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

April 20, 2023

Abstract

Contents

1 Introduction

1.1 Preface

1.2 List of abbreviations

Abbreviation	**Meaning**
AST	Abstract syntax tree. A structure providing an abstract tree representation of syntax.
R&D	Research and development.

Table 1.1: List of abbreviations for the report.

2 Project description

This chapter provides an overall description of the project including a brief overview of the main technologies used.

The overall topic of the R&D project is source code generation, which derives from automatic programming. Automatic programming can be defined as the automation of some part of the programming process [1]. The process consists of parsing a specification as an input to the automatic programming system which outputs a program [2].
An example of an automatic programming system could be a compiler, where the specification is a program written in the desired programming language. Another example could be the tools provided by Visual Paradigm that translates the contents of UML diagrams to source code of different programming languages [3].
The motivation behind automatic programming is that it allows the developer to express themselves more abstractly through specifications, allowing for smaller, more understandable and less error-prone programs [2].

Source-code generation can be considered a specific area of automatic programming where the output of the automatic programming system is source code. In contrast, the output of an automatic programming tool can also be a finished executable program. This implies that the results of a source-code generation tool must be passed onto another automatic programming tool in order to be executed.
The motivation for source-code generation is similar to that of automatic programming, however, it is intending to refine the specification.

This R&D project tries to investigate the usage of the library LibTooling for writing C++ source-code generation tools. The project involves the writing of three separate tools of increasing complexity that address some issues related to writing source-code generation tools through LibTooling.

The first tool is a simple renaming tool that can be used to rename functions and their matching invocations. The second tool is a refactoring tool that transforms arrays written using C-style notation into the more modern and secure `std::array` notation.
The third tool generates a `to_string` function for each enum declaration inside the program, i.e., a function that takes an instance of an enum as an argument and returns a string corresponding to the name of the value of the enum. If an existing `to_string` function exists, e.g., in another namespace, the tool overwrites it instead.

2.1 Technology overview

The LLVM project is a collection of compiler and toolchain technologies that can be used to build compiler-frontends for programming languages and compiler-backends for different instruction set architectures [4]. One project built with LLVM is Clang which is a compiler-frontend for languages in the C language family including C, C++, Objective C and many others [5].
The library LibTooling was built as part of the Clang project to allow developers to build standalone tools based on Clang [6]. It provides the benefit of using the AST generated by Clang and thereby guaranteeing that the parsed source code is valid. Furthermore, the AST is very fine-grained allowing for a detailed analysis of the provided source code.

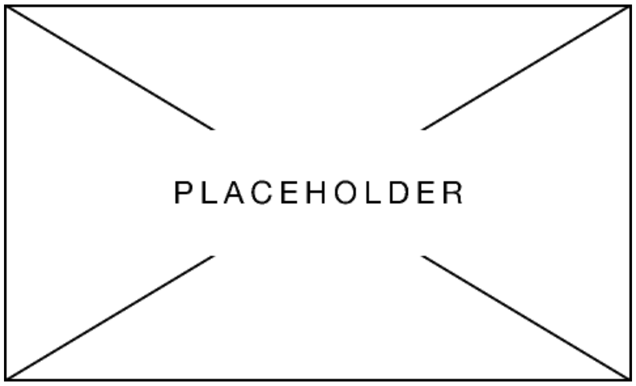

Figure 2.1: LLVM and Clang overview

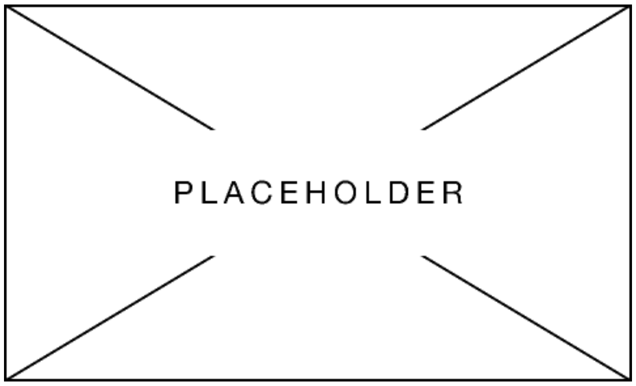

Figure 2.2: Clang and LibTooling overview

2.2 Project delimitations

While LibTooling supports writing tools for programming languages in the C language family, the project is delimited to focusing on tools written for C++.

In many cases, it may be beneficial to integrate the tools developed with LibTooling into the development flow by providing it as a Clang plugin. The project will not go into this topic but the tools built should be portable as Clang plugins.

At last, LibTooling provides several APIs for writing standalone tools. The APIs look different but essentially they provide the same functionality. They can all be used to write tools which use the Clang AST but do so following different software patterns. A few examples of the APIs are RecursiveASTVisitor, LibASTMatchers and Clang Transformer [7, 8]. The project will not go into detail about these APIs but instead, focus on using Clang Transformer.

3 Methods

4 Requirements

4.1 Functional requirements

4.2 Non-functional requirements

5 Architecture

6 Development

6.1 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 [9] 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 6.1. 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.

```bash
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 6.1: 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 6.2. 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.[10]

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

Listing 6.2: 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 [11], 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 6.3: Bash commands to compile LLVM, LibTooling and Clang with Clang as compiler.

Finally, the steps from listing 6.2 should be repeated to verify the recompilation and install the tools.

6.2 Build environment

The documentation for writing applications using LibTooling such as [12, 13] 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 [14] but the project was abandoned in 2020 and LLVM has since moved from a distributed repository architecture to a monolithic one making most of [14] obsolete. The following section is dedicated to describing the important decisions made related to the build environment.

6.2.1 Build settings

Initially, some general settings for the project are configured which can be seen in listing 6.4. 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 [15]. 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 6.2.2. 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 propagated to the subprojects that were compiled with LLVM such as LibTooling. By default a project in CMake is compiled with RTTI and CMake will assume that the used libraries were compiled with the same flags. This will result in nasty linker errors 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 6.4: General settings for the CMake build environment.

6.2.2 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.

6.2.3 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.[16] 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.

6.3 Tool structure

Through inspection of existing LibTooling tools, such as [17] and [18], and the development of tools for the R&D project, a common structure for LibTooling tools has been identified. The structure can be used as a way of categorizing the different parts of such tools and can be seen below.

- Command line parsing
- AST node matching
- Node data processing
- Handling the results

The **Command line parsing** part 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 6.3.1.

The **AST node matching** part of the tool structure is where the AST node matchers are defined and certain nodes of interest are bound to identifiers. 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. For a detailed description of the AST matching step, see section 6.3.2.

The **Node data processing** part 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 used to extract information from bound nodes and they are described in section 6.3.3.

The **Handling the results** part of the tool structure is where the matching and processing, defined in the previous parts, is applied. This is also where the results are handled and presented to the invoker of the tool. This part of the tool structure is described further in section 6.3.4.

6.3.1 Command line parsing

A good way of configuring the behaviour of a tool is through command line arguments. All LibTooling tools come with some common options which are always present, e.g., the option to parse a list of source files.

The way to add command line options to a tool is through the command line library provided by LLVM. This library makes it very easy to add new commands and provide help text for the options. A great example of how to use the library can be found in the implementation of ClangFormat [19], where they have multiple options with different types and default values.

6.3.2 Matching the AST

The next step in working with LibTooling is to find the AST that is relevant for the tool through matching and binding. This is a very important part of the tool development as it defines what kind of information is available for the later stages of the tool. If the wrong AST is matched or the wrong nodes are bound, the later processing can become unnecessarily complex and complicated. Furthermore, the task of matching the AST can also be very difficult as there are currently more than 700 different matchers available as part of the library [20].

A proposed way of matching the AST is by writing or finding a very simple example of the code that should be matched upon. The entire AST for that file can be printed 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 analyzed manually for the wanted nodes/patterns.

When an AST node has been identified as interesting to the tool it should be matched, which is typically done by using the predefined matchers of LibTooling [20]. It is also possible to write custom matchers, which can be very powerful when the standard matchers are lacking. An example of such could be if one wishes to recursively traverse the declaration context, which was the case in one of the implementations of the enum tool. However, most the time the builtin matchers will be sufficient and they should therefore be the initial place to look when matching the AST.

LibTooling defines three different basic categories of "Matchers" available to the users: Node Matchers, Narrowing Matchers and Traversal Matchers. The Node Matchers are the most general matchers and matches specific types of AST Nodes, e.g., `enumDecl` and `constantArrayType` . A subset of the Node Matchers are also bindable, meaning that they can be bound to an identifier and processed in the later stages of the tool. The Narrowing Matchers can be used to filter nodes that fulfil certain requirements, e.g. `hasName` and `isClass` . It also contains logical expressions such as `allOf` , `anyOf` and `unless` . The Traversal Matchers can be used to traverse the node with its parents or children and thereby specify and bind to certain relations of subnodes. Examples of Traversal Matchers are `hasDescendant` , `specifiesType` [20].

When composing a matcher, i.e., combining several matchers, one typically starts by identifying the overall node type that wishes to be matched. This will be the outermost Node Matcher and in most cases, this should be bound. A combination of narrowing matchers and traversal matchers can then be used to filter the matches depending on the information the tool needs. In many scenarios, it may also be necessary to bind child nodes in order to get all the needed information.

Putting it all together, one could write a matcher that matches function declarations that are named "f" and that takes at least one parameter. Furthermore, the function declaration and parameter declaration could be bound to the identifiers "function" and "parameter". This can be achieved using a combination of the three categories of matchers with the statement:

```
functionDecl(hasName("f"), hasParameter(0, parmVarDecl().bind("parmBind"))).bind("funcBind")
```

In the example the matchers `functionDecl` and `parmVarDecl` are Node Matchers, `hasName` is a Narrowing Matcher and `hasParameter` [2] is a Traversal Matcher.

[2]The 0 provided in `hasParameter` indicates that it must match the first argument of the function.

The composition of matchers is typically done iteratively where one starts by matching a superset of nodes which are iteratively narrowed down until the tool is left with the nodes that contain the needed information.

6.3.3 Node data processing

When the nodes have been matched and bound, the information stored inside of them must be extracted. The way to do so is by using an object that inherits from `MatchFinder::MatchCallback`. The `MatchFinder::MatchCallback` class defines the `run` method which must be overridden by its children. The run method has a `const MatchFinder::MatchResult &Result` parameter which contains the bound nodes of the match (e.g. “funcBind” and “parmBind” in the example provided earlier). The information inside the nodes can then be extracted and used e.g., for source code generation, diagnostic messages or other information relevant to 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 an AST edit and potential metadata. The AST edit is a change in the source code comprised of a source code location to rewrite and a concatenation 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 6.3.3.

The result consumer is responsible for saving the relevant results so they can be processed by the `ClangTool` later. The result consumer is further described in section 6.3.3.

Stencils

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

Examples of use-cases for the stencil interface could be to extract the element type of an array or to issue a warning at a given location. The functions of the predefined stencils are primarily focused on control flow, concatenation 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 necessary data from the bound nodes.

Luckily the stencil interface 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::MatchResult &)` with `std::string` as the template parameter. This API allows the creator of a tool to write small methods that extract the necessary information from a bound node as a string and seamlessly concatenate them together through the `cat` stencil.

All the predefined stencils return strings but it is possible to create an similar library that returns any type of data if that is required for the tool. The reason the predefined stencils work solely with strings is that it is primarily used to generate source code changes which must be converted to strings of some sort in order to be written to disk.

Similarly to the stencil interface, LibTooling also defines the range selector interface. This interface also builds upon `MatchConsumer<T>` but with `CharSourceRange` as an output instead of strings. A `CharSourceRange` refers to a range of characters defined at a specific location of the provided source files. It thereby allows the tool implementor to add or modify source code exactly where they want it.

Combining matchers and stencils

It should be clear by now that in order to create tools, it is necessary to have both matchers and stencils. Within this lies some interesting design decisions of how the matchers and stencils should be used in conjunction.

In general, there are two approaches to take when using matchers and stencils in conjunction.
The first approach is to create a simple matcher that binds only to the outermost node and then create detailed stencils that extract the information based on the single binding.
The second approach is to create a detailed matcher that binds to multiple nodes and then create simple stencils that utilize the many bindings.
Both approaches can be used to implement the same functionality but the implementations look vastly different.

If the implementor chooses the first approach, then the stencils can easily become complicated because one must filter and extract information from a single node.

If the second approach is chosen, the responsibility of extracting information from the nodes is placed inside the composed matcher. This can lead to some very complex matchers that can be difficult to understand. However, the stencils that extract information from the bound nodes will be much simpler.
Furthermore, when following this approach one can easily fall into the pitfall of trying to match too much data that is not required at the end.
E.g., when developing the enum tool it was attempted to bind the namespace of the parameter of an existing `to_string` function since it was thought to be needed when writing the transformation. In the end, the binding was unnecessary as the namespace could easily be extracted through a stencil and the namespace binding itself did not provide enough context. However, a significant amount of time was spent on writing an exact matcher that could recursively traverse the namespace qualifiers of a parameter and bind it.

Ultimately, the best approach to follow depends upon the specific scenario. In some cases, it may be better to write detailed stencils and in other cases detailed matchers. The important thing is to not tunnel-vision too heavily on a single approach and keep an open mind towards the other. Perhaps the best approach lies within a mixture of the two.

Consuming the transformation changes

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

The consumer can make decisions 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 external variables so it can be later used for further processing.

6.3.4 Handling the results

When the rules and transformers have been specified it is time to run them on the source code, which is done through a `ClangTool`. `ClangTool` is the API that runs the match finders over all the specified source code. All tools made with clang need to use `ClangTool` to tie it all together.
The `ClangTool` class has a `run` method which takes a `FrontendAction` and runs it on the specified source code. This can be considered the method that executes the tool.

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

7 Tool examples

This section contains examples of implementations of clang tools. The purpose of the examples is to show how the theory described in section 6.3 can be used. All the code for the examples can be found in the git repository for this project.

Each example will be split into the four sections of a clang tool as described in section 6.3.

7.1 Simple rename refactoring tool

The goal of this tool is to rename all functions in the provided source code that has the name "MkX" into "MakeX". The tool should both rename the function declaration and the locations where it is called. The code in this section has been mostly stripped of the namespace specifiers in order to simplify the code.

Command line parsing

In order to make this tool as simple as possible, the name of the method to rename and the new name have been fixed in the code. Therefore the Command line parsing element of the tool will use only the general options available for all LibTooling tools. The common command line options can be used by making a `CommonOptionsParser`. The way to create such an object can be seen on listing 7.1.

```cpp
int main(int argc, const char* argv[]) {
        auto ExpectedParser = CommonOptionsParser::create(argc, argv, llvm::cl::getGeneralCategory());
        if (!ExpectedParser) {
                // Fail gracefully for unsupported options.
                llvm::errs() << ExpectedParser.takeError();
                return 1;
        }
        CommonOptionsParser &OptionsParser = ExpectedParser.get();

    return 0;
}
```

Listing 7.1: Example code which shows the creation of the `CommonOptionsParser` used for all ClangTools.

AST node matching

For this tool to work two different types of nodes need to be matched. First, the function declaration with the name “MkX” has to be matched, and then all expressions which call the method have to be matched. This can be achieved through the two matchers shown in listing 7.2 and listing 7.3.

```
auto functionNameMatcher = functionDecl(hasName("MkX")).bind("fun");
```

Listing 7.2: This example shows a matcher that will match on any function declaration which has the name “MkX”.

```
auto invocations = declRefExpr(to(functionDecl(hasName("MkX"))));
```

Listing 7.3: This example shows a matcher that will match on any expression which calls to a function declaration with the name “MkX”.

Node data processing

In this tool, the act of processing the nodes is simple, as the tool just has to rename the method and all the locations where it is called. This is a native part of the rules API as described earlier (section 6.3.3).

The two renaming rules can be seen on listing 7.4 and listing 7.5.

```
auto renameFunctionRule = makeRule(
        functionNameMatcher,
        changeTo(name("fun"), cat("MakeX")));
```

Listing 7.4: The rename function rule used in the example. The rule consists of the functionNameMatcher as specified in listing 7.2 and the renaming action. In this case, the action is to change the name of the bound method to “MakeX”.

```
auto renameInvocationsRule = makeRule(
    invocations, changeTo(cat("MakeX")))
```

Listing 7.5: The rename invocations rule which updates the invocations to the renamed method. Here the entire expresion is changed to the new method name.

The two rules specified here are closely coupled as running just one of the rules would result in invalid source code. There is a way to group rules into a single rule and it is called `applyFirst`. This method creates a set of rules and applies the first rule that matches a given node. That means that there is an ordering to `applyFirst`. This ordering can be ignored for independent rules, like the two specified in this section, and in that case, it will simply create a disjunction between the rules. The combined rule can be seen on listing 7.6.

```
auto renameFunctionAndInvocations = applyFirst({renameFunctionRule, renameInvocationsRule});
```

Listing 7.6: A rule that both renames the function declaration and the invocations of that function.

The rules required for the simple renaming tool have been specified, but in order to extract the source code changes specified by the rules they have to be coupled with a transformer. The transformer is described in detail in section 6.3.

The transformer needs a consumer that saves the generated source code edits to an external variable. The consumer callback receives an expected array of `AtomicChange` objects which in turn contain the `Replacements` in the actual source code. The consumer shown in listing 7.7 extracts the `Replacements` from the `AtomicChange`s and saves them in a map variable that is defined externally.

```
auto consumer(std::map<std::string, Replacements> fileReplacements) {
    return [=](Expected<TransformerResult<void>> Result) {
        if (not Result) {
            throw "Error generating changes: " + toString(Result.takeError());
        }
        for (const AtomicChange &change : Result.get().Changes) {
            std::string &filePath = change.getFilePath();
            for (const Replacement &replacement : change.getReplacements()) {
                Error err = fileReplacements[filePath].add(replacement);

                if (err) {
                    throw "Failed to apply changes in " + filePath + "! " + toString(std::move(err));
                }
            }
        }
    };
}
```

Listing 7.7: A transformer consumer that saves all the generated source code edits to an external map by filename.

The consumer and the rules can be used to create the transformer as shown in listing 7.8.

```
Transformer transformer(renameFunctionAndInvocations, consumer(externalFilesToReplaceMap));
```

Listing 7.8: A rule that both renames the function declaration and the invocations of that function. The externalFilesToReplaceMap variable passed to the consumer will be discussed later.

Handling the results

This part of the tool is responsible for the creation of the actual tool and saving the results to disk.

The goal of this tool is a form of refactoring and Clang already has a tool for refactoring called clang-refactor. This tool is also created through LibTooling and it defines a class called `RefactoringTool` which extends the

`ClangTool` class. The `RefactoringTool` adds a way to save `Replacements` to disk. The changes that should be saved to disk are located in a `std::map<std::string, Replacements>` map which is contained inside of the `RefactoringTool`. The `RefactoringTool` implementation already contains all the needed functionality to finish the rename refactoring tool. All that remains is therefore to create the tool and invoke it, which is shown in listing 7.9.

```
int main(int argc, const char* argv[]) {
// CL parsing
...

RefactoringTool Tool(OptionsParser.getCompilations(),
                     OptionsParser.getSourcePathList());
auto &externalFilesToReplaceMap = Tool.getReplacements();

// transformer creation
...

//Register the transformation matchers to the match finder
MatchFinder Finder;
transformer.registerMatchers(&finder);

//Run the tool and save the result to disk.
return Tool.runAndSave(newFrontendActionFactory(&finder).get());
} // end main
```

Listing 7.9: This code snippet shows the creation of a `RefactoringTool` called ‘Tool’. The construction of the tool requires the source code that was passed through the command line. The internal map in the Tool is used as input to the transformer, as seen in listing 7.8.

As can be seen in listing 7.9 the tool combines the results from the other parts of the tool structure into the final tool. The tool is then invocated by the `runAndSave` method call, which invokes the tool and saves the results to disk afterwards.

7.2 CStyle array converter

The goal of this tool is to find all the raw CStyle arrays in C++ code and convert them into `std::array`s. This change provides more type information to the compiler without changing the intent of the code.

One of the test cases for this tool is to convert `static const int *const_pointer_array_static[2]` into `static std::array<const in`
In order to achieve this, the storage class and qualifier of the type must be preserved through the transformation, which is a bigger challenge compared to the simple rename tool section 7.1.

Command line parsing

Like the renaming tool section 7.1 the customization of the command line arguments for this tool has been left out in order to cut down on complexity.

AST node matching

This tool works on CStyle arrays with a constant size so the AST matcher for that type of node must be identified.

A CStyle array is a type and multiple matchers which match on different variants of CStyle arrays are provided by Clang. The types of CStyle arrays are: `Array`, `Constant`, `DependentSized`, `Incomplete` and `Variable`. The `Array` type is a base type for all the other types of CStyle arrays. The `Constant` arry type is a CStyle array with a constant size. The `DependentSized` array is an array with a value dependent size. The `Incomplete` array is a CStyle array with an unspecified size. The `Variable` array type is a CStyle array wit ha specified size that is not an integer constant expression.

Each of the types have a corrosponding matcher which allows the creator of a tool to match only the wanted types of CStyle arrays. The focus of this tool is solely `Constant` arrays, as they are directly convertible to `std::array`s. The same would probably also be true for the `DependentSized` array type, but this has been left out in order to simplify the tool.

Node data processing

Handling the results

7.3 Enum to string tool

This section describes a tool that is capable of generating `std::string_view to_string(EnumType e)` functions for each enum declaration defined in a C++ program. The `to_string` functions take an instance of the enum as argument and returns a string corresponding to the name of the enumerator.

An example of the outputs of running the tool can be seen in listing 7.10. In the example, at part (1), the enum `Animal` is declared with two enumerators: Dog and Cat. In part (2), the `to_string` function that would be generated by the tool can be seen. Part (3) and (4) shows another enum declaration with another generated `to_string` function.

```cpp
// (1): Example enum declaration:
enum class Animal{
    Dog, // Dog is an example of an enumerator (aka. enum constant)
    Cat // Cat is another example of an enumerator
};



// (2): Function that the tool generates:
constexpr std::string_view to_string(Animal e){
    switch(e) {
        case Animal::Dog: return "Dog";
        case Animal::Cat: return "Cat";
    }
}

// (3): Another enum declaration:
enum Greetings {
    ... // enumerators for Greetings
};

// (4): The other enum declaration also gets a to_string function
constexpr std::string_view to_string(Greetings e){
    switch (e) {
        ...
    }
}
```

Listing 7.10: Example (1) declaring an enum in C++ and (2) the `to_string` function that the tool generates. In (3) another enum was declared from which another `to_string` function is generated (4).

The tool differentiates itself from the previous examples by being a generative tool, meaning that it inserts source-code into a file. In contrast, the renaming tool and CStyle comparison tool were refactoring tools that would overwrite existing code-lines. While the difference may seem subtle, it can be more challenging to design generative tools, as there are stricter requirements related to analyzing if the generated code is syntactically correct when combined with the existing code-base.
E.g., with the enum to string tool it is necessary to determine if there already exists a function named `to_string` in the same namespace that takes a single parameter of the enum type, since redefinition of

functions is not allowed in C++. In case such a function already exists, one needs to determine a strategy on how to handle the conflicts, e.g., by leaving the function untouched or overwriting it. The act of determining whether such a function exists can be quite complex, as one must analyze the compilation-unit for its existence.

Likewise, there are typically extra semantic considerations to be made when designing a generative tool. E.g., if `std::string_view to_string(Animal e)` function exists in a namespace "A" and `Animal` was declared in namespace "B", then it would be syntactically correct to add the `to_string` function to namespace "B". The question of whether it semantically makes sense for these functions to coexist arises [1] and one needs to select a strategy for handling such scenarios.
Examples of such strategies could be to ignore the cases, warn the user about them, delete the non-generated version or overwriting the non-generated version.
It can be difficult to find and consider all the possible semantic strategies when developing a tool. For some problems there could be infinite ways to generate the wanted behaviour, like there is when creating a program through a programming language. Some of the posibilities may be better than others but there is still a large design space that could be explored. To demonstrate this, a list of scenarios where one might need to consider the behaviour for the enum to string tool can be seen below. The examples in the list increasingly become more abstract and difficult to implement.

- A `to_string` function taking multiple arguments already exists.
- The enum is declared privately inside a class.
- The enum is declared inside an anonymous namespace.
- The enum is declared inside a namespace that by convention is intended to be ignored by users (e.g., `detail`, `implementation`, etc.).
- A function that implements the same behaviour as the generated `to_string` function exists.
- A function that implements a similar behaviour as the generated `to_string` function exists.
- A similarly named function exists that implements a similar behaviour as the generated `to_string` function exists.
- ...

For the enum to string tool, it was decided to overwrite syntactically conflicting implementations of the `to_string()` functions. This has the benefit of allowing the user to change the enum and re-run the tool to see the updated changes. E.g., in listing 7.10 if a `Animal::Horse` was added to the enum declaration, re-running the tool would update the corresponding `std::string_view to_string(Animal e)` function. However, it also has the downside of essentially reserving the `to_string` name leaving the user unable to write their own versions of the function.
Furthermore, it was decided to also implement the semantic rule that if a `to_string` function already exists in a different namespace, then the existing version must be overwritten. This was mainly decided as there are some interesting challenges to consider in relation to recursively traversing the namespaces, which are described in section 7.3.2.

[1] This must be determined on a case-by-case basis. E.g. it might make sense for two `print(X)` functions to exist in seperate namespaces. One that is part of the public API and one that is intended for debugging. However, it might not make sense for two `release(X)` functions to exist in seperate namespaces as this would indicate there are several ways of releasing the ressources allocated in X. (And yet in other cases it might make perfect sense for two `release(X)` functions to exist.)

7.3.1 Command line parsing

7.3.2 AST node matching

8 Testing

9 Related work

10 Conclusion

10.1 Future work

References

[1] Avron Barr and Edward A. Feigenbaum. "Automatic Programming". In: *The Handbook of Artificial Intelligence*. Vol. 1. Elsevier, 1982, pp. 295–379. ISBN: 978-0-86576-090-5. DOI: 10.1016/B978-0-86576-090-5.50010-0. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780865760905500100 (visited on 01/31/2023).

[2] Gordon S. Novak Jr. *CS 394P: Automatic Programming p. 2*. The University of Texas at Austin. URL: https://www.cs.utexas.edu/users/novak/cs394p2.html (visited on 04/03/2023).

[3] Visual Paradigm. *UML/Code Generation Software*. URL: https://www.visual-paradigm.com/features/code-engineering-tools/ (visited on 04/03/2023).

[4] LLVM. *The LLVM Compiler Infrastructure Project*. URL: https://llvm.org/ (visited on 04/03/2023).

[5] LLVM. *Clang C Language Family Frontend for LLVM*. URL: https://clang.llvm.org/ (visited on 04/03/2023).

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

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

[8] LLVM. *Welcome to Clang's Documentation! — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/index.html (visited on 04/03/2023).

[9] *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).

[10] *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).

[11] 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).

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

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

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

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

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

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

[18] LLVM. *The LLVM Compiler Infrastructure - GitHub*. LLVM, Apr. 3, 2023. URL: https://github.com/llvm/llvm-project (visited on 04/03/2023).

[19] LLVM. *Clang Format - ClangFormat.Cpp - GitHub*. LLVM, Apr. 3, 2023. URL: https://github.com/llvm/llvm-project/blob/db3dcdc08ce06e301cdcc75e2849315a47d7a28d/clang/tools/clang-format/ClangFormat.cpp (visited on 04/03/2023).

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