**Γεώργιος Λυδάκης**

**AM: 1143 (geolyd@csd.uoc.gr)**

**Αναφορά project HY-546, χειμερινό εξάμηνο 2020-2021**

**Summary of "Memlog" pass creation process**

**1.** I began by reading some introductory information as to how LLVM passes work, mainly from the official documentation. Using this, I concluded that I'd need to implement a **ModulePass**, since the problem we are asked to solve depends on module information (e.g. we need to declare function prototypes such as fopen, visible on a module scope). The basic function for such an implementation is **runOnModule.**

**2.** Then, by googling specific queries, I learned more about how programs are structured in LLVM bitcode: **Instructions** form **BasicBlocks**, which in turn form **Functions**, contained in **Modules**. Again, via StackOverflow and other message boards dedicated to LLVM, I managed to find information as to how a function call can be injected in bitcode via C++, and as to how a function prototype can be declared. This was enough to do the first modifications on the program, by declaring fopen and fclose, a step I judged must be performed first, since the user may not necessarily call these functions in their program (and so they must somehow be linked to the basic library of C).

**3.** However, in order to correctly carry out all the tasks necessary for solving the problem, such as declaring global variables and string constants, injecting functions calls in the code, detecting calls of malloc and free, as well as extracting their arguments, continuing "step 2" proved to be tiresome and ineffective. At the same time, there was no apparent "official" large-scale tutorial on using all of LLVM's components. Fortunately, at about that time I came across a post which mentioned the command "llc -march=cpp". This is a tool that receives as input a bitcode file and creates an indicative example of LLVM C++ code that could have been used to generate said bitcode. This tool proved to be invaluable for understanding how LLVM instructions can be generated in a structured way: for example, to find out how a global variable should be declared, one simply has to declare one in a .c file, compile with clang, feed the output bitcode to llc and look for the code that is relevant to the declaration of said variable.

The output code can, admittedly, be slightly verbose, and features machine-generated variable names, but if one has a basic understanding of LLVM's basic classes, the extraction of useful snippets becomes a manageable task.

**4.** With all that in mind, the final "steps" that are visible in the source code I'm turning in are:

- Declarations: declare prototypes for malloc, free, fprintf, fopen and fclose, so that they can be linked to the basic library of C. Here, an understanding of LLVM's **Type** class, as

well as a few of its subclasses is necessary. Basically, each function prototype is constructed by giving a result type as well as a type for each argument. Additionally, **FILE** struct needs to be declared (as "**struct._IO_FILE**", information taken from bitcode), so that it can also be linked to the **FILE** API.

- Global variables: it is necessary to declare a global **FILE** pointer, since every function must have access to the "memlog.txt" file. Furthermore, the strings that will be used as arguments to the fopen and fprintf functions must also be declared globally. The classes **GlobalVariable** and **IRBuilder** prove useful here. In particular, **IRBuilder** is a class that simplifies many of the operations found in an llc output file, and global string declaration is one of them. I name all variables with identifiers that begin with "$", so as to avoid conflicts with user-define
- Finding call sites of malloc and free: for this we basically iterate over all instructions in all blocks of all functions, and compare the opcode of each one with the LLVM "call" opcode. Then, we check whether or not the function being called matches the prototype that we declared in the first step.
- Injecting calls to fprintf: **IRBuilder** comes into play again, since we can use the methods **SetInsertPoint** and **CreateCall, CreateStore** etc. to generate new instructions. Basically, we have to use the functions as they were declared at the beginning, create a list of their arguments (based on the global strings defined previously, etc.), and create the calls via the **IRBuilder**. One thing to note here is that we must gain access to the arguments of the previous malloc or free, which can be done via methods of their **Instruction** subclasses.
- Injecting calls to fclose: this step is similar to the previous one, the difference being that this time we must find every "return" statement in main. This is again done by comparing opcodes. Using the **IRBuilder**, we insert a call to fclose before every return statement of main.

**Things to note**

**1.** Technically, "fclose" is not called at *every* program exit point. "Return" statements are not the only way to end a program in C. "exit" and "abort" functions (and perhaps others, of which I am not aware) can also end execution, and these may not necessarily be located directly inside main's body. However, covering every possible case was cumbersome, and at the same time I believe that the OS performs some cleanup when the program ends. Some buffered data that may not yet have been written to the disk may end up destroyed, and so perhaps this issue could be investigated further (e.g. if the pass was to be used in some memory debugging tool).

**2.** If a user defined a function named "fopen", with a different signature than the one provided by stdlib.h, the pass would erroneously call this function instead of the library one. If the two signatures did not match, the pass will fail with some sort of "Bad function call" message (at the point where my code attempts to inject a call to the function with the mismatch). If the two

signatures did match, then the pass would end up inserting a call to the user-defined function. Obviously, this is not the desired result. One way of fixing this would be to run llc -march=cpp on the source code for fopen (and others), extract the code that generates their body, and force the pass to generate a function with an illegal name, e.g. “$fopen”, where the body would be constructed from the code that we got from llc. Then, the calls the pass needs to make would refer to the “illegally named functions”, whose implementation is actually the correct one.

However, this is a tiresome task, and I believe that it’s not exactly a goal of the project, but I did think that it was worth mentioning, since it came up as a potential “failure point” of my implementation.