the body is generated.
2. After the generation of the bitcode, in each LLVM basic block all the LLVM instructions after a return are removed.
3. Some operations are executed in order to guarantee that each basic block has a terminator instruction. If a basic block doesn't contain a terminator instruction, it is added by distinguishing two cases:
   - if the basic block is inside a function whose return type is void, a `Llvm.build_ret_void` instruction is added at the end of the basic block;
   - if the basic block is inside a function whose return type is different than void, a `Llvm.build_ret` instruction is added, and the return value is the default value for the function return type.
4. For functions whose return type is different than void, it must be guaranteed that at most one LLVM return instruction is present. In order to do that, a new basic block `ret_merge` is added at the end of the function. The block `ret_merge` contains a phi node, whose value represents the right value to be returned by the function. The idea is to perform the following operations for the basic blocks containing a return instruction:
   (a) get the return value;
   (b) delete the LLVM return instruction;
   (c) add an instruction that jumps to the block `ret_merge`;
   (d) add the return value to the *incoming* values of the phi node in `ret_merge`;