Student: Jérôme Beclin **PH-SFT**

Supervisor: Vasil Vasilev

Validation of Code Unloading and error recovery in ROOT's new C++ Interpreter Cling

CERN SUMMER STUDENT PROGRAMME 2013
June 2013 – September 2013

INTRODUCTION

During my summer at CERN, I have been working on the software ROOT, and in particular on the new compiler of ROOT, named Cling.
ROOT is the LHC physicists' common tool for data analysis. The main user interface is a C++ interpreter; physicists run, edit, and re-run code until it does what they want.
A crucial part of re-running is the prior unloading of the code's old version. In this project, my job was to create a tool to validate the compiler representation of the code after unloading.

MY PROJECT

I had to create a tool to make sure that the internal state of the compiler and interpreter was the same before and after introducing an error or unloading of code.

To do so, we have to check that all data structures of the compiler state are the same.
Examples of those data structures would be:

- AST (Abstract Syntax Tree)
- Included Files
- Lookup tables
- Preprocessor macro definitions
- ID Resolving chain
- And many more

During my three months at CERN, I had time to work on the AST, the included files and the lookup tables.

1) Definitions:

AST (Abstract Syntax Tree)

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is 'abstract' in not representing every detail appearing in the real syntax.
Abstract syntax trees are a data structure widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

Lookup tables

These tables are the place where all the identifiers are stored.

Included files

When including a file to your code (using #include <header name>), that file will be treated as if its contents were inserted into the original file.
The include directive instructs the preprocessor to paste the text of the given file into the current file. Generally, it is necessary to tell the preprocessor where to look for header files if they are not placed in the current directory or a standard system directory.

2) My implementation

I extended the existing functionalities in cling in order to validate the internal representations. I implemented "dot" commands, so I worked on the Interpreter, MetaSema and MetaParser files.

The two commands I implemented are called .storeState and .compareState.
These commands dump the internal compiler data structures (AST, included files and lookup tables).
They are usable in Cling.

1).storeState "stateName"

That command store the current state of the internal data structures into files.
It stores in three different files the AST, the included files and the lookup tables.

2) .compareState "stateName"

That command compare the current state of the data structures with the previous one stored, then make a diff between both states in order to create a file containing the differences between both states (if there are some).

3) Details

When using .storeState "stateName"

• Creation of three files: - stateNameAST.tmp (containing the current AST)
- stateName.lookup (containing the current lookup tables)
- stateName.includedFiles (containing the current included files)

Those files are created in the current directory.

• If stateName is already used, an error message is printed and Cling stops

When using .compareState "stateName"

• Creation of three files: - stateNameAST**cmp**.tmp (containing the current AST)
- stateName**cmp**.lookup (containing the current lookup tables)
- stateName**cmp**.includedFiles (containing the current included files)

• A diff is made between: - “stateNameAST.tmp” and “stateNamecmpAST.tmp”
- “stateName.lookup” and “stateNamecmp.lookup”
- “stateName.includedFiles” and “stateNamecmp.includedFiles”

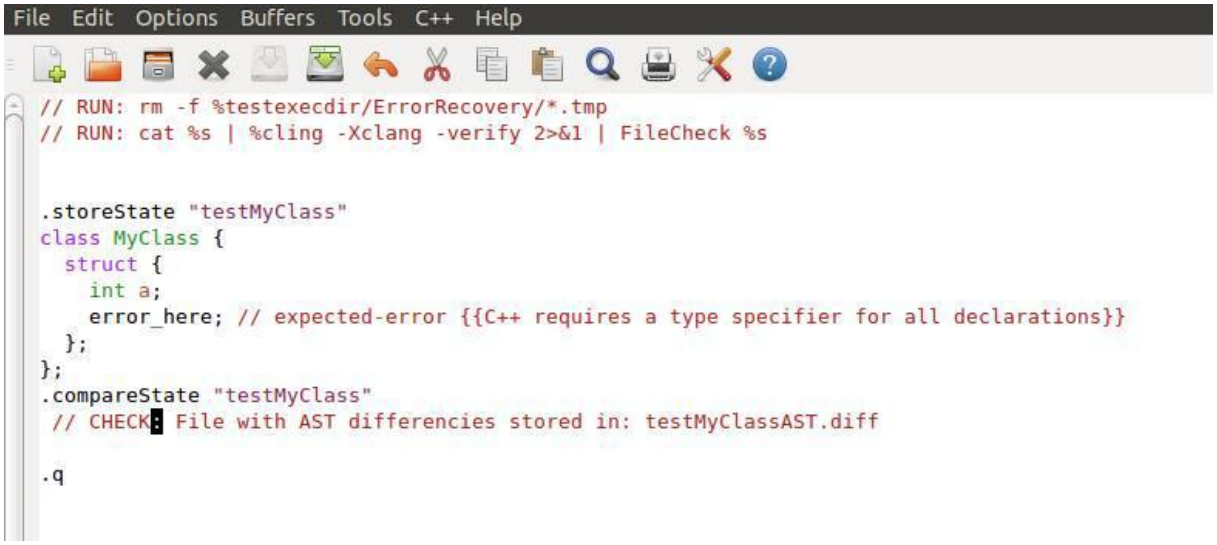
• If there are differencies in the between those files, a new file is created:
- stateNameAST.diff if there are differences in the AST
- stateNameLookup.diff if there are differences in the lookup tables
- stateNameIncludedFiles.diff if there are differences in the included files

We then print on the screen a message of that kind: “File with AST differences stored in: stateName.diff”, followed by the path to the file.

• All the files previously created for the diff are then deleted

4) Skeleton of a test

After implementing, I had to test that everything worked well.
To do so, I used a test like the one below (this test was used to make sure that the differences in the AST were printed correctly).



.storeState is introduced before the error.
.compareState is introduced after the error.

We then do a check to verify that we have the message expected printed on the screen.

5) What is yet to implement

With a bit more time, a few implementations would have been interesting.
First of all, it would have been nice to be able to choose a concrete path as a name for both meta commands, instead of storing the created files in the current directory.
Second of all, the verification of the lookup table is not fully ready yet, because a new source revision of llvm needs to be imported in ROOT mainline.

Conclusion

My project at CERN was a very interesting one.
I had the chance to help creating a new compiler, a totally new experience for me. I also learned to use C++, a language that I didn't know very well before. I also discovered GIT, a new source code management system for me. It was professionally very stimulating, and I gained a lot of experience here.
In the mean time, it was also an unforgettable social experience, because I had the chance to work in a very multi-cultural team, and even more, in a very-multicultural environment.
My summer at CERN is definitely a summer that I'll remember.