Plugins for Clang’s Compiler

Miri Kogan

The repository includes:

* Dependencies Mapping Plugin
  + source file of the plugin (.cpp)
  + python script (see explanation below)
  + test files:
    - .cpp and .h files that comprise a small project (“Capture The Flag”) with files that are mutually dependent
    - Expected output (as generated by the python script)
    - Input file for python script, based on the files above
* Type Inconsistency Plugin
  + source file of the plugin (.cpp)
  + test files:
    - .c file with type inconsistencies related to enums
    - .cpp file with different cases of type inconsistencies (not enums, as these cases are already flagged as error by the Clang++ compiler)
    - Expected output in the form of screenshots of the command line window from where the plugin was invoked, displaying the emitted warnings
* Traces Plugin
  + source file of the plugin (.cpp)
  + python script (see explanation below)
  + test files (.c files and .cpp files are separated to 2 different folders):
    - .cpp and .c files that include various points of potential branching
    - Expected output in the form of the re-written versions of the source files above
    - Input file for python script, based on the files above

Read all about it

The main conclusions and insights I’ve had while working on this project are documented in the [blog](https://mirikog.wixsite.com/clangplugin)

Using the plugins:

# (1) Dependencies Mapping Plugin

This plugin creates a dependencies graph for a file, based on its #include directives.

The output of the plugin is a “<file-name>.dot” file, which is in a format that is required by Graphviz. Graphviz is the program that is used to draw the graphs that are generated by the plugin - it can be downloaded from [here](https://www.graphviz.org/download/).

Like the preprocessor, that handles the #include directives, the mapping of dependencies can go on recursively, until a file that has no #include directives is reached. By default, however, the plugin will stop at system headers and will not cover the full depth of the #include-s chain. This means that the graph will focus on the files created by the user. If, for some reason, the entire dependencies chain is required, the plugin can accept an argument that will make it generate the full dependencies map. This argument is the word “angle”, signifying that headers with angled parenthesis (i.e. system headers) should also be handled by the plugin.

The plugin also has a mandatory argument that specifies the name (full path) of the output file. As mentioned above, the name of the output file should have a .dot extension, as this is the files that Graphviz accepts. The file name argument is passed by proceeding it with the flag “-out-file”.

To run the plugin from the command line, you can either directly invoke Clang’s frontend, or you can call Clang’s driver, which will pass the relevant arguments to Clang’s frontend. The main difference between the two options is in the generation of a binary file: calling Clang’s driver will run the plugin and also create a binary file (like a normal compilation), while calling Clang’s frontend will only run the plugin (this difference is also covered in the [blog post](https://mirikog.wixsite.com/clangplugin/post/running-a-plugin-from-the-command-line)).

### Example usage:

1. Running the plugin from the command line, using Clang’s **driver** (generating a binary file):

/path/to/clang/bin/clang++ \

/path/to/compiled/file.cpp \

-o /path/to/executable \

-Xclang -load -Xclang /path/to/compiled/plugin/DependenciesMappingPlugin.dylib \

-Xclang -add-plugin -Xclang DependenciesMappingPlugin \

-Xclang -plugin-arg-DependenciesMappingPlugin -Xclang -out-file \

-Xclang -plugin-arg-DependenciesMappingPlugin -Xclang /path/to/dot/out/file.dot \

-stdlib=libstdc++ -std=c++11

1. Running the plugin from the command line, using Clang’s **frontend** (running only the plugin, without generating a binary file):

/path/to/clang/bin/clang++ \

/path/to/compiled/file.cpp \

-o /path/to/executable \

-Xclang -load -Xclang /path/to/compiled/plugin/DependenciesMappingPlugin.dylib \

-Xclang -add-plugin -Xclang DependenciesMappingPlugin \

-Xclang -plugin-arg-DependenciesMappingPlugin -Xclang -out-file \

-Xclang -plugin-arg-DependenciesMappingPlugin -Xclang /path/to/out/dot/file.dot \

-stdlib=libstdc++ -std=c++11

As mentioned above, in order to get an image file with the dependencies graph, you should pass the plugin’s output (the .dot file) to Graphviz. This can be done by running the following command from the command line:

dot -T png -o/path/to/out/image/file.png /path/to/dot/file.dot

The above works fine when using the plugin on single file. However, the purpose of the plugin is to give an insight regarding the project as a whole, comprised of multiple files. To achieve this, there is a python script that accepts a .txt file with the list of files in a project (full paths) and generates a graph (.png file) for the dependencies of the entire project. Script usage:

**mandatory arguments** **-**

* file listing all files that are to be included in the dependencies graph
* -clang-dir <path to clang’s binary>
* -plugin-dir <path to the plugin dynamic library>

**optional argument** **-**

* -gen-subgraphs  
  if specified, the script will also generate an image file with a graph for each file in the list, besides the graph that is generated for the project

The files will be generated in a newly created directory, in the directory from where the script will run.

Please note that in order to successfully run the script, you need to have Graphviz installed on your machine (the plugin itself does not require Graphviz in order to run and its output can be read by a simple text editor. Nonetheless, this output will get its full meaning only when processed by Graphviz).

### Script usage example:

/path/to/python/script/DependenciesMappingPlugin.py \

/path/to/script/input/file.txt \

-clang-dir /path/to/clang/directory/bin \

-plugin-dir /path/to/plugin/directory \

-gen-subgraphs

\* Note: the example project is not actually functional, so don’t try to run it

# (2) Type Inconsistency Plugin

This plugin extends the compiler’s warning messages, to point the user’s attention to places in the code that deal with inconsistent types. The inconsistencies that the plugin will flag are not considered as errors and will not interfere the compiler’s functionality. However, they do have the potential of causing unexpected behaviour in runtime.

The plugin is designed to warn for inconsistencies between the following types:

1. int and bool
2. int and enum (in .c files)
3. types defined by typedef and original types
4. types defined by the ‘using’ keyword and original types

Also, the cases where the plugin will look for inconsistencies are:

1. assignments: type lhs vs. type rhs
2. return statements: type of return value vs. type in function’s signature
3. function arguments: type of passed argument vs. type of parameter in function’s signature
4. binary operations: type lhs vs. type rhs

As with the plugin described above, this plugin can be called from the command line in two formats: with Clang’s frontend and with Clang’s drives (but, as opposed to the Dependencies Mapping plugin, it makes sense to run this plugin frequently, as part of a normal compilation - meaning that the driver option is more relevant here than in the other plugin).

**Note**: I’ve added support to the case that was tested in the he projects’ presentation event and didn’t output the desired result. The example that we used in the event is in the .cpp test file that is provided for this plugin (“typeInconsistency\_test.cpp”). It is marked with the “ctor example” comments. The code that was added to handle this kind of cases is marked in the plugin’s code by the comment “constructor expression”.

### Example usage:

1. Running the plugin from the command line, on a .**cpp** file, using Clang’s **driver** (generating a binary file):

/path/to/clang/bin/clang++ \

/path/to/compiled/file.cpp \

-o /path/to/executable \

-Xclang -load -Xclang /path/to/compiled/plugin/TypeInconsistencyPlugin.dylib \

-Xclang -add-plugin -Xclang TypeInconsistencyPlugin \

-std=c++11 -stdlib=libstdc++

1. Running the plugin from the command line, on a .**cpp** file, using Clang’s **frontend** (running only the plugin, without generating a binary file):

/path/to/clang/bin/clang++ \

-cc1 -load /path/to/compiled/plugin/TypeInconsistencyPlugin.dylib \

-plugin TypeInconsistencyPlugin \

/path/to/compiled/file.cpp \

-std=c++11 -stdlib=libstdc++

1. Running the plugin from the command line, on a .**c** file, using Clang’s **driver** (generating a binary file):

/path/to/clang/bin/clang \

/path/to/compiled/file.c \

-o /path/to/executable \

-Xclang -load -Xclang /path/to/compiled/plugin/TypeInconsistencyPlugin.dylib \

-Xclang -add-plugin -Xclang TypeInconsistencyPlugin

1. Running the plugin from the command line, on a .**c** file, using Clang’s **frontend** (running only the plugin, without generating a binary file):

/path/to/clang/bin/clang \

/path/to/compiled/file.c \

-cc1 -load /path/to/compiled/plugin/TypeInconsistencyPlugin.dylib \

-plugin TypeInconsistencyPlugin

# (3) Traces Plugin

This plugin inserts trace prints in all points in the code that represent a fork, i.e. a split of the flow to different branches. The plugin doesn’t change the input file, but rather it generates a new file that has the content of the input file with the trace prints interwoven in it. So, in order to see the prints in action, it is needed to compile the file created by the plugin and run it. Of course, it is also possible to open and examine the newly generated file and find the added print statements in the relevant places.

The plugin can accept an argument specifying a name for the output file. If this argument is not passed, the plugin will use the name of the input file and add “\_traces” to it.

Implementation remarks:

* To give the user flexibility in using the files generated by the plugin (if desired), all print statements that are added are wrapped by an #ifdef \_\_TRACES\_\_ directive, so that prints can be turned on/off in different compilations.
* Entering a loop (for/while) will result in a single trace print and will not emit a print for each iteration of the loop. The motivation behind this decision was to avoid clutter in the prints (this decision can be easily undone).

### Example usages:

Please note that there’s no option of running the plugin with Clang’s driver - only frontend options are available.

1. Running the plugin from the command line, on a .**cpp** file, using Clang’s **frontend** (running only the plugin, without generating a binary file):

/path/to/clang/bin/clang++ \

-cc1 -load /path/to/compiled/plugin/TracesPlugin.dylib \

-plugin TracesPlugin \

-plugin-arg-TracesPlugin -rewrittern-out \

-plugin-arg-TracesPlugin /path/to/rewritten/output/file.cpp \

/path/to/compiled/file.cpp \

-stdlib=libstdc++ -std=c++11

1. Running the plugin from the command line, on a .**c** file, using Clang’s **frontend** (running only the plugin, without generating a binary file):

/path/to/clang/bin/clang \

-cc1 -load /path/to/compiled/plugin/TracesPlugin.dylib \

-plugin TracesPlugin \

-plugin-arg-TracesPlugin -rewrittern-out \

-plugin-arg-TracesPlugin /path/to/rewritten/output/file.c \

/path/to/compiled/file.c

Since adding the print statements is usually not an objective on its own, but a mean to keep track of the program’s flow in runtime, there’s a python script that handles the insertion of the print statements, followed by a compilation that takes in the new file/s and generates an executable ready to be run. The script accepts a .txt file with a list of files that are to be compiled (= linked) together. The files that need to have trace prints incorporated into them are marked by a ’T’ at the end of their line in the file holding the list (’T’ stands for ‘Traces’). Files not marked with a ’T’ will be compiled as-is. Script usage:

**mandatory arguments** **-**

* file listing all files that are to be compiled to an executable
* -clang-dir <path to clang’s binary>
* -plugin-dir <path to the plugin dynamic library>

**optional argument** **-**

* -keep-rewritten
  + if specified, the script will keep the files that the plugin generated (otherwise, they will be removed - this is the default option)
  + These files can be useful if the user wants to manually extend the information that is being printed
  + If the source file from which the new file (with trace prints) was generated *hasn’t changed*, the new file can be reused in future compilations performed via the script and save the compiler some work, by not re-generating it, time and time again. To do this, replace the original source file in the list of files (that had ’T’ at the end of its line) with the new file (without a ’T’ at the end!)
* all optional arguments that should be used by Clang for compiling the input files to an executable, not related to the functionality of the plugin (example: -o <name of executable>)

Note that for every file that is marked with a ‘T’, the script will generate the file with trace prints in the same directory of the input file.

### Script usage example:

/path/to/python/script/TracesPlugin.py \

/path/to/script/input/file.txt \

-clang-dir /path/to/clang/directory \

-plugin-dir /path/to/plugin/directory \

-keep-rewritten -o /path/to/executable