Clang LLVM frontend as a modern C++ source-code generation tool

Research and Development Project

Name	Student Number
Morten Haahr Kristensen	201807664
Mikkel Kirkegaard	201808851

Supervisor	Email
Lukas Esterle	lukas.esterle@ece.au.dk

Department of Electrical and Computer Engineering
Aarhus University

March 23, 2023

Abstract

Contents

1 Introduction

2 Background

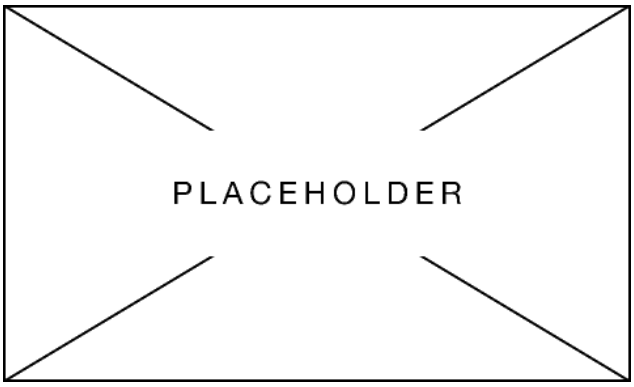

Figure 2.1: LLVM and Clang overview

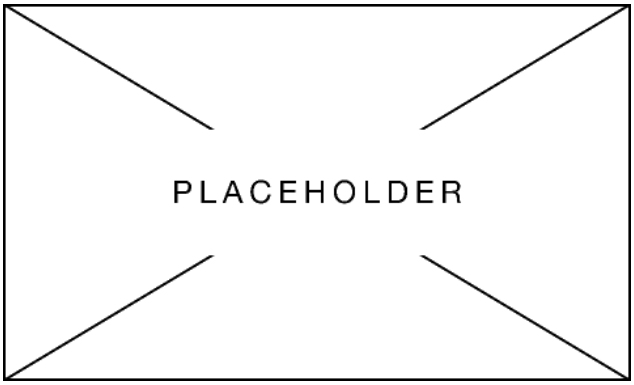

Figure 2.2: Clang and LibTooling overview

There are two overall methods of writing standalone tools for the Clang frontend. They can either be made with the visitor pattern using the RecursiveASTVisitor, or they can be made with LibTooling which abstarcts away from the RecursiveASTVisitor but allows for the same functionality. This R&D project will use LibTooling for the development of Clang tools.[1, 2]

3 Project description

4 Methods

5 Requirements

5.1 Functional requirements

5.2 Non-functional requirements

6 Architecture

7 Development

7.1 Installing LLVM and Clang

One might think installing LLVM and Clang should be a straightforward process but in reality, it can be quite complex. Therefore, this section means to describe the process that was used during the R&D project. The section is heavily inspired by [3] but more specialized to account for the concrete project.

The process of compiling LLVM, Clang and LibTooling can be considered a two-step process. Initially, the tools must be compiled using an arbitrary C++ compiler and then recompiled using the Clang compiler itself.

Before compiling the projects, one must install the needed tools, which include "CMake" and "Ninja". They can be installed using a package manager or by compiling it locally through https://github.com/martine/ninja.git and git://cmake.org/stage/cmake.git. Furthermore, one needs to have a working C++ compiler installed. The rest of this section assumes the compiler GCC is installed.

LLVM and Clang can then be compiled for the first time using GCC. Assuming the terminal is used, this can be done as seen in listing 7.1. First, the LLVM repository is cloned which also contains the Clang project. Line 2 - 4 goes inside the repository and sets up the build folder. Line 5 uses CMake to configure the project where Ninja is used as the generator[1], Clang and Clang Tools are enabled, tests are enabled and it should be built in release mode. The final line instructs Ninja to build the projects. Note that it is possible to skip building and running the tests but it is recommended to ensure that the build process succeeded.

```
1 git clone https://github.com/llvm/llvm-project.git
2 cd llvm-project
3 mkdir build
4 cd build
5 cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang" -DLLVM_BUILD_TESTS=ON -DCMAKE_BUILD_TYPE=Release
6 ninja
```

Listing 7.1: Bash commands to initially compile LLVM and Clang.

The next steps consist of testing the targets to ensure that the compilation was successfull. This is done by running the tests as seen in the two first lines on listing 7.2. Finally, the initial version of Clang that is compiled with an arbitrary compiler is installed.

[1] A CMake generator writes input files to the underlying build system.[4]

```
ninja check
ninja clang-test
sudo ninja install
```

Listing 7.2: Bash commands to test the LLVM and Clang projects and then finally install them.

Clang should now be recompiled using Clang to avoid name mangling issues [5], i.e., ensure that the symbolic names the linker assigns to library functions do not overlap. This time the project "cmake-tools-extra" should also be included to build LibTooling and the complementary example projects. The option `-DCMAKE_BUILD_TYPE=RelWithDebInfo` is also a possibility if one wishes to include debug symbols in the libraries which can be useful during development. This however comes with a performance trade-off, as Clang itself will also be compiled with debug symbols which will slow it down. The group has yet to find a way to compile LibTooling with debug symbols but Clang without it.

```
cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -DCMAKE_BUILD_TYPE=Release
↪  -DCMAKE_CXX_COMPILER=clang++
ninja
```

Listing 7.3: Bash commands to compile LLVM, LibTooling and Clang with Clang as compiler.

Finally, the steps from listing 7.2 should be repeated to verify the recompilation and install the tools.

7.2 Build environment

The documentation for writing applications using LibTooling such as [6, 7] mainly concerns writing tools as part of the LLVM project repository. While this is good for contributing to the project, it is not ideal for version control and developing stand-alone projects. It was necessary to create a build environment that allowed for out-of-tree builds which utilize LibTooling. A similar attempt was made in [8] but the project was abandoned in 2020 and LLVM has since moved from a distributed repository architecture to a monolithic one making most of [8] obsolete. The following section is dedicated to describing the important decisions made related to the build environment.

7.2.1 Build settings

Initially, some general settings for the project are configured which can be seen in listing 7.4. Line 1 forces Clang as the compiler which is highly recommended as LibTooling was compiled with Clang. Choosing another compiler may result in parts of the project being compiled with another standard library implementation, e.g., libstd++ that is the default for GCC. This may cause incompatibility between the application binary interfaces (ABIs) which is considered undefined behaviour, essentially leaving the entire program behaviour unspecified [9]. This concept is also known as ABI breakage. Line 2 defines the C++ standard version, which is set to C++17 since LibTooling was compiled with this. Line 3 defines the output directory of the executable to be in `<build_folder>/bin` which has importance concerning how LibTooling searches for include directories at run-time as described in section 7.2.2. Finally, line 4 disables Run-Time Type Information (RTTI). RTTI allows the program to identify the type of an object at runtime by enabling methods

such as `dynamic_cast` and `typeid` among others. When compiling LLVM it is up to the user whether RTTI should be included or not. RTTI is disabled by default when compiling LLVM as it slows down the resulting executable considerably. This flag is propogated to the subprojects that were compiled with LLVM such as LibTooling. By default a project in CMake is compiled with RTTI and CMake will asume that the used libraries were compiled with the same flags. This will result in nasty linker erros and the RTTI should therefore be explicitly disabled in the tool project.

```
set(CMAKE_CXX_COMPILER clang++)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")
add_compile_options(-fno-rtti)
```

Listing 7.4: General settings for the CMake build environment.

7.2.2 Run-time include directories

When executing binaries created with LibTooling, a big part of the process is the analysis of the target source code. The analysis is done following the pipeline as shown in fig. 2.1. Most projects written in C++ make use of the C++ standard library that implements many commonly used functionalities in C++. Naturally, the tool needs to know the definitions for the standard library in order to analyse the target source code. For practical reasons, LibTooling provides a mechanism for the automatic discovery of header files that should be included when parsing source files. It finds the headers by using a relative path with the pattern `../lib/clang/<std_version>/include` from the location of the binary. Where `<std_version>` indicates which version of the standard library which the tool was compiled with (in this project it was 17).

This hard-coded approach is quite simple but limited, as it forces the users to only run the tool in a directory where the headers can be found in the relative directory `<current_dir>/../lib/clang/17/include`. If the user attempts to run it somewhere else, and the analyzed files make use of standard library features, they will get an error while parsing the files (e.g. that the header `<stddef.h>` was not found). This issue makes it more difficult to write truly independent tools as they still need some reference to the Clang headers, which would essentially mean moving the executable to the directory where Clang was compiled.

One existing solution is to provide the location of the headers as an argument to the binary when executed. This is possible since tools written with LibTooling invoke the parser of Clang, from where it is possible to forward the include directory as an argument to the compiler e.g. by specifying `-- -I"/usr/local/lib/clang/17"`. However, this was found to be impractical since the location of the include path may vary depending on the system and forgetting to write the path results in errors that can be very difficult to decipher.

Instead, it was decided to create a build environment where the user must provide the location of the Clang headers when configuring CMake or an appropriate error message is generated. Through CMake, the necessary headers are then copied to the build directory.

The solution is by no means perfect, as the user is still forced to execute the binary from the build directory. In many situations, this is sufficient, as most IDEs follow this behaviour as default and it allows the projects to be built out-of-tree. If the user wishes to run the binary from outside the build directory, they still have the option of specifying the location through the `-- -I"<clang_include>"` option. The solution can be found in the functions.cmake file.

In the future, it may be desirable to explore a solution using the LLVM command line library to search some

commonly used directories for the Clang headers.

7.2.3 Configuring the target project

The target project on which source files the tool will be run should generate a compilation database. A compilation database contains information about the compilation commands invoked to build each source file. This file is used by LibTooling to detect the compile commands and include files necessary to generate the correct AST for the compilation unit.[10] Tools like CMake can auto-generate the compilation commands at configuration time. When using CMake the compilation database generation can be enabled by inserting `set(CMAKE_EXPORT_COMPILE_COMMANDS 1)` in the CMakeLists.txt file.

7.3 Tool structure

Tools created through the LibTooling framework all have a similar structure which can be seen below.

1. Command line parsing
2. AST node matching
3. Node data processing
4. Handling the results

The **Command line parsing** section of the tool structure is responsible for the parsing of the command line options which the tool was invoked with and setting the configuration data that specifies the behaviour of the tool. A detailed description of this section of the tool structure can be found in section 7.3.1.

The **AST node matching** section of the tool structure is where the AST node matchers are defined. AST node matchers traverse the AST of the source code passed to the tool through the command line and binds relevant nodes to identifiers which can be used in the Node data processing section. A detailed description of the AST matching step see section 7.3.2.

The **Node data processing** section of the tool structure is where information is extracted from the nodes that were bound during the AST node matching step. This is where the primary functionality of the tool is implemented. There are multiple interfaces which can be to extract information from bound nodes and they will be described in section 7.3.3.

The **Handling the results** part of the tool structure is where the matching and processing, defined in the previous sections, is applied. This is also where all the results are handled and presented to the invoker of the tool. This part of the tool structure is described further in section 7.3.4.

7.3.1 Command line parsing

TODO: Me :D

7.3.2 Matching the AST

The first step in working with LibTooling is to find the AST that is releavant for the tool. This is a very important part of the tool development as the matching defines what kind of information is avaliable for the later stages of the tool. If the wrong AST nodes are matched, the later processing becomes unessesarily complex and complicated.

The easiest way to do that is by creating or finding a very simple example of the code that should be changed. The entire AST for that file can be dumped through clang by invoking: `clang -cc1 -ast-dump input_file`. This command will print all the AST information in the input file to the console, which can then be analysed manually for the wanted nodes/patterns.

After an AST node has been identified as interesting for the tool it should be matched. The way to create a matcher in LibTooling is by using the predefined matchers in the LibTooling framework.[11] Custom matchers can also be constructed and they are a very powerfull tool when the standard matchers are lacking. The builtin matchers are however enough most of the time and they should therefore be the default place to look when creating a matcher.

The first step when creating a matcher is to get the wanted node through a Node matcher. Theese are the matchers witch allows one to match some kind of node type and bind it to an identifier. Binding a node allows it to be used in the later stages of tool processing. The node matchers will often match quite a bit of nodes. The amout of matches can be controlled through the use of Narrowing matchers which allows the tool authors to filter the matches E.G. only match variable declarations with a constant array type.

The creation of the matcher is therefore an iterative process where nodes are matched and the matches are narrowed untill the tool is left with the nodes that contain the needed information. The matching iterations should stop when all the wanted nodes are matched.

7.3.3 Bound node processing

When all the nodes have been found, the information stored inside of them must be extracted. The way to do that, is by using an object that inherits from `MatchFinder::MatchCallback`. The `MatchFinder::MatchCallback` class defines the `run` method which must be overriden in it's children. The run method has a `const MatchFinder::MatchResult &R` parameter which contains all the bound nodes. The information inside the nodes can then be extracted and used to generate diagnostic messages or other information releavant for the tool.

The `MatchFinder::MatchCallback` is the raw interface which allows for node processing but it lacks many convenience methods and requires the user to save the extracted information explicitly. The Clang developers have created an abstraction over the `MatchFinder::MatchCallback` with convenience methods and an automatic way of extracting information. This interface also allows for easy conversion to source code changes. The abstraction is called `Transformer` and is what will be used in this R&D project.

Transformers combine a rewriting rule with a result consumer. Rewriting rules combine a matcher with a AST edit and potential metadata. The AST edit is a change in the source code comprised of a range to rewrite and a concatination of multiple Stencils which generate the new source text. Stencils extract information from bound nodes and convert the information to strings. A more detailed description of Stencils can be seen in section 7.3.3.

The result consumer is responsible for the saving of the relevant results so they can be processed by the `ClangTool` later. The result consumer is further described in section 7.3.3.

Stencils

The stencil library is used to extract information from bound nodes and convert the information to strings. The stencil library is an abstraction on top of `MatchComputation<std::string>` which is called on matched nodes through the Transformer API.

Examples of usecases for the stencil library could be to extract the element type of an array or to issue a warning at a given location. The methods in the stencil library are primarily focused on control flow, concatination of stencils and expression handling. Therefore it is very likely that the creator of a tool will have to create custom stencils to extract the nessecary data from the bound nodes.

Luckily the stencil library allows the simple conversion from `MatchConsumer<std::string>` to a `Stencil` through the `run` method. The `MatchConsumer<T>` type is a typename for `std::function<Expected<T>(const ast_matchers::MatchFinder::Mat` with `std::string` as the template parameter. This API allows the creator of a tool to write small methods that extract the nessecary information from a bound node as a string and seamlessly concatinate them together through the `cat` method from the stencil library.

All the stencils in the stencil library return strings but it is possible to create an identical library that returns any type of data if that is required for the tool. The reason the stencil library works solely with strings is because it is primarily used to generate source code changes which must be converted to strings in some way or form in order to be written to disk.

Along with the stencil library is a range selector library. The range selector library also uses the `MatchConsumer<T>` API but with `RangeSelector` as an output instead of strings. This library contains convenience methods to select ranges of bound nodes. It is also possible to select locations right before or right after a bound node. This feature allows the tool implementor to add source code exactly where they want it.

Combining matchers and stencils

In order to create a tool it is nessecary to have both matchers and stencils and the tool authors must therefore consider how they should be combined.

There are two approaches when combining the matcher and the stencils. The fist approach is to have the matcher bind only the top node and then extract all the information from the node through stencils. The other mehtod is to bind multiple nodes inside the matcher to get access to the important information. Both approaches can be used for the same purpose but the implmentations look vastly different.

If the implementor chooses the first combination method, then the stencils can become rather complicated because they have to filter and extract information from the node.

If the second method is chosen instead, then the complexity of extracting information from the nodes is placed inside the mathcer. This can lead to complicated matchers with many bound subnodes. The stencils that extract information from the bound nodes are however much simpler. When using this mehtod it can be hard to tell when all the nessecary information has been bound and the developers can be stuck looking for a matcher that binds to exactly the information needed when it could easily be extracted from a higher node.

It is always possible to refactor later, so the developers of the tool should consider the two approaches carefully and choose their battles wiesly.

Consuming the transformation changes

When the bound nodes have been processed through the Transformer API, the transformation changes have to be consumed. This is done through a `Consumer` which is a type alias of a functor which takes `Expected<TransformerResult<T>>` as a parameter. The `TranformerResult<T>` type contains any source code changes that was generated by the rule and the provided metadata with type T.

The consumer can make desicions based on the received edits and metadata, but the most common use case for the consumer is to have it store the metadata and changes to extrnal variables so it can be later used for further processing.

7.3.4 Handling the results

When the rules and transformers have been specified it is time to run them on the source code. This is done through a `ClangTool`. `ClangTool` is the API that runs the match finders over all the specified source code. They are common across all clang tools. The `ClangTool` class has a `run` method which takes a FrontendAction and runs it on the specified source code.

The `ClangTool` class can be extended to handle the results from the Transformer in different ways. A tool witch runs a Transformer could, for example, save the source code changes to disk or precent the changes to the caller of the tool and have them choose if the changes should be made. The extended `ClangTool` is also often contains the variables which will be updated in the node processing step.

8 Testing

9 Related work

10 Conclusion

10.1 Future work

References

[1] *How to Write RecursiveASTVisitor Based ASTFrontendActions. — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/RAVFrontendAction.html (visited on 03/17/2023).

[2] *LibTooling — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/LibTooling.html (visited on 02/23/2023).

[3] *Tutorial for Building Tools Using LibTooling and LibASTMatchers — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/LibASTMatchersTutorial.html (visited on 02/02/2023).

[4] *Cmake-Generators(7) — CMake 3.26.0 Documentation.* URL: https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html#cmake-generators (visited on 03/16/2023).

[5] IBM. *IBM Documentation.* Apr. 14, 2021. URL: https://ibm.com/docs/en/i/7.3?topic=linkage-name-mangling-c-only (visited on 02/28/2023).

[6] *Matching the Clang AST — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/LibASTMatchers.html (visited on 02/08/2023).

[7] *Clang Transformer Tutorial — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/ClangTransformerTutorial.html (visited on 02/08/2023).

[8] Firat Kasmis. *Clang Out-of-Tree Build.* Jan. 28, 2023. URL: https://github.com/firolino/clang-tool (visited on 02/28/2023).

[9] cppreference. *Undefined Behavior - Cppreference.Com.* URL: https://en.cppreference.com/w/cpp/language/ub (visited on 02/28/2023).

[10] *JSON Compilation Database Format Specification — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/JSONCompilationDatabase.html (visited on 02/16/2023).

[11] *AST Matcher Reference.* URL: https://clang.llvm.org/docs/LibASTMatchersReference.html (visited on 02/07/2023).