Compilation

Megi Dervishi

January 12, 2020

1 Code Structure.

The compiler is made up of the following files:

Filename	Dependencies	Brief Description
compiler.ml	go_ast.mli x86_64.ml go_lexer.mll	The compiler takes the AST generated from the Parser and converts it into Assembler x86. Assuming that the code is well typed.
go_ast.mli	smap_definer.ml	The Ast file defines all the types and syntax that are used throughout the files.
go_lexer.mll	go_parser.mly	The go_lexer.mll transforms go code into a sequence of tokens that can be fed to the parser.
go_parser.mly	go_ast.mli	The go_parser.mly takes a tokens from the lexer and transforms them into an AST and also performs a syntax check.
go_typer.ml	go_lexer.mll, graph.mll	The typer takes the AST from the parser and applies the go-language's rules to it to check that the code is well typed.
graph.ml		This is a helper file to detect cycles in the typer.
main.ml		The main file checks for user input, takes all the other files and puts them together in order to compile a go script.
smap_definer.ml		Dummy file to define a map.
x86_64.ml		Allows for easy writing of assembler.
Makefile		`make` compiles and runs ./test.go, `make tests` runs all the standard tests.

2 Compiler.

The compiler works in the following way.

2.1 General Assembly Code Principles.

All variables are stored from an offset stored in *rbp*. Whenever we call a function the output is sent to *rax*. For binops the values are passed respectively in *rax* and *rcx*. All other registers are used for temporary storage. One last specificity is that the registers: *rbp*, *rbx*, *r12*, *r13*, *r14*, are callee-saved and reset to their respective values after the called function returns. This is especially useful for *rbp*, since in a way *rbp* represents our 'environment', it allows for the variables of a called function not to mix with the variables of the callee. When calling a function all the variables of the current environment are copied over to a second one which is passed to the function and upon return the environment is reset to its original value.

2.2 Ocaml Code Structure.

2.2.1 Data structures.

Variables are kept track of in 2 Hashtables *vars.vartable*, *vars.vartypes*. The first maps the variable's id to their offset from *rbp* (i.e. "i" $\rightarrow$ 8 means that "i" is stored in 8(*rbp*)). The second maps a variable to its type. Types are encoded using the parser's syntax (which seemed a good idea at first and turned out to be a real hassle). Possible types are:

Eident id $\leftrightarrow$ Struct of name id; *Econst const* $\leftrightarrow$ any constant
Eunop(Pointer, t) $\leftrightarrow *t$ where t is a type

Similarly structs are kept track of in another hashtable *structs* which maps a struct id to it's size, and two maps who send the attribute's ids to their offsets and types respectively. The input ids and return types of functions are stored in a Hashtable as well. Finally there is one last table to keep tracked which offset (specifically for pointers) are nil or not. Then there are a few counters in order to give labels distinct names: *nb_strings, nb_ifelse, nb_for, nb_vars, nb_printbool.*

2.2.2 Code flow.

There are two distinct parts to the compiler. The first takes the information that has been computed in the typer in order to allocate memory offsets to all the variables, add all the information to the aforementioned structures and returns a list of expression ('compiler types') which contain these information. For example, an *Eident "a"* (corresponding to variable "a" in the go code) will first add "a" to our Hashtables, find a free offset for it and return *Cident (offset, "a").* Once all this is done the now modified AST is sent to the 'real' compiler. This section will now transform all these given expressions into assembly code, taking care of updating the relevant information as it proceeds (e.g. the nil table). Once this is done the program terminates.

2.2.3 Main functions.

Allocation.
For the allocation the main functions are: *alloc* `env`; *alloc_expr* `fid is_type e`; *alloc_instr* `fid instr`; *add_var* `fid id target_e list_id`. The first is simply the 'main' function of the allocation section, it takes care of allocating first all the structs, then all the functions and their inputs and finally the body of these functions. The structure here is overall very similar to the one in the typer. The next two functions allocate (resp.) expressions and instructions. Finally the last function is a helper function that takes care of adding a variable to our tables knowing it's 'target' expression. What we mean by this is that when we say `x := *y` then `*y` is the target of `x`.

Compilation.
For the compilation the main functions are: *compile_program* `env ofile`; *compile_expr* `cexp`; *compile_instr* `cinstr`. These are the exact correspondent of the first 3 functions of allocation. The first is the function that is called from the main file and which takes care of putting everything together. The second and third transform expressions and instructions into assembly code.

Noteworthy helper functions.
Some important helper functions are: *compile_struct* `strct`; *add_to_nil* `offsets` ; *get_e_type/type_ce* `exp/cexp`. The first is used to get all the information we have on a given struct from the typer and transfer it to the compiler. The second is used to check whether a given variable is nil and add it to the table. The third/fourth compute the type of an expression but for parser expressions and compiler expressions respectively.

2.3 Difficulties, Remarks and Improvements.

There were multiple challenges in writing the compiler. The simplest yet most annoying one is that I was not very familiar with Ocaml and Assembly before this course/project, but more specifically for the compiler the main difficulty was to find a way to manipulate some complex structures using only registers, the best example are functions. It took me a very long time to find a way to implement functions in assembly. Mainly because normally I was always dealing with single valued data or pointers and therefore all the 'transfer of information' from one Ocaml function to another was always done in *rax*. However when implementing functions since functions can have multiple returns I couldn't find a simple way to transfer that data from a function to another. Finally I managed by splitting a function in pieces when allocating it, so that one function call that yields n returns would become an explicit list of n compiler instructions. The compiler is very far from being efficient mainly due to my limited Ocaml skills. When writing the compiler I was not very preoccupied with efficiency. A first obvious improvement is that it is probably not necessary to create a full copy of all our variables every time we call a function. It is a terrible waste of time and space but I did not really have the time to worry about that. We also never call the `free` function which of course is not viable in the long run.
There are still a few issues with the compiler. They are all due to the management of nils. The compiler needs to keep track of what pointers are nil or not in order to print them out correctly. However when we assign a parameter in the following way:

`x = foo(),   x := foo(),   var x = foo()` and `x : *`T

The compiler has no way of knowing whether the function on the right is returning a nil value or not. I tried a few different solutions to this. I first tried to add an argument to the `func_info` table. Called `ret_ofs` which would keep track of the offsets that a function returns. This partially fixed the problem however it did not work for statements of the type:

```
func boo() { ...  return foo()},    if e {return e1} else {return e2}
```

The first did not work if the function `foo` was allocated after the function `foo`. I thought of creating a graph of the function calls and do a topological sort on it to solve this however I soon realized the idea was doomed since there could be cycles in this graph. A possible solution that I did not have time to implement would be that at every return we store a pointer to a memory location corresponding to where the called function will save its return offsets once it returns. The use of pointers removes the cycle problem and the function writing its return types to memory during compilation should solve the if/else problem.
Finally I also have a problem with the test file *exec/struct.go* My code gives the correct output that corresponds exactly to the output I get when I run `go run /exec/struct.go` however the test still fails.

Finally I think that it is probably not the best idea to encode all of the data of our program from a single byte in memory, since if by any chance this byte gets corrupted we would loose instantly all of our data.