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
MAY 23, 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.
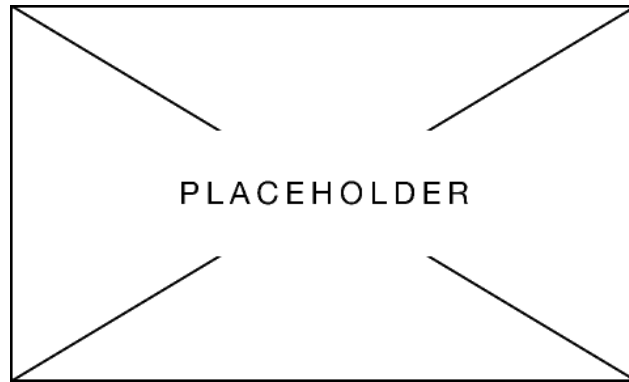
Figure 2.1: LLVM and Clang overview
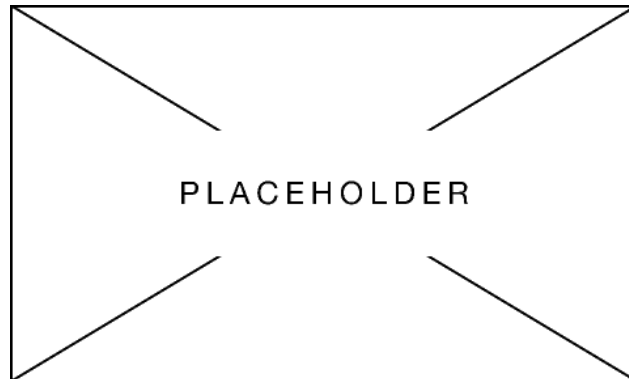
Figure 2.2: Clang and LibTooling overview

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

As described earlier three different tools will be developed during this project. The tools are used as a progressive learning platform for exploring increasingly complex parts of the LibTooling framework.

The first tool is a renaming tool, that will refactor a method with an illegal name ("MkX") into a legal name ("MakeX"). The tool will also rename all the calls to that method in order to keep the exact same functionality. The purpose of developing this tool is to get familiar with the basics of the LibTooling framework, which will make later development easier.

The second tool is also a refactoring tool but with more complexity than the renaming tool. The purpose of the second tool is to convert traditional C-style arrays into the more modern and strongly typed `std::array`s. This tool is more complex than the renaming tool because there is more information associated with arrays than function names. Furthermore, there are more semantic considerations which have to be taken into account when making this type of change.

The third tool will analyze the code base for enums and generate "to_string" methods for each enum definition. The "to_string" method will return a string representation of the named enum constants defined in the enum. In order to ensure valid source code after the tool is run, existing "to_string" methods should be overwritten or updated as a part of the process. This tool is comparable in complexity to the CStyle converter tool but adds the complexity of code generation to the tool.

Through these three tools, a deeper understanding of the LibTooling framework and the C++ language is obtained.

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.

```
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]

```
1  ninja check
2  ninja clang-test
3  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.

```
1  cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -DCMAKE_BUILD_TYPE=Release
   ↪  -DCMAKE_CXX_COMPILER=clang++
2  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.

```
1  set(CMAKE_CXX_COMPILER clang++)
2  set(CMAKE_CXX_STANDARD 17)
3  set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")
4  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 AST node matching

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.

7.1.1 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.

```
1  int main(int argc, const char* argv[]) {
2      auto ExpectedParser = CommonOptionsParser::create(argc, argv, llvm::cl::getGeneralCategory());
3      if (!ExpectedParser) {
4          // Fail gracefully for unsupported options.
5          llvm::errs() << ExpectedParser.takeError();
6          return 1;
7      }
8      CommonOptionsParser &OptionsParser = ExpectedParser.get();
9
10      return 0;
11  }
```

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

7.1.2 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.

```
1  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".

```
1  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".

7.1.3 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.

```
1  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".

```
1  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.

```
1  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 external-FilesToReplaceMap variable passed to the consumer will be discussed later.

7.1.4 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.

```
1   int main(int argc, const char* argv[]) {
2   // CL parsing
3   ...
4
5   RefactoringTool Tool(OptionsParser.getCompilations(),
6                        OptionsParser.getSourcePathList());
7   auto &externalFilesToReplaceMap = Tool.getReplacements();
8
9   // transformer creation
10  ...
11
12  //Register the transformation matchers to the match finder
13  MatchFinder Finder;
14  transformer.registerMatchers(&finder);
15
16  //Run the tool and save the result to disk.
17  return Tool.runAndSave(newFrontendActionFactory(&finder).get());
18  } // 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 invoked by the `runAndSave` method call, which invokes the tool and saves the results to disk afterwards.

7.2 CStyle array converter

The traditional way of making arrays in C is by using the subscript operator. A C array consists of a type, a name and a constant size. The line: `int my_array[10]` will create a variable called "my_array" which is a collection of 10 consecutive integers. In C++11 a standard container called `std::array` was added to C++. This container uses templates to create stronger typed arrays and to ensure the element type and the array size are known at compile time. This change gives the compiler more information about the intent of the code and potentially allows the backend of the compiler to make more optimisations. In C++ both ways of declaring arrays are allowed and the C way of declaration is called a CStyle array.

The goal of this tool is to find all the 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 const_pointer_const_array_static[2]` into `static std::array<const int* const>, 2> const_pointer_const_array_static`. 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.

7.2.1 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.

7.2.2 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 in the Clang AST. The library contains multiple matchers which match different variants of CStyle arrays. 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` array 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 with a specified size that is not an integer constant expression.

Each of the types has a corresponding matcher which allows the creator of a tool to match only the wanted types of CStyle arrays. The focus of this tool is solely on `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.

The constant array type node contains the element type of the array as well as the number of elements in the array. The constant array type node does not contain the storage specifier for the type (e.g., static or extern) so that information must be bound in another way. The storage specifier is stored in the declaration of the array which is a `VarDecl` . For this tool, the bound declaration will also make it easy to access the qualifiers for the CStyle array.

There is a problem however as this tool also aims to refactor the raw CStyle arrays declared inside of classes. This distinction matters for the declaration of the variables, as variables inside classes are `FieldDecl` s and not `VarDecl` s. For this matcher, the type of node to match must therefore be more generic than the `VarDecl` . This can be achieved by using the `DeclaratorDecl` matcher which matches on `DeclaratorDecl` s, which is the common base class between `VarDecl` s and `FieldDecl` s.

The variable declaration for the arrays also contains the bounded location of the declaration in the source code. This can be easily extracted through the `typeLoc` matcher. This information will make it easier to change the correct location in the source code in the following steps and is therefore present in the matcher.

```
1   auto ConstArrayFinder =
2       declaratorDecl(
3           isExpansionInMainFile(),
4           hasType(constantArrayType().bind("array")),
5           hasTypeLoc(typeLoc().bind("arrayLoc")))
6       .bind("arrayDecl");
```

Listing 7.10: CStyle array matcher with bindings.

7.2.3 Node data processing

In this section, the relevant data from the bound nodes will be extracted and used to generate source-code changes. The Stencil and Transformer libraries will be used for the data extraction and code refactoring. As described in the previous section, there are multiple pieces of information which need to be extracted from the nodes.

From listing 7.10 three node types were bound. "array" is the type of constant array that was found, "arrayLoc" is the location of the constant array in the source code and "arrayDecl" is the entire declaration of the array.

As shown in the introduction to this section, a test case for this tool is to convert `static const int* const ID[2]` into `static std::array<const int* const, 2> ID`.

RangeSelector

The first thing needed in order to make the change is to extract the source code range from the nodes. The "arrayLoc" node, which has been bound, is a `TypeLoc` which spans the `[number_of_elements]` part of the CStyle array. This may seem a bit confusing at first, but the behaviour is due to the `typeLoc` referring to the underlying `ArrayTypeLoc`, i.e., the part of the source code that makes it a CStyle array. "arrayLoc" can be used in conjunction with the "arrayDecl" node in order to get the entire source code range for the array declaration. The `RangeSelector` API has a convenience method `encloseNodes` which is meant for exactly this purpose.

StorageClass

Now that the correct source code range has been found, it is time to populate it with the correct information. As specified earlier the storage specifier for the declaration must be kept and it can be found through the `getStorageClass` method on VarDecl nodes. This method is not present in FieldDecl nodes because they don't have storage specifiers. static FieldDecls inside classes are converted to VarDecls during the AST generation, as these variables are independent of the individual class instances.

The `getStorageClass` method can be implemented as a `MatchConsumer<std::string>` which will make it easy to use the method to make source-code refactoring (see section 6.3). An implementation of such a method can be seen on listing 7.11.

```cpp
auto getVarStorage(StringRef Id) {
    return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
        if (auto var = Match.Nodes.getNodeAs<VarDecl>(Id)) {
            auto storage_class = var->getStorageClass();
            if (storage_class != StorageClass::SC_None) {
                return std::string(VarDecl::getStorageClassSpecifierString(storage_class)) + " ";
            }
        }

        return "";
    };
}
```

Listing 7.11: Method to extract the storage specifier string from a VarDecl node bound to Id.

Array element type

The array element type is stored inside the `ConstantArrayType` node. It can be accessed through the `getElementType` method call. Like the storage class specifier on listing 7.11 the easiest way to work with the element type for this tool is through the Stencil library. The getArrayElementType method will therefore be implemented as a MatchConsumer. The implementation of the method can be seen on listing 7.12.

```
1  auto getArrayElementType(StringRef Id) {
2      return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
3          if (auto array = Match.Nodes.getNodeAs<ArrayType>(Id)) {
4              return array->getElementType().getAsString();
5          }
6
7          throw std::runtime_error("ID not bound or not ArrayType: " + Id.str());
8      };
9  }
```

Listing 7.12: Method to extract the element type from the ConstantArrayType node.

Array size

The size of the array can also, like the array element type, be extracted through the ConstantArrayType node. The implementation of the extraction method can be seen on listing 7.13.

```
1   auto getConstArraySize(StringRef Id) {
2       return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
3           if (auto array = Match.Nodes.getNodeAs<ConstantArrayType>(Id)) {
4               auto size = array->getSize().getZExtValue();
5               std::stringstream ss;
6               ss << size;
7               return ss.str();
8           }
9
10          throw std::runtime_error("ID not bound or not ConstantArrayType: " + Id.str());
11      };
12  }
```

Listing 7.13: Method to extract the element size from the ConstantArrayType node. The array size is a llvm::APInt and must be converted to a `uint64_t` through the `getZExtValue` method.

Declaration qualifiers

The last piece of the puzzle is to keep the qualifiers of the declaration. The qualifiers are the explicit namespaces in front of the name of the declaration. E.g., `uint8_t my_namespace::array[5];` has the qualifiers `my_namespace::`, while `uint8_t array[4];` has no qualifiers. These qualifiers are part of the declaration and the easiest way to extract them is to copy the literal text in the source code. The literal source code of

declarations can be extracted through the source range of the node. listing 7.14 shows how the qualifiers of a `DeclaratorDecl` can be extracted from the source-code.

```cpp
auto getDeclQualifier(StringRef Id) {
    return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
        if (auto decl = Match.Nodes.getNodeAs<DeclaratorDecl>(Id)) {
            CharSourceRange qualifierRange = getTokenRange(decl->getQualifierLoc().getSourceRange());
            return getText(qualifierRange, *Match.Context).str();
        }

        throw std::runtime_error("ID not bound or not DeclaratorDecl: " + Id.str());
    };
}
```

Listing 7.14: Method to extract the qualifiers from the source code range of a DeclaratorDecl node.

Creating the transformation rule

With all the relevant information extracted from the bound nodes, a transformation rule can be made to create the wanted source-code refactoring. The code for the rule can be found on listing 7.15.

```cpp
auto CStyleArrayConverterRule = makeRule(
    ConstArrayFinder,
    {
        addInclude("array", IncludeFormat::Angled),
        changeTo(
            encloseNodes("arrayDecl", "arrayLoc"),
            cat(run(getVarStorage("arrayDecl")),
                "std::array<",
                run(getArrayElementType("array")),
                ", ",
                run(getConstArraySize("array")),
                "> ",
                run(getDeclQualifier("arrayDecl")),
                name("arrayDecl")
            )
        )
    });
```

Listing 7.15: The entire rule for generating the wanted `std::array` declaration. The rule both adds the `<array>` header and makes the source-code refactoring in one step.

As it can be seen from listing 7.15, the refactoring rule is relatively simple to read, as it solely uses the methods created through this section and strings of text, which is very descriptive. This makes each line easy to reason about and transparent. The difference between the node processing step for the CStyleConverter tool as compared to the RenamingTool, is that more information has to be extracted from the bound nodes. This requires a deeper understanding of which node types contain the wanted information, and how the

information is encoded in the nodes.

In the renaming tool (section 7.1) the `Tranformer` consumer extracted the `Replacements` from the `AtomicChanges` generated by the rule. This was necessary, as the tool used the `RefactoringTool` class provided by the library. In the renaming tool, the rule only had a single change for the matches, which was quite easy to extract from the `AtomicChanges`. In this tool, the rule adds multiple changes and it even includes the `array` header. This added complexity to the extraction of the replacements in the consumer. ' During the development of this tool it was discovered, that there is a convenience method in the library that allows the developer to apply `AtomicChanges` directly to the source code instead of unpacking it manually. This method will take care of all the header removals and includes them in each `AtomicChange`. The team wanted to use this feature in the tool to reduce the complexity of the consumer, so a custom `ClangTool` class had to be created.

The custom class has to work with `AtomicChanges` instead of `Replacements` but the overall structure of the `runAndSave` method, which saves the changes to disk, is the same. As opposed to `Replacement` objects, `AtomicChange` objects already contain the file name for the change so all the changes can be inserted into a single `std::vector` and handled by the tool instead of the consumer. This change in semantics is what makes the consumer method simpler.

The new consumer method takes a `AtomicChanges` reference as a parameter and returns a lambda method which is a `ChangeSetConsumer`. The lambda inserts valid `AtomicChange` objects into the provided `AtomicChanges` array for later processing. The implementation can be seen on listing 7.16.

```
1   auto RefactorConsumer(AtomicChanges &Changes) {
2           return [=](Expected<TransformerResult<void>> C) {
3               if (!C) {
4                   throw std::runtime_error("Invalid Changes! " + toString(C.takeError()));
5               }
6
7               Changes.reserve(Changes.size() + C.get().Changes.size());
8               std::move(C.get().Changes.begin(), C.get().Changes.end(),
9                       std::back_inserter(Changes));
10          };
11  }
```

Listing 7.16: Implementation of the `AtomicChange` consumer.

7.2.4 Handling the results

In the previous section, a new consumer was made for the `Transformer`. This consumer uses an external instance of `AtomicChanges` to save the results of the tool. After all the `AtomicChange` objects have been gathered, the changes can be applied to the source code and saved to disk - like it was done in the renaming tool. This logic has to be custom-made to use `AtomicChange` instead of `Replacements`. This change will be discussed here.

The renaming tool calls the `runAndSave` method on the `RefactoringTool` class. This method first runs the specified `FrontEndAction` to get the result of the tool. Afterwards, it creates a `Rewriter` with sources provided to the tool. The `Rewriter` class is an interface between the tool and the rewrite buffers - aka the OS. This `Rewriter` will allow the tool to make source-code changes. With the created `Rewriter` object, the

`RefactoringTool` calls the `applyAllReplacements` and `saveRewrittenFiles` methods. These methods will apply the `Replacements` to the source code and then save the rewritten source to disk. It is this behaviour that needs to be reimplemented for the `AtomicChanges`.

The creation of the `Rewriter` class is identical between the implementation in the library and the custom version. This is the case because the `Rewriter` is used solely to apply the changes and not to reason about the changes. The creation of the `Rewriter` instance has thus been excluded from this report. It can be found in the implementation of the CStyleArrayConverter on the project GitHub page.

The `applyAllReplacements` method is where the custom tool class gets interesting. This method has to be created by hand and it needs to call the `applyAtomicChanges` method on all the gathered `AtomicChanges` for each filename. The `applyAtomicChanges` method can apply changes from multiple `AtomicChange` objects at once, but they must be changes in the same source file. This has to be checked by the developer before the method is called. This check was performed in the consumer in the RenamingTool but has to be implemented in the tool here.

In order to apply all the gathered changes, the tool needs access to those changes. In the `RefactoringTool`, the `Replacements` container was a part of the tool instance. This part will be copied into the custom tool, so it contains an instance of the `AtomicChanges`.

The `AtomicChange` objects each contain the name of the file to which the changes should be applied. Because this information is present in the objects they can be grouped by their source files in a map. The grouping is shown in listing 7.17.

```
1   AtomicChanges Changes;
2   bool applyAllChanges(Rewriter &Rewrite) {
3       std::unordered_map<std::string, AtomicChanges> FileChanges;
4       for (const auto& Change : Changes) {
5           FileChanges[Change.getFilePath()].push_back(Change);
6       }
7       ...
8   }
```

Listing 7.17: Code snippet that will group the collection of `AtomicChange` objects based on the files they should change. The use of the unordered map is for performance reasons and could just as easily be a `std::map`.

Now that the changes have been grouped, it is time to apply the changes. In order to apply the changes, the specific source code and the file handle have to be used. This information can be extracted from the `Rewriter` through its internal `SourceManager` and `FileManager` using the file path stored in the groups. In order to get access to the source file from a given path, first an `FileEntryRef` is extracted from the `FileManager`. Then the `SourceManager` can be used to create a `FileID`, which can be used to access the source code. The created `FileID` can also be used to specify the range which should be overwritten by the `Rewriter`. Putting all of this together will commit the found file changes to disk. The implementation of this method can be seen on listing 7.18.

```
1   bool applyAllChanges(Rewriter &Rewrite) {
2       // Group the Changes
3       ...
4
5       auto &sm = Rewrite.getSourceMgr();
6       auto &fm = sm.getFileManager();
7
8       for (const auto &[File, ChangesForFile] : FileChanges) {
9           auto Entry = fm.getFileRef(File);
10          if (!Entry) {
11              llvm::errs() << Entry.takeError();
12              return false;
13          }
14
15          auto id = sm.getOrCreateFileID(Entry.get(), SrcMgr::C_User);
16
17          auto code = sm.getBufferData(id);
18          auto new_code = applyAtomicChanges(File, code, ChangesForFile, ApplyChangesSpec());
19
20          if (!new_code) {
21              llvm::errs() << new_code.takeError();
22              return false;
23          }
24
25          Rewrite.ReplaceText(
26              SourceRange(sm.getLocForStartOfFile(id), sm.getLocForEndOfFile(id)), new_code.get());
27      }
28
29      return Rewrite.overwriteChangedFiles();
30  }
```

Listing 7.18: Implementation of the `applyAllChanges` method which will apply all the generated `AtomicChanges` to the source code and save it to disk.

7.3 CStyle array parameter converter

During the development of the CStyle array converter tool (section 7.2) discussed what impact the transformation of the constant arrays would have on the target project. There are multiple side effects when the entire codebase is refactored to use `std::array`s instead of CStyle arrays. These side effects occur because CStyle arrays and `std::array`s have different semantic meanings and definitions. For one CStyle arrays decay to pointers when used as a function parameter, while `std::array`s are strong types that do not decay. In order to investigate the semantic implications and challenges which arise when doing the CStyle array conversion, a modification was made to the CStyle array converter tool, which makes the tool also convert function parameters into `std::array`s.

This additional transformation will show some of the considerations and challenges that will have to be considered when making the conversion over a codebase.

7.3.1 Command line parsing

The functionality described in this section is an extension of the CStyle array converter that is focusing solely on the parameter conversion, so it has no changes to the command line as compared to the original tool.

7.3.2 AST node matching

For this augmentation to the existing tool, a new matcher has to be created. The matcher must find all the function parameter declarations which are constant arrays. The matcher should additionally still extract the needed information like the variable declaration converter (section 7.2.2).

There is an existing matcher in the LibTooling catalogue that will match a function parameter declaration. That matcher is called `parmVarDecl` and it will be used in this matcher instead of the `declaratorDecl` used previously.

The `parmVarDecl` matcher will find all the parameter declarations, but for this tool only the constant array parameters are interesting. Because of this, the nodes must be filtered based on their types. An analysing the AST of a simple parameter example:

(**void** test(**int** parm[3]){})

through clang, shows that the constant array parameter type decays to a pointer type. This decay happens because of the semantics of the language. There is a way to match a node if the declaration decays to a pointer, and that is through the `decayedType` matcher which is provided in the library.

This matcher can be provided with an inner matcher which further filters the nodes that are matched. The inner matcher is called with a `AdjustedType` node, which is a type that has been implicitly adjusted based on the semantics of the language. The adjusted type contains both the adjusted type and the original type. In this case, the adjusted type is the pointer to which the array has decayed. This original type can be used to check if the decayed type was originally a constant array, which is what is needed for this tool.

No matcher in the library will ensure that the original type of an adjusted type is a constant array type so that matcher has to be implemented. Creating a custom matcher is done through helper macros in the library. There are many different helper macros, which allow the creator of the matcher to fine-tune the matcher to the exact needs. This flexibility also makes it somewhat complicated for first-time developers, as there are many options to sort through. In this case, the matcher is provided with an adjusted type node, which has to be filtered on the original type. The original type has to be compared with another type. This means that a parameter is needed for the matcher. The correct macro for this type of matcher is the `AST_MATCHER_P` macro. This macro allows the user to specify the input node type and a single parameter which is given to the matcher. Because this matcher will compare types, the type of the input parameter is a `Matcher<QualType>`. The signature of the matcher is shown on listing 7.19.

```
1   AST_MATCHER_P(AdjustedType, hasOriginalType,
2                   ast_matchers::internal::Matcher<QualType>, InnerType) {
3       ...
4   }
```

Listing 7.19: Signiture of the custom matcher `hasOriginalType`.

The implementation of the matcher is quite simple. As mentioned earlier, the matcher needs to extract the

original type from the AdjustedType and compare it to the type provided as the parameter of the matcher. This can be achieved with the code shown on listing 7.20.

```
1   return InnerType.matches(Node.getOriginalType(), Finder, Builder);
```

Listing 7.20: Implementation of the custom matcher `hasOriginalType`.

The `Finder` and `Builder` variables are common across all matchers. The `Finder` variable is the `MaatchFinder` variable created by the tool, and it is responsible for calling the callbacks when a valid match has been found. The `Builder` variable is used to bind nodes to specific names through the `.bind(NAME)` construct used in the tools.

With all the building blocks in place, the CStyle array parameter matcher can be constructed. It looks similar to the variable CStyle array matcher and is therefore very expressive. The CStyle array parameter matcher can be seen on listing 7.21.

```
1   auto ParmConstArrays = parmVarDecl(
2           isExpansionInMainFile(),
3           hasType(
4               decayedType(hasOriginalType(constantArrayType().bind("parm")))),
5           hasTypeLoc(typeLoc().bind("parmLoc"))
6       ).bind("parmDecl");
```

Listing 7.21: CStyle array parameter matcher.

7.3.3 Node data processing

This tool is a great example of the differences between extracting information through node processing as compared to filtering through matchers. The goal of this tool expansion is the same as it was for CStyle array variables, it just has to use `ParmVarDecl` s instead of `VarDecl` s. `ParmVarDecl` s are a specialisation of `VarDecl` s, so all the node processing is the same for both types. Therefore the node processing can be reused for this tool expansion, and the different match filtering is all that is needed.

7.3.4 Handling the results

As this is an expansion to the CStyle array converter tool, the handling of the results is identical.

7.4 Enum-to-string tools overview

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 7.22. In the example, in 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. Parts (3) and (4) show 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 7.22: 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).

7.4.1 Difference from previous tools

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 overwrite 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 possibilities 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 of 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 7.22 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 concerning recursively traversing the namespaces, which are described in section 7.5.2.

---

[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 separate 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 separate namespaces as this would indicate there are several ways of releasing the resources allocated in X. (And yet in other cases, it might make perfect sense for two `release(X)` functions to exist.)

7.4.2 Implementations

The enum-to-string tool is more complex than the previously described tools as it consists of matching on multiple independent declarations of the source code, i.e., the enum declarations and the existing `to_string` function declarations. The added complexity allows for a wider variety of design approaches, which was shown as part of the project, where three different implementations were considered and two were implemented.

The first approach could implement the tool in a single step, similarly to how it was done in section 7.2. Pseudocode for such a tool can be seen in listing 7.23. The pseudocode iterates over all the enum declarations in the source code and determines if a `to_string` already exists. Depending on the outcome, the function is either updated or generated.

```
1   for enum_decl in source_files:
2       to_string_inst := find(to_string(enum_decl))
3       if to_string_inst:
4           update to_string_inst
5       else:
6           generate to_string(enum_decl)
```

Listing 7.23: Pseudocode for the enum-to-string tool.

An alternative implementation of the tool can be seen in listing 7.24, which follows a multi-step procedure. The first step consists of updating the existing `to_string` functions and saving the relevant enum type in a collection (`parameters`). The second step consists of finding all the enum declarations and generating the `to_string` functions that were not updated in step 1.

```
1   parameters := []
2   for existing_enum_to_string in source_files:
3       update existing_enum_to_string
4       parameters.append(existing_enum_to_string.parameter)
5
6   for enum_decl in source_files:
7       if not enum_decl in parameters:
8           generate to_string(enum_decl)
```

Listing 7.24: Pseudocode for the enum-to-string tool.

A third way of implementing the tool can be seen in listing 7.25, which is also a multi-step procedure. The first step consists of identifying the existing `to_string` functions. The second step consists of finding all the enum declarations. If the declaration already has a `to_string` function then it is updated. Otherwise, it is generated.

The main difference between the second and the third implementation is regarding semantics. It might be simpler to divide the tool into two distinct phases consisting of a data collection phase and a post-processing phase, where the post-processing performs the actual logic of the tool. This appears to be a common division of responsibilities and is among others used in a helper library that was written by Bloomberg [21].

31

```
1  parameters := []
2  for existing_enum_to_string in source_files:
3      parameters.append(existing_enum_to_string)
4
5  for enum_decl in source_files:
6      if enum_decl in parameters:
7          update parameters[enum_decl]
8      else:
9          generate to_string(enum_decl)
```

Listing 7.25: Pseudocode for the enum-to-string tool.

The two designs from listing 7.24 and listing 7.25 have the benefit of being more modular than the one in 7.23 since they implement the tool behaviour in multiple steps. Each step in the designs can essentially be considered independent tools that are chained together. This makes it easier to reuse the tools in other projects and makes it easier to test during development. For instance in listing 7.24, the first for-loop can be considered a tool that identifies existing enum-to-string functions, logs them and updates them, and the second for-loop can be considered a tool that generates enum-to-string functions if they are not in the log.

During the project, the tool was implemented with the designs seen in listing 7.23 and listing 7.24. The design from listing 7.25 was considered but was not implemented.

The following sections describe the two "enum-to-string" tools that were implemented.

7.5 Enum-to-string – single-step

The following section describes the enum-to-string tool where the entire tool is implemented as a single `ClangTool`.

7.5.1 Command line parsing

Most of the command line parsing was done similarly as in section 7.1. However, the behaviour was extended with the introduction of some new command line options for the user to specify. The command-line option "in_place" was introduced, allowing the user to have the file changes written to the terminal instead of directly to the file. The other option "debug_info" makes the tool print extra debug information to the console.

The additional options were easily introduced as booleans options through the LLVM command line API, as seen in listing 7.26. One simply needs to specify a description and add it to the `OptionCategory` – which is `MyToolCategory` in this case.

```
1  static llvm::cl::opt<bool> Inplace(
2      "in_place",
3      llvm::cl::desc("Inplace edit <file>s, if specified. If not specified the "
4                     "generated code will be printed to cout."),
5      llvm::cl::cat(MyToolCategory));
6  static llvm::cl::opt<bool> DebugMsgs(
7      "debug_info", llvm::cl::desc("Print debug information to cout."),
8      llvm::cl::cat(MyToolCategory));
```

Listing 7.26: Implementation of the newly introduced command line options.

The options can then be used as normal booleans throughout the implementation, as seen in listing 7.27.

```
1  if (!Inplace) {
2      llvm::outs() << new_code.get();
3  }
```

Listing 7.27: Using the `Inplace` command line option to print the changes to the command line if `--in_place` was not specified when running the tool.

The addition of "in_place" option is in particular useful for the future development of tools, as it allows for better system testing, where test cases are written that compares the contents of a virtual file, before and after running the tool. Similar tests can be seen throughout the tools in the LLVM repository.

7.5.2 AST node matching

The AST node matching was by far the most challenging part of developing this tool as one needs to write a matcher that implements the following logic for each enum declaration:

1. Find and bind enum declaration

2. Find the outermost namespace

3. Recursively traverse the namespace to potentially find a matching `to_string` function

The behaviour described above is quite complex compared to the previous tools and required the implementation of custom matchers that worked recursively, something that the built-in matchers do not support.

The matcher will be described in the steps below.

Finding enum declarations

A matcher for finding the enum declarations can be seen in listing 7.28 with the parts related to finding the `to_string` functions left out. The matcher is fairly straightforward except for lines 4 and 5. These are discussed below the listing.

```
1   auto enumFinder = enumDecl(
2           isExpansionInMainFile(),
3           has(enumConstantDecl(hasDeclContext(enumDecl().bind("enumDecl")))),
4           matchers::is_named(),
5           optionally(
6               // Find matching enum_to_string
7           ));
```

Listing 7.28: Matcher for finding enum declarations.

The intention of line 3 in listing 7.28 is simply to bind the enum declaration, similarly to how it was done with other declarations in the previous tools. However, the implementation is different from the other tools, since the binding is required by the matcher that is responsible for finding the matching `to_string` functions. I.e., if the implementation was written as `enumDecl(optionally(/*find to_string*/)).bind("enumDecl)`, the "enumDecl" binding would not be accessible to the "to_string" matcher, as the outer binding does not happen until all the inner matchers are evaluated. Therefore, it was necessary to make the binding earlier, and line 3 is a way of achieving this.

The behaviour of the line is to go to the child in the AST of the enum declaration by entering the enum constant[2], and then going back to the parent, i.e., the enum declaration. This allows the enum declaration to be bound before running the "to_string matcher".

Line 4 in listing 7.28 fixes a bug that was found when running the tool on an external project, i.e., the JSON [22]. Before the line was introduced, the tool was unable to handle unnamed enums [3], which would throw an unhandled exception.

In the AST, an `EnumDecl` inherits from a `NamedDecl`, which contains the `IdentifierInfo` related to the node. An unnamed enum can be interpreted as a `NamedDecl` with no `IdentifierInfo` since it does not have a name. The matcher can therefore be written as `code:085tool:sin_is_named`, where the `IdentifierInfo*` is implicitly converted to a boolean, returning false in it is a nullptr.

```
1   AST_MATCHER(NamedDecl, is_named) {
2       return Node.getIdentifier(); // nullptr if no name
3   }
```

Listing 7.29: Custom matcher for determining if a `NamedDecl` has been given a name.

Finding the outermost namespace

In order to potentially find the `to_string` method matching the `EnumDecl`, one must consider that the function might be placed inside a different namespace than the `EnumDecl`. For that reason, it is necessary to consider all the namespaces inside the compilation unit when searching. The implementation for such a matcher is somewhat similar to the implementation of the built-in `hasDeclContext` that was used in listing 7.28. The behaviour of `hasDeclContext` is to return false, if the declaration does not have a `DeclContext` [4], otherwise return the result of evaluating the `InnerMatcher` on the context casted to a `Decl`.

---

[2]Note that this also filters out empty enum declarations which are uninteresting in terms of having a `to_string` function.

[3]Unnamed enums were mainly used in C++ before `constexpr` was introduced, for defining compile-time evaluated constants for metaprogramming usages.

[4]An example of a case of a `Decl` that does not have `DeclContext` is the outermost `TranslationUnitDecl`.

The behaviour for the recursive version[5] (`has_rec_decl_context`) is similar, but instead of evaluating the immediate parent context, it is evaluated in the outermost context. The implementation can be seen in listing 7.30.

```
AST_MATCHER_P(Decl, has_rec_decl_context, Matcher<Decl>, InnerMatcher) {
    auto cur_ctx = Node.getDeclContext();
    if (!cur_ctx) {
        return false;
    }
    const DeclContext *nxt_ctx = nullptr;
    while (true) {
        nxt_ctx = cur_ctx->getParent();
        if (!nxt_ctx) {
            return InnerMatcher.matches(*Decl::castFromDeclContext(cur_ctx), Finder, Builder);
        }
        cur_ctx = nxt_ctx;
    }
}
```

Listing 7.30: Custom matcher for finding the outermost context of an AST node.

Find matching "to_string"

The remaining part of the `enumFinder` matcher that was seen in listing 7.28 is to potentially identify the `to_string` functions. The implementation can be seen in listing 7.31 with a description below the listing.

```
auto enumFinder = enumDecl(
    /* The rest of enumFinder...*/
    optionally(
      matchers::has_rec_decl_context(hasDescendant(
        functionDecl(
          hasName("to_string"),
          parameterCountIs(1),
          hasParameter(0,
            parmVarDecl(hasType(
              elaboratedType(namesType(
                hasDeclaration(
                  equalsBoundNode("enumDecl")))))
          ).bind("parmVar"))
      ).bind("toString")))));
```

Listing 7.31: The part of the `enumFinder` matcher that was left out of listing 7.28. It is responsible of optionally finding a `to_string` function matching the "enumDecl".

The `optionally` matcher indicates that the inner expressions are not required and `has_rec_decl_context` is

---

[5]Note that while the implementation seen in listing 7.30 has been transformed to its iterative version, the methodology can be considered recursively traversing the AST.

then used to traverse to the outermost context. Curiously enough, unlike `hasDeclContext` , the `hasDescendant` function recursively matches on the descendants. This means that it can be utilized to recursively match the `functionDecl` s. Furthermore, the `functionDecl` must have the name "to_string" and take a single parameter, that is of the same type as "enumDecl". The latter can be expressed by defining that the `parmVarDecl` has a type (`hasType`) that is an `elaboratedType` [6]. The `elaboratedType` must be equal to the enum type, which is accomplished through the `namesType` matcher. However, the "enumDecl" tag refers to a `Decl` instead of a `Type` , so `hasDeclaration` is used to refer to the `Decl` of the `elaboratedType` . At last, it is specified that the `Decl` of must be equivalent to the node bound by "enumDecl".

The combined matcher satisfying the steps defined at the start of section 7.5.2 was now been completed. It should be clear to the reader that writing a single matcher that matches all the necessary nodes of the enum-to-string tool was no trivial achievement.

Finding the namespace in code

When developing the single-step enum-to-string tool, the initial solution also consisted of potentially binding the namespace of the `parmVarDecl` , as it was thought to be needed during node data processing. Later, this approach turned out to be unnecessarily complex during the "node data processing" step, compared to retrieving the namespace through the `SourceLocation` s. However, the resulting matcher turned out to be interesting and will be discussed in the following.

Binding the namespace of the `parmVarDecl` was challenging, as it, similarly to `has_rec_decl_context` , required recursively[7] traversing the AST until arriving at the outermost namespace qualifier of the `parmVarDecl` . Inspiration could be found in the implementation of `specifiesNamespace` that only considers the immediate namespace of the node.

The implementation of the matcher can be seen in listing 7.32. Initially, it is verified that the node can be cast to a `NamespaceDecl` . The function `getPrefix()` can be used to check if there is a `NestedNameSpecifier` prefixing the current node. The function `getAsNamespace` can be used to convert the `NestedNameSpecifier` to its underlying `NamespaceDecl` . Finally, the `InnerMatcher` matches on the outermost `NamespaceDecl` of the `ParmVarDecl` .

```
1   AST_MATCHER_P(NestedNameSpecifier, rec_specifies_namespace, Matcher<NamespaceDecl>, InnerMatcher) {
2       auto ns = Node.getAsNamespace();
3       if (!ns) {
4           return false;
5       }
6       auto prefix = Node.getPrefix();
7       while (prefix && prefix->getPrefix()) {
8           ns = prefix->getAsNamespace();
9           prefix = prefix->getPrefix();
10      }
11      return InnerMatcher.matches(*ns, Finder, Builder);
12  }
```

Listing 7.32: Implementation of the `rec_specifies_namespace` custom matcher.

---

[6]An `elaboratedType` refers to a type that potentially has a qualifier ahead of it.

[7]Once again, the implementation has been transformed into its iterative version.

7.5.3 Node data processing

The logic for the node data processing step of the enum-to-string tool can be seen in listing 7.33. The aforementioned `enumFinder` is used as the *matcher* and the *edits* is a single `changeTo` expression. Inside `changeTo` the `RangeSelector` is determined dynamically for each match, depending on whether "toString" was bound. If it is bound then the `SourceLocation` of the "toString" node is used, otherwise, the `SourceLocation` after the "enumDecl" node is used. The edit is performed as seen in lines 5 to 10. The interesting parts are `print_correct_name` and `NodeOps::case_enum_to_string`, which will be discussed below.

```
1   auto enumRule = makeRule(
2       enumFinder,
3       changeTo(
4           ifBound("toString", node("toString"), after(node("enumDecl"))),
5           cat("\n\nconstexpr std::string_view to_string(",
6               print_correct_name,
7               " e){\n\tswitch(e) {\n",
8               run(NodeOps::case_enum_to_string(print_correct_name, "enumDecl")),
9               "\t}\n}")
10      ));
```

Listing 7.33: Implementation of the `enumRule` that is responsible for node data processing. The `RewriteRule` also adds the necessary headers, which have been left out of the listing to simplify.

Printing the correct name

In order to generate the `to_string` function, the correct notation for referring to the parameter must be used. E.g., if the enum declaration "Animals" is in the namespace "ns" and the `to_string` function is written outside the namespace, then the declaration must be `to_string(ns::Animals e)`. This is the behaviour that the callable `print_correct_name` implements. If "toString" is bound then the notation from the existing `to_string` function must be used, which is implemented in `run(NodeOps::get_declarator_type_text("parmVar"))`. Otherwise, the `to_string` function is placed in the same namespace as the enum declaration, so no prefix is necessary.

```
1   auto print_correct_name = ifBound("toString",
2       run(NodeOps::get_declarator_type_text("parmVar")),
3       transformer::cat(name("enumDecl")));
```

Listing 7.34: Logic behind `print_correct_name`.

The implementation of the stencil `get_declarator_type_text` can be seen in listing 7.35. The stencil operates on `DeclaratorDecl`s, which can be considered an abstraction that incorporates the shared behaviour of specific declaration types, such as `FunctionDecl` and `VarDecl`. It first extracts the `SourceRange` from the node, converts it to a `CharSourceRange` and extracts the corresponding text from the source file, which is finally returned.

```
1   resType get_declarator_type_text(StringRef Id) {
2   return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
3       auto node = Match.Nodes.getNodeAs<DeclaratorDecl>(Id);
4       if (!node) {
5           throw std::invalid_argument("ID not bound or not DeclaratorDecl: " + Id.str());
6       }
7       auto sourceRange = node->getTypeSourceInfo()->getTypeLoc().getSourceRange();
8       auto charRange = CharSourceRange::getTokenRange(sourceRange);
9       auto sourceText = tooling::getText(charRange, *Match.Context).str();
10      return sourceText;
11  };
12  }
```

Listing 7.35: The `get_declarator_type_text` stencil.

Generating enum cases

The final part in terms of node data processing, is to generate the switch case expressions inside the `to_string` function, i.e., generate a case expression for each `EnumConstantDecl` inside the enum declaration.

Given the complexity of the stencil, the group decided to employ the concept of "function composition" to combine several simpler stencils into a more intricate one. This approach allows for the construction of a complex stencil by sequentially applying and composing simpler stencils.

The first stencil that was developed with such an approach, is `foreach_enum_const` and can be seen in listing 7.36. The generic stencil allows for the execution of a callback function for each `EnumConstantDecl` that is defined inside an `EnumDecl` and concatenates the results into a single string. The callback function is required to accept two parameters: A `MatchResult` and an `EnumConstantDecl*` Additionally, the function should return a string as its result. The specific function definition is enforced through metaprogramming techniques, although they are omitted from the example provided in the listing.

```
1   template <typename F>
2   resType foreach_enum_const(StringRef Id, F callback) {
3       return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
4           if (auto enum_decl = Match.Nodes.getNodeAs<EnumDecl>(Id)) {
5               std::stringstream ss;
6               for (const auto enum_const : enum_decl->enumerators()) {
7                   ss << callback(Match, enum_decl, enum_const);
8               }
9               return ss.str();
10          }
11          throw std::invalid_argument("ID not bound or not EnumDecl: " + Id.str());
12      };
13  }
```

Listing 7.36: The `foreach_enum_const` stencil.

A stencil capable of generating case expressions for each `EnumConstantDecl` was now required. This is seen in listing 7.37 where `case_enum_to_string` is implemented. Inside the stencil, a lambda expression is created that is capable of generating a case expression for a single `EnumConstantDecl`. It is then used in conjunction with `foreach_enum_const` to iterate over each `EnumConstantDecl`.

The stencil takes another stencil as a parameter that is used to retrieve the correct identifier to use in the switch case. In practice, this involves using `print_correct_name` as seen in listing 7.34. The stencil then returns a string containing the complete case expression, e.g., `case ns::Animal::Dog: return "Dog";`.

```
1  resType case_enum_to_string(StringRef Id, Stencil getName) {
2      auto lambda = [getName](const MatchFinder::MatchResult &Match, const EnumDecl *enum_decl,
3                     const EnumConstantDecl *enum_const_decl) {
4          auto ns = getName->eval(Match);
5          if (!ns) {
6              throw std::invalid_argument("Could not get potential namespace");
7          }
8          return "\t\tcase " + ns.get() +
9                 "::" + enum_const_decl->getNameAsString() + ": return \"" +
10                 enum_const_decl->getNameAsString() + "\";\n";
11      };
12      return foreach_enum_const(Id, lambda);
13  }
```

Listing 7.37: The `case_enum_to_string` stencil.

7.5.4 Handling the results

7.6 Enum to string – multiple steps

8 Testing

9 Discussion

10 Related work

11 Conclusion

11.1 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).

[21] *Clang-Metatool - A Framework for Reusing Code in Clang Tools*. Bloomberg, May 4, 2023. URL: https://github.com/bloomberg/clangmetatool (visited on 05/04/2023).

[22] Niels Lohmann. *JSON for Modern C++*. Version 3.11.2. Aug. 2022. URL: https://github.com/nlohmann (visited on 05/11/2023).