Languages, Compilers and Interpreters

MicroC

Marco Antonio Corallo

531466

Contents

Introduction

MicroC is a statically typed subset of the language *C*.
The main simplification of this language with regards to C are:

- Primitive data types: it supports only integers `int`, characters `char` and booleans `bool`;
- Data structures: it supports only one-dimensional arrays as data structures and pointers as compound data types;
- Heap: there is no support for dynamic allocation of memory;
- Functions: functions can only return `void`, `int`, `bool` or `char` values;
- There is no pointer arithmetic, there is no pointer to functions and pointers and arrays are not interchangeable;
- There is no support for separate compilation: all the code must stay in a unique compilation unit;

Despite these simplifications, MicroC is a *Turing-complete* language, so it is expressive enough to implement complex systems. It provides two library functions to interact with the user:

- `void print(int n) // prints n to the stdout`
- `int getint() // get an integer from the stdin`

It also provides all types of cycles: `for`, `do-while` and, naturally, `while`. There is support for the initialization of variables during the declaration and for multiple declarations.
This implementation of MicroC also provides the operators for pre/post-increment `(++)`, pre/post-decrement `(--)` and for the abbreviate form of assignments `(+=, -=, *=, /=, %=)`.
Furthermore, MicroC provides a strong(er than C) type system and a

static analysis that recognizes and eliminates dead code.
This stuff will be discussed in the *Language Extensions* chapter, but we note here that some of the simplifications we made allowed us to define a stronger type system for a safer language. Are these restrictions or opportunities?
The main purpose of this project is to implement a compiler for MicroC using the *OCaml* programming language and a set of tools provided by it, in order to learn and explore the techniques, the solutions and the state-of-the-art technologies for developing a compiler. For this purpose, the project assignment required to use *Ocamllex* and *Menhir* to implement respectively the scanner and the parser of the language and the *LLVM* toolchain for generating and optimize the code.
In the following chapters we discuss the main choices we did during the design and the development of the MicroC compiler. First, in chapter 2 and 3 we talk briefly about the development of the lexer and the parser, then in the chapter 5 we present the semantic analysis of the code, that is the type checking and the dead-code elimination. In the end, in chapter 6 we discuss the code generation through the LLVM toolchain.
We conclude with an overview of the chosen language extensions in chapter 7 and the results of a large number of test-cases in chapter **??**.

Lexer

As we introduced in the previous chapter, the *lexer* is generated using *Ocamllex*, a *Lex*-based tool provided from Ocaml.
The lexical elements of MicroC follow the syntax of C.
Identifiers starts with a letter or an underscore and then can contain letters, underscore and numbers. I.e.:

```
['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

32-bit integer literals are sequence of digits in base 10 or digits in base 16 prefixed with `0x`, i.e.:

```
['0'-'9']+ | ("0x"|"0X") ['0'-'9' 'A'-'F']*
```

Character literals are single-quoted and boolean literals are `true` and `false`. The keywords of the language follow the same syntax of identifiers, so they are stored in an hashmap that is checked before the recognition of the string as identifier. In this way we keep the generated automaton a little smaller.
The lexer recognizes also the escape sequences recognized from C and both single-line and multi-line C-style comments.

Parser

For the development of the parser has been used *Menhir*, a LR(1) parser generator for Ocaml.
The parser builds the AST of the program, annoting each node with the position of that element in the source file.
The given C-based grammar was ambiguous and has been disambiguated introducing a new token `THEN` that imposes to the rule a lower priority than the `ELSE` rule. In this way, the usual scenario

```
if(e1) c1
    if(e2) c2
    else
```

is resolved in favor of the second `if`, associating the `else` with the nearest `if`.
The *shift/reduce* conflict is thus resolved in favor of shifting.
When the sequence of tokens for the `for` statement is read, the parser reassemble the three expressions of the guard to build a `while` statement.
The syntactical category of binary operations is *inlined* to avoid the usual arithmetic shift/reduce conflict. The `%inline` keyword causes all references to that syntactical category to be replaced with its definition.
Another decision that had to be made is how to handle the value `NULL`. A first attempt was to treat it like a special character, for example using its ASCII code (`'\000'`). However, we have no guarantees that the users don't use that character for their MicroC programs. So we choose to treat the value like an access to a system variable `NULL`. Thus the AST node for the value NULL looks like

```
Access( AccVar("NULL"); $loc ) |@| $loc
```

The parser has been extended for supporting the language extensions introduced in chapter 1; this will be discussed in chapter 7.

Symbol Table

In order to implement the semantic analysis (and the code generation then) we need a *symbol table.*
For the implementation of the symbol table we followed a well-known solution that is a list of hash tables. The symbol table acts like a *stack* of *scopes*, where the head of the stack is the current scope and the last block is the global scope. The implementation of the symbol table follows the provided interface and provides methods to

- create an empty table;
- insert a new block (*scope*) into the table;
- remove a block from the table;
- insert an entry `<var, informations>` in the current block of the symbol table;
- looks for a given variable;

If an access to a variable is made, that variable is looked up in the table, going from the current scope to the global scope, block by block, until it is not found.
If a block starts, then a block is added to the symbol table, thus a new scope starts. In this way the required static scope is implemented.
The informations binded to an identifiers into the symbol table are undefined: the data structure is generic and the user can specify the type of the information when he creates a new table.

Semantic Analysis

The goal of this part of project is to implement a semantic checker for types an name usage.
MicroC adopts a static scoping, so there may be *shadowing* of identifiers.
The semantic analysis phase returns the AST of the program, if all checks are successful. For the semantic analysis phase, the symbol table binds to identifiers a set of informations

```
type info = {
  sort  : sort;
  ty    : nullable_typ;
  params: nullable_typ list;
} and sort = Fun | Var
```

`sort` tells if the symbol refers a variable or a function; in the first case, `ty` represents the type of the variable, otherwise represents the return type of the function. In the last, `params` is a list of types that represents the types of the parameters of the function. Naturally, if the identifier refers a variable, params will be the empty list.
The type of the fields `ty` and `params` is `nullable_typ`, that is a *wrapper* type for the type `Ast.typ`, with a special constructor `Null`. It will be the type of the value `NULL`.
The main function of the module creates a new symbol table, into which adds the entries for the system variable `NULL` and for the system functions `print` and `getint`. It executes then the *dead-code elimination* and passes the *pruned* AST to the type-checker. The dead-code pass will be discussed in chapter 7.
Let's we start introducing a few type rules for MicroC:

- Conditional guards in `if` and `while` statements expect boolean values;

- Logical operators expect only boolean values;

- Variables of type `void` are not allowed;
- Functions can return only `int`, `bool`, `char` and `void`;
- Arrays should have a size of at least 1 element when declared; an array can be unsized when it is passed as parameter to a function;
- Array cannot be assigned;
- multi-dimensional arrays are not supported;

Briefly, we defined a type-check routine for almost all AST nodes. The type-checker is invoked on the program, that is a list of *top-declaration*. Each declaration may be either a *global variable declaration* or a *function* declaration. We discuss the former in chapter 7, since the corresponding routine has been modified to allow initialization-in-declaration and multiple declaration. We just say that the checker adds all global symbols to the global scope of the symbol table.
For functions, the type checker first checks that the type rules are respected (that is, the return type and the parameters type respect them), then the checker adds the function's information into the symbol table and checks the body of the function.
The body of a function is a *block*, that is a list of local declaration and statements. Again, the former will be better discussed in chapter 7; as for the the latter, instead, the analyzer performs type checks into a new scope that contains the informations about the parameters passed to the function.
A statement can be

```
statement:
  | If expr s1 s2
  | While expr s
  | Return expr
  | Expr expr
  | Block b
```

For `if` and `while` statements, the analyzer recognizes the type of the conditions and checks that it is *boolean*. Then it checks recursively the statements of the branches.
When the analyzer meets a `Return e` statement, it checks if the type of `e` matches the return type of the function. `e` is an expression, that is

```
expression:
  | CLiteral | BLiteral | ILiteral
  | Not e | Neg e
  | e1 + e2 | e1 * e2 | ...
  | Access a
  | Assign a e
  | Addr a
  | Call f params
```

For literals it doesn't do checks, except for integer literals, for which it checks that the value is not a value for *overflow* or *underflow* occurr; if it is, the analyzer emits a *warning*.
For the unary and binary operators, the analyzer performs checks on the types of operands: **arithmetic** operations can be applied only to two **integer** values, **logical** operators can be applied only to **boolean** values and **comparison** operators must be applied to two operands of the **same type**.
For the assignment expression, the type-checker gets the types of the variable to be assigned and of the expression to assign and checks that these two types match and they are not arrays.
The type of a variable is an access to the symbol table; plus, the type-checker has to perform some trivial checks in the cases of *dereferencing* a pointer or *subscripting* an array.
As last case of expressions, there is `Call f params`.
For this case, the analyzer gets the information about `f` from the symbol table and checks that the type of the expressions passed as *actual* parametes match the type of the *formal* ones.
In the end, since MicroC does not support separate compilation, the analyzer checks that exists a function with the signature of the main function, that can be

```
    | int main()
    | void main()
```

Code Generation

The code generation phase makes use of the *LLVM toolchain* to compile a MicroC program, that is an AST, to a *three-address IR* that will be then compiled to LLVM *bitcode*.
We note that the AST that the semantic analyzer passes to the code-generator is a reduced AST: the analyzer already performed the dead-code elimination on the original AST before to pass it to the code-generator.
For this phase, we use a symbol table that binds identifiers to `llvalue`, that are *"Any value in the LLVM IR. Functions, instructions, global variables, constants, and much more are all llvalues"*. In the global scope of the symbol table are added the built-in functions `print` and `getint`, that are binded to the corresponding function llvalue.
As for the semantic analysis, the code-generator preceeds with a standard traversal of the AST, generating first the code for defining global variables and functions and adding them into the symbol table. Then it scans the body of the functions.
Here, the generator creates a new scope into which adds the function parameters, after allocating (and storing) them into the stack.
Then the generator iters on the statements (and the local declarations) of the function.
For `if` statements, the generator checks if there is an else-statement: that is, it is not an empty block or an empty statement (`;`). So it knows if generating an `if-then-else` or an `if-then`. This is a little optimization the compiler does. The `if-then-else` statement generates three new *basic blocks* and three *branch* instructions. The `if-then` statement, instead, only two basic blocks and two branch instructions.
The generation of a statement can cause the generation of a terminator instruction in the end of the block. If it happens, the generator must avoid the generation of a branch instruction. Indeed, there cannot be more than one terminator instruction in a block, because the first one should be in the middle of the block. The generator makes these

checks during the generation of an `if` (but also `while`) statement.
The generator of expressions and accesses use an option parameter that tells if it should generate only an access to a variable or also a load of its value. For example, when we want to generate code for an assignment

```
    x = e;
```

we want the address of x, in order to store the value of e into it. As for the case we want to generate an `Addr` operation (`&`): we want the address of the variable.
A more complex case is when we want to generate a subscript operation (`[ ]`): the base operand can be an array or a pointer, if it is the result of a `cast` operation of a function parameter. So the generator must check and choose the correct `gep` instruction.
The `bitcast` instruction is generated when an array is passed as argument to a function. It casts the arrays to pointers and pointer of array to pointer of array of zero elements `[0 x type]`.
In the end, we implemented logical `and` and `or` with *short-circuit evaluation* rules using a couple of basic blocks and the `phi` instruction to merge the two branches.

Language Extensions

Until now, the design and the development of the compiler has been based on the given modules. The representation and implementation of the `NULL` value, for example, exploits a dummy system variable to avoid a change of the given AST. For some of the following extensions, instead, we need to modify the AST.

Declaration with Initialization and Multiple Declarations

In order to implement this feature we need to modify the given AST. The nodes for local and global declaration become

```
topdecl_node =
  | Fundecl of fun_decl
  | Vardec of ((typ * identifier) * (expr list)) list
stmtordec_node =
  | Dec of ((typ * identifier) * (expr list)) list
  | Stmt of stmt
```

A `((typ * identifier) * (expr list)) list` is a list of declarations, each one with a type, an identifier and a list of initial values. Indeed, in order to implement this feature both for variables and arrays, we treated all initial values as array of constant values; i.e.:

```
    int a = 1;
    becomes
    int a = {1};
```

To do this, in the parser we added some syntactical category that compute the type of each inline declaration and maps a list of `(vardesc, initexpr)` to a list of `(typ, initexpr)`. For declaration without an initialization, an empty list is provided. For declaration of variables

with a constant initial value, a *singleton* list is provided.
In this procedure the parser also converts unsized array with an initial value to sized array, exploiting the length of the initializer. Following the semantic of C and the behaviour of Clang, it doesn't care if an array is sized with a different size than the initializer. A declaration like

```
A[5] = {1,2}
```

is still a declaration of an array of five elements and the elements without an initial value will be initialized to 0.
Instead, a declaration like

```
A[2] = {1,2,3,4,5}
```

has an undefined behaviour. It may overlap the memory region of another variable in the same address space, or it may cause a *buffer overflow*.
The type-checker performs checks about the two length and emits a warning if they are different.
For global variable declarations, the analyzer also checks that the provided initial values are literals. The semantic of C requires that to optimize exploiting compile-time constants and operations on them. And so we did.

Do-While

In order to implement the `do-while` statement, we easily added a rule for the statement syntactical category of the parser, that *"desugare"* it, producing an equivalent `while` statement, as we did for the `for` statement.

Pre/Post Increment/Decrement Operators

The first approach to these features exploited the *syntactic sugar* of these operators, desugararing them as follows:

```
i++; becomes i = i+1;
```

However, in testing and studying the AST of the product code we have noticed that this equivalence is not always true. The counterexample

is `A[i++]++` whose AST is `A[i = i+1] = A [i = i+1] + 1`.
To avoid this we extended the AST, introducing three new constructors:

```
| Postincr of access
| Postdecr of access
| BinaryOpEq of binop * access * expr
```

While there are constructors for post-increment and post-decrement operations, there are no for pre-increment or pre-decrement. That because the semantic of these operators is the same of that of the `BinaryOpEq` with the corresponding arguments. I.e.:

```
    ++i becomes i += 1
```

So the parser desugare the operations of pre-increment and pre-decrement and builds `BinaryOpEq` nodes.
The type-checker performs checks on the type of the arguments, that must be integer. The code-generator produces the expected instructions; that is, for post-increment and post-decrement operations, it performs the operation and the assignment of the updated value, then returns the original one.

Dead-Code Elimination

The dead-code elimination phase prunes the AST, removing dead statements or blocks of code and eventually making some simple code transformation.
First of all, define an *empty statement* like a statement that is

- An empty block;
- A block with only empty statements;
- A while statement when the condition is `false`;
- An if statement when the condition is `true` and the `then-branch` statement is empty;
- An if statement when the condition is `false` and the `else-branch` statement is empty;

Exploiting this definition, it performs a step of analysis for recognizing dead statements. This pass returns a pair `(AST, b)`, that are the pruned statement and a boolean value that tells if the statements performs a **sound** `return` instruction. It means that the analysis doesn't remove an instruction if it could be reached; it only prune instructions that we are secure that won't be executed.
In this way we can *backpropagate* this information to the parent node of the AST, and perform a more precise pruning.
The analyzer can do some simple code transformations, for example the code `if(e); else;` becomes `e;`.
In order to make the user aware that some code transformation happens, when the analyzer recognizes dead-code, it emits a warning.

Test cases

The compiler has been tested on the provided large battery of test-cases, using a script `testall.sh` that evidences eventual differences between the expected output and the actual one. All the correct test cases works and the output match the expected one, except for the file *text-ex7.mc*, that executes an infinite loop, and the file *test-ops2.mc*, that attempts to use the pre-decrement operator on an integer literal. The former compiles, the compilation of the latter is instead stopped at the type analysis.
The fail-driven test cases correctly fail, except for the two *fail-dead.mc* files, whose dead-code has been removed from the analyzer, and the two *fail-int* files. For overflow and underflow, indeed, the compiler just emits a warning.
The compiler has been also tested on many others test-cases, since the design and the development of the project followed the *test-driven* paradigm.
Has been tested the operations on `NULL` values, on characters, on the short-circuit logical gates, on pointers to arrays and on arrays of pointers;
All the language extensions has been tested and seems to work as expected, although *"program testing can be used to show the presence of bugs, but never to show their absence"*!

References

[1] URL: http://pages.di.unipi.it/gori/Linguaggi-Compilatori2022/.

[2] URL: https://github.com/lillo/compiler-course-unipi.

[3] URL: https://github.com/lillo/compiler-course-unipi/tree/main/microc.

[4] URL: https://ocaml.org/.

[5] URL: http://gallium.inria.fr/~fpottier/menhir/manual.html.

[6] URL: https://llvm.moe/ocaml/.

[7] URL: https://clang.llvm.org/.