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

February 28, 2023

Abstract

Contents

1 Introduction

2 Background

3 Project description

4 Methods

5 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 [1] but more specialized to account for the concrete project.

The process of compiling LLVM, Clang and LibTooling can be considered a two-step process where they initially are compiled using an arbitrary C++ compiler and then compiled using Clang 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 went successfully. 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]What is a generator

```
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 compiled using Clang to avoid name mangling issues [2], 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. This however comes with a performance trade-off, as Clang itself will also be compiled with debug symbols, but can be useful during development. 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 new compilation and install the tools.

7.2 Build environment

The documentation for writing applications using LibTooling such as [3, 4] 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 that utilize LibTooling. A similar attempt was made in [5] but the project has been abandoned since 2020 and LLVM has since moved from a distributed repository architecture to a monolithic repository architecture, so most of [5] was obsolete. The following section is dedicated to describing the important decisions made related to the build environment.

Initially, some general settings for the project are configured which can be seen in listing 7.4. Line 1 forces Clang as the compiler

```
1 set(CMAKE_CXX_COMPILER clang++)
2 set(CMAKE_EXPORT_COMPILE_COMMANDS 1)
3 set(CMAKE_CXX_STANDARD 17)
4 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")
5 add_compile_options(-fno-rtti)
```

Listing 7.4: General settings for the CMake build environment.

7.3 Transformer

Clang provides an interface into its C++ AST called LibTooling. LibTooling is aimed at developers who want to build standalone tools and services that run clang tools.[6]

Tools that use LibTooling run what is called 'FrontendActions' over the specified code. It is through these frontend actions the tool can interact with the source code. The LibTooling tools work by parsing the command line options the tool is invoked with through the llvm command line parser. These options can be customised to allow the tools to work in a user-defined manner. After the command options are parsed, the tool needs an ASTMatcher. The ASTMatcher is the DSL formula for traversing the Clang C++ AST. There are many different matchers which allow for extensive and custom matching of the AST.[7, 3] If the matcher finds a valid C++ AST a call to a user-defined *Consumer* callback is made with the matched AST. This allows the user to further process the AST and to perform the task the tool is meant to solve. It is also possible to write custom matchers if the built-in ones are not enough to solve the task.

LibTooling is used in many different types of tools such as static analysis tools, code refactoring, language standard migration tools and much more.[8]

Because the Clang AST is so comprehensive, the Clang development team has provided a construct called a `Transformer`. A Transformer is a way to couple a *rule* together with a consumer callback. A rule is a combination of an AST matcher, a *change* and some metadata. A rule could be specified as follows: "The name 'MkX' is not allowed in our code base, so find all functions with the name 'MkX' and change it to 'MakeX"'.This could be translated into LibTooling rule:

```
auto RenameFunctionWithInvalidName = makeRule(
    functionDecl(hasName("MkX")).bind("fun"),
    changeTo(clang::transformer::name("fun"), cat("MakeX")),
    cat("The name ``MkX`` is not allowed for functions; the function has been renamed")
);
```

Listing 7.5: Example of a LibTooling Rule that renames a method 'MkX' to 'MakeX' and provides a reason for the renaming.

Where `functionDecl(hasName("MkX")).bind("fun")` is the ASTMatcher that matches all function declarations with the name "MkX" and binds it to the name "fun". The line `changeTo(clang::transformer::name("fun"), cat("MakeX"))` is specifying the change to be made, which is to change the name of match bound to "fun" to "MakeX". The last line in listing 7.5 is the metadata that is associated with this rule. When the rule matches the wanted AST expression it creates a `AtomicChange` according to the specified change.

This is then where the `Transformer` comes in. The transformer takes the resulting `AtomicChange` and the provided metadata and calls a *Consumer* callback with the findings. In this callback, the developers of the tool are then able to provide diagnostic messages and make source code changes if that is what the tool is providing.[4]

In the official documentation from Clang transformer shown in listing 7.5 is shown and explained. However, there is no official documentation on how to invoke the transformer and make it write the changes into the existing source code.

In the public repository for this R&D project, there is an example folder. This example folder contains multiple examples which show how to create Clang tools that write the results from a `Transformer` onto disk.[9] In the examples the `RefactoringTool` helper class is used to facilitate the source-code changes. This helper class is not documented in Clang's official documentation, but it contains helper methods that make source code changes easier for the developers.

8 Testing

9 Related work

10 Conclusion

References

[1] *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).

[2] 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).

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

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

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

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

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

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

[9] Morten Haahr Kristensen. *Mortenhaahr/RD.* Feb. 23, 2023. URL: https://github.com/mortenhaahr/RD (visited on 02/23/2023).