VRIJE
UNIVERSITEIT
BRUSSEL

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science

Towards Template-Driven Source Code Transformations in C++

Ruben Opdebeeck

Promotor: Prof. dr. Coen De Roover
Advisor: Dr. Tim Molderez

Juni 2017

Abstract

Maintaining a software project often includes performing numerous changes that occur throughout the whole codebase. Changing these occurrences manually is not only time-consuming, it is also error-prone. There already exist a number of transformation tools that can automate these changes in C++, but using them is quite difficult and requires some experience. In this thesis, we introduce a tool prototype that serves as a proof of concept of template-driven source code transformations in C++. It uses a fully template-driven approach to matching and rewriting transformation candidates and allows its source code templates to use metavariables. During matching, metavariables can be used as a wildcard to which arbitrary AST nodes can be bound. During rewriting, the metavariables act as placeholder values which get instantiated with the original source code of the node bound to this metavariable. We look at some of the important design decisions that were made during the development of the prototype, as well as describe the inner workings of the tool. An evaluation of the expressiveness and ease-of-use of the transformations is also given.

Contents

1 Introduction

1.1 Motivation

Although C++ is a rather mature programming language and is seeing a lot of competition in recent years, it is still one of the main programming languages used for professional software development. Due to the longstanding popularity of the language, there currently exist a lot of codebases containing thousands upon thousands of lines of code.

In any development process, the need for refactoring or transforming the codebase arises naturally from time to time. It often occurs that similar changes need to be made in multiple source code locations, such as fixing multiple occurrences of a bug, refactoring source code, or adapting code to a changed API. Additionally, C++ has recently seen some major revisions which each added new functionalities and removed old and outdated language features. Many existing projects can benefit from these new functionalities or even have to change to cope with the removal of some of the language primitives.

Performing such changes manually tends to be tedious and error-prone, and automation is desired. While several code transformation tools exist, they often come with a steep learning curve. A template-based transformation tool lowers this learning curve, as transformations are essentially specified in terms of code snippets.

Furthermore, a company called OM Partners has expressed interest in a template-driven transformation tool for C++. They are interested in transforming their C++ codebases using such a tool and have provided inspiration for one of the transformation examples used in this thesis.

1.2 A Template-Driven Transformation Tool

Our goal is to develop a tool that allows these changes to be made automatically. Moreover, the transformation tool should be template-based, to lower the tool's learning curve. Users should be able to specify transformations without requiring knowledge about the compiler's internal source code representation. To achieve this, transformations can be specified by source code snippets as its left-hand and right-hand sides. The left-hand side snippet specifies which occurrences of source code need to be transformed, while the right-hand side snippet describes how these occurrences should be replaced. Parts of these snippets can be parameterized with a **metavariable**, turning the snippets into source code **templates**. At the left-hand side of the transformation, the metavariable acts as a wildcard, similar to metavariables in logic programming. It will cause the tool to match anything where the metavariable occurs, as long as the surrounding source code matches. The metavariable will then be instantiated and it will be bound to the source code that has been matched, allowing it to be used later on in the right-hand side template. The right-hand side template will be instantiated by replacing all occurrences of metavariables by the source code they were bound to while matching.

We opted for using template-driven program transformations as it is one of the easiest ways to specify a source-to-source transformation. Consider for instance the transformation described in figure 1. It depicts a left-hand side template in the upper-left subfigure, and a right-hand side template in the upper-right subfigure. Metavariables are typeset using bold text. Using source code templates, it is immediately visible to a novice developer what this transformation will match and how it will replace these matches.

Building left-hand side templates is also straightforward: the user can simply take a few transformation candidates and replace all differing parts with metavariables. Because specifying templates is rather easy, building transformations is done quickly, making a template-driven program transformation tool a viable option to integrate into the day-to-day development process.

It also turns refactoring and rejuvenation into a less tedious job, preventing developers from postponing helpful but unpleasant modernizations.

```
int ?functionName(int ?param) {
    int result = ?computation;
    return result;
}
```

(a) An example of a left-hand side template.

```
int ?functionName(int ?param) {
    return ?computation;
}
```

(b) An example of a right-hand side template.

```
int square(int x) {
    int result = x * x;
    return result;
}

int area(int length) {
    int result = square(length);
    return result;
}
```

(c) An example codebase before applying the transformation.

```
int square(int x) {
    return x * x;
}

int area(int length) {
    return square(length);
}
```

(d) An example codebase after applying the transformation.

Figure 1: An example of a template-driven transformation for C++ code. The upper listings depict the left and right-hand side of the transformation. The lower listings show an example of a source file before and after transforming its code.

The main goal of this thesis is to show that template-driven program transformations are possible in C++. We do this by implementing a basic framework that allows us to specify and apply simple template transformations. The main contribution of this research is "Framework X", an initial prototype of such a template-driven transformation tool for the C++ language. The source code for this prototype is digitally available in a git repository.[1]

The rest of this thesis is structured as follows: Section 2 introduces some related works, such as existing transformation frameworks in C++ and the template-driven Ekeko/X tool for Java. As our tool is built on top of the Clang compiler's libraries, a description of its most important concepts is necessary. This is the topic of section 3. Section 4 provides an overview of the architecture of our prototype. Afterwards, sections 5 and 6 describe the implementation of the tool in more detail, handling the left-hand side and right-hand side of transformations, respectively. In these sections, we will look at how templates are represented by the tool, what challenges the tool has to face, important design choices, and algorithms used in the transformation process. Later, section 7 will perform an evaluation of our prototype based on two factors, the expressiveness of its transformations and its ease-of-use. Finally, section 8 lists a number of features that should still be implemented, and section 9 concludes this thesis.

[1] Available at https://gitlab.soft.vub.ac.be/ropdebee/Framework_X.

2 Related Work: Existing Transformation Tools

There already exist a number of program transformation frameworks. Most notably, the Clang frontend for the LLVM compiler project provides us with two transformation strategies. In this section, we will briefly introduce these strategies and discuss why these are not optimal. A more detailed description of these strategies can be found in the research paper preceding this thesis [Opd17]. Afterwards, we will look into an existing template-driven transformation tool for Java, called Ekeko/X [DRI14], from which our tool borrows a number of concepts. Finally, we will discuss how the tool presented in this thesis is influenced by these existing transformation frameworks and tools, which design concepts are borrowed, and where they differ.

2.1 Clang Transformation Framework

One of the key functionalities of Clang is its support for static analysis and as such, it contains a number of useful abstractions to inspect an AST and find AST nodes that match certain condititions. Furthermore, it makes these abstractions available in a public API, so developers can create their own tools based on the Clang framework.

2.1.1 Imperative Transformations: Recursive AST Visitor

The first and most important AST inspection framework of Clang is the **Recursive AST Visitor**, which is a class that allows users of the framework to override certain methods in the class to specify which AST nodes they want to visit. It also allows the user to take control of the way the traversal is performed, such as the order of the traversal, whether or not implicit nodes need to be visited, or even manually descending into children and walking up to the parent node.

This framework can be used to match certain parts of the AST and to rewrite these matches. However, it turns finding transformation candidates into an imperative process and users must write the matching algorithm themselves. As such, this transformation strategy can incur a lot of overhead when creating transformations, and users must have extensive knowledge about Clang and its internal AST representation to create a transformation. Although this strategy is useful to specify recurring changes throughout a codebase, due to this overhead, it is not an attractive option when performing minor changes in a small codebase as it would take more time to write the transformation than it would to apply all changes manually. Hence, this strategy is only viable when there are many transformation candidates, such as for example in a very large codebase.

To briefly illustrate its usage, an example of a recursive AST visitor implementing a transformation is shown in figure 2. Note how the visitor is notified of all function call nodes found in the AST, after which it imperatively refines its search in multiple steps by inspecting the AST node's properties and other nodes it relates to. When it has determined that the node is indeed a transformation candidate, it performs a rewrite. The way it rewrites the source code is not shown in this example.

2.1.2 Declarative Transformations: AST Matching

A number of years after the introduction of the recursive AST visitor, a team of developers at Google set out to migrate their C++ compilation process to use Clang. Among the goals they had in mind, was trying to transform their immense codebase using the Clang static analysis framework. They developed Clang MapReduce [Car11], a parallel version of the Clang project running on their MapReduce [DG08] framework. In order to facilitate source-to-source transformation, they created an **AST matching library**, which was later integrated back into Clang.

```
class LogReplaceASTVisitor : public RecursiveASTVisitor<LogReplaceASTVisitor> {
public:
    bool VisitCallExpr(CallExpr* pCall) {
        const FunctionDecl* pCalleeDecl = pCall->getDirectCallee();

        if (pCalleeDecl != NULL && pCalleeDecl->isCXXClassMember()) {
            const CXXMethodDecl* pMethodDecl = static_cast<const CXXMethodDecl
             ↪  *>(pCalleeDecl);
            const CXXRecordDecl* pClassDecl = pMethodDecl->getParent();

            if (pClassDecl->getNameAsString() == "AdvLogger") {
                const StringRef logLevel = pMethodDecl->getName();
                rewriteLogCall(pCall, logLevel);
            }
        }
        return true;
    }
};
```

Figure 2: An example of a recursive AST visitor implementation.

The main strength of this AST matching library is that it allows a developer to specify *what* they want to match, not *how* the matching process should be performed. In other words, it is a declarative strategy. The developer only has to create a compound AST matcher, which can be built by combining the hundreds of simple AST matchers provided. Some examples of these matchers are listed in table 1. The compound matcher can then be passed to a **match finder** which internally uses the recursive AST visitor to traverse the ASTs of the source files to be transformed. It decides whether or not an AST node is a transformation candidate by applying the matchers on the nodes, using node matchers to verify the node type, narrowing matchers to narrow down the candidate list based on the properties of a node, and traversal matchers to traverse through a candidate to match its descendants or parents. In essence, it abstracts the imperative strategy's search for transformation candidates in an AST.

The match finder will call a user-provided callback on each match, which can be used to transform the source code. Furthermore, matchers allow AST nodes to be bound to identifiers, which can later be retrieved in the callback, providing a way to use the original source code in the replacement. Figure 3 shows a brief example illustrating the declarative matching strategy. It performs the exact same transformation as depicted in figure 2. Note how the developer states what is to be matched by creating an AST matcher and can rewrite the source code without having to do further inspection in the callback.

Hence, the AST matching library significantly simplifies the process of creating transformations, but users still need to be somewhat familiar with Clang to do so. First of all, they need to have a basic understanding of the Clang AST structure in order to create the matchers. Additionally, the matchers can be combined in complicated ways, so chaining these together requires some experience. Lastly, in order to rewrite the source code in the callbacks, knowledge regarding some of Clang's most important facilities, such as the **source manager** and the **rewriter**, is necessary.

In conclusion, although Clang offers us very powerful abstractions to investigate and match an AST, using them requires some expertise with the Clang project and its AST structure. As the vast majority of C++ developers is not familiar with the complexity of Clang, creating transformations imposes a steeper learning curve and makes it difficult to integrate this into their

Node Matchers *Matchers that match a specific type of node.*	
`functionDecl`	Matches a function declaration.
`callExpr`	Matches a call expression.
`cxxRecordDecl`	Matches a C++ class declaration.
Narrowing Matchers *Matchers that match attributes on AST nodes.*	
`anyOf`	Matches if any of its argument matchers also match on this node.
`isOverride`	Matches if a method declaration overrides another method.
`hasOperatorName`	Matches if a unary or binary operator's name equals the argument.
Traversal Matchers *Matchers that allow traversal between AST nodes.*	
`hasDescendant`	Matches if any of the node's descendants matches the argument matcher.
`hasLHS`	Matches if the left-hand side of a binary operator matches the argument matcher.
`hasBody`	Matches if the body of a `for`, `while`, `do while` or function definition matches the argument matcher.
`returns`	Matches the return type of a function declaration.

Table 1: Examples of AST matchers with a brief description of their purpose.

workflow. Hence, an easier method of creating transformations is necessary to attract developers to apply their recurring changes automatically. We need a transformation strategy that abstracts not only the AST matching, but also abstracts the creation of transformations itself so that users should not be aware of how the AST is structured.

2.2 The Ekeko/X Program Transformation Tool

Our prototype is inspired by a template-driven source-to-source transformation tool for Java code, called Ekeko/X. Users of this tool can create transformations by selecting a source code snippet and parameterizing parts of it. It also allows the addition of directives in order to specify constraints on matches that cannot be expressed by simple metavariables. We reuse a number of concepts used in Ekeko/X to form the basis of our C++ template-driven transformation tool.

Figure 4 shows an example of a transformation in Ekeko/X. The `?meta` syntax is used to identify a metavariable `meta`, shown in blue in the figure. The `[node]@[directive]` syntax identifies a directive on the AST node. The figure shows both the left-hand and the right-hand side templates, separated by =>.

Ekeko/X is built as an extension to Ekeko [DRS14], which is an applicative logic meta-programming library that allows querying Java projects against an Eclipse workspace. The tool represents left-hand side templates as a wrapper around AST nodes, containing the metavariables and directives applied to the nodes. It uses Ekeko's logic queries to build a list of potential matches, based on the first node in the template. Afterwards, it traverses through the template and uses the logic queries to match template nodes to nodes in the potential matches. It compares the types of the nodes, their non-node properties such as the value of a literal expression or variable identifiers, and applies directives on the nodes. This way, the template need only be traversed once; at each step, one of its nodes is matched against the corresponding node in every potential match, filtering out mismatches from the potential match list.

```
class ReplaceLogger : public MatchFinder::MatchCallback {
public:
    void run(const MatchFinder::MatchResult& res) {
        const CallExpr      *pCall      = res.Nodes.getStmtAs<CallExpr>("call");
        const CXXMethodDecl *decl       =
         ↪  res.Nodes.getStmtAs<CXXMethodDecl>("decl");
        const StringRef     logLevel    = decl->getName();

        rewriteLogCall(pCall, logLevel);
    }
};

StatementMatcher logMatcher =
    callExpr(callee(
        cxxMethodDecl(ofClass(hasName("AdvLogger")))).bind("decl")
    )).bind("call");

int main(int argc, const char **argv) {
    // Initialization
    // ...

    MatchFinder Finder;
    ReplaceLogger cb;
    Finder.addMatcher(logMatcher, &cb);

    // Invoke the finder
    // ...
}
```

Figure 3: An example of a declarative transformation implementation using AST matchers.

The tool provides powerful directives that can be used to express semantic, control flow and data flow relationships, such as the `refers-to` directive, which expresses that a certain expression node in the AST must refer to a certain variable. It also contains complicated directives used for syntactic relationships, such as `child*`, which specifies that a match must be nested somewhere in an AST node. Directives such as these can significantly increase the complexity of the matching process. Lastly, it supports directives in the right-hand side templates in order to specify how replacements should be applied.

Another interesting property of Ekeko/X is that it does not parse its template source code snippets. When a user selects a source code snippet, it obtains the underlying AST from the Eclipse workspace. Users may then only derive a template from this source snippet on predefined locations such that Ekeko/X can simply wrap directives and metavariables around AST nodes. This way, no complicated parser need be constructed to detect metavariables and directives in plain source code.

2.3 Framework X

We will build our tool on top of the Clang libraries, exploiting existing abstractions, such as the recursive AST visitor, in order to create a template-driven matching algorithm. In order to create an internal left-hand side template, we will use a technique similar to Ekeko/X's AST node wrappers, such that we do not need to parse templates. Some other concepts from Ekeko/X will also be borrowed, such as the potential match list that gets modified while traversing the template, in order to only run through the template once.

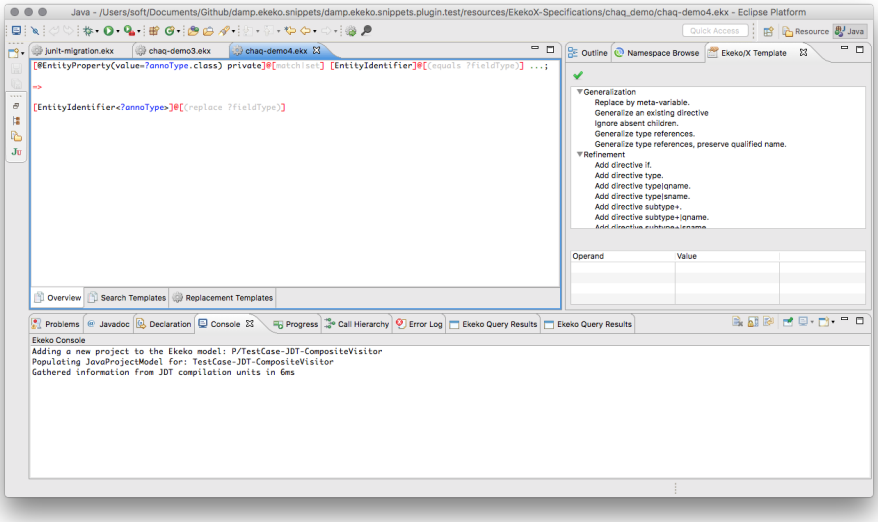


Figure 4: An example of a transformation in Ekeko/X.

Naturally, metavariables will be incorporated into the templates. This will be done similarly to Ekeko/X metavariables, allowing them to be matched to anything if the surrounding context is consistent. Metavariables will be made available in the right-hand side template, allowing the original source code to be used in the replacement code, similar to the AST matching library in Clang. However, the tool will not support directives, as this complicates the transformation in many ways. Most importantly, the matching algorithm will need to keep track of more information, as some of the directives may significantly influence the traversal, such as Ekeko/X's `child*`. As the tool is only a proof of concept, our main interest is verifying that Clang can indeed be used for template-driven program transformation, focusing on representing templates, matching AST nodes and instantiating metavariables, and replacing matched source code with replacement templates.

One of the key differences between Framework X and Ekeko/X is the lack of a logic querying engine. Although the AST matching library's match finder somewhat resembles a logic query engine, mapping a template to matchers is difficult because of the sheer amount of AST matchers and the ways they can be combined. Hence, Framework X will need to implement its own traversal strategy and AST node comparators from scratch. As directives are simply constraints on the logic queries in Ekeko/X, the absence of a query engine is another reason why implementing directives in a template-driven C++ source-to-source transformation framework is difficult.

Finally, although our tool and the declarative AST matching strategy share some similarities, the difference lies in how the matching is performed. The AST matching strategy decides which AST nodes are transformation candidates by applying user-specified matchers on the source ASTs. Our tool does this by comparing a user-specified template AST to the source ASTs. Additionally, the AST matching library does not provide abstractions to describe how these candidates should be replaced.

3 The Clang Libraries

As our tool is built on top of the Clang libraries and makes extensive use of their facilities, a basic introduction to Clang's AST structure and its important modules is necessary. We will first take a brief look at Clang's source manager, preprocessor and lexer. Afterwards, a description of Clang's AST and its generation is given. Finally, Clang's central facilities for AST traversal and source code rewriting will be introduced.

3.1 The Source Manager

Clang's `SourceManager` and its helper classes are one of the main utilities to map AST nodes to the original source code. The source manager keeps a table of `SourceLocations`, which are identifiers that map onto locations in a source file. The source manager provides methods for extracting the line and column number out of these source locations, providing an easy mapping from logical source locations to exact locations in a source file. Furthermore, using the source manager, we can find out in which file a certain source location is written and whether this source location is written directly in this file, or is the result of header inclusion. Another useful helper class is the `SourceRange`. It is simply a pair of source locations indicating the start and the end of a range in the source code. It can be used to represent the source range corresponding to an AST node, or an arbitrary source range in a source file.

Apart from managing source locations, the source manager also keeps a table of file identifiers and separate memory buffers for each file's content. It also keeps track of macro expansions, allowing us to check whether or not an AST node is the result of a macro expansion, retrieving the original macro location and the location of its arguments in case of a functional macro.

3.2 The Preprocessor and the Lexer

The `Preprocessor` is in charge of performing the first phase in the compilation process: Preprocessing. It expands macros, removes comments and whitespace, and includes header files. Afterwards, the `Lexer` reads the individual characters in the source file and turns them into a stream of `Tokens`. These tokens represent simple atomic parts of the source file: symbols such as question marks and brackets; identifiers as a sequence of characters; keywords such as **`if`** and **`void`**; and literal values such as numeric values or strings. Both of these classes communicate extensively with the source manager to create source locations for tokens, notify it of file inclusion and macro expansion, and other important facts.

We can also use the lexer to find, for example, the source location directly following the end of a certain token, or to find a trailing semicolon. A source range for an AST node does not include the potential semicolon following the source code that leads to this AST node. Furthermore, literal values are collapsed to their most compact representation, for example, a boolean literal `true` would be collapsed to simply **`1`**, and a hexadecimal literal **`0x0002`** would be collapsed to **`2`**. As a result, the source range for the AST nodes corresponding to these literals would not include the whole literal as it is written in the source file. To remedy this, we can use the lexer to find the trailing semicolon, or to find the end location of the literal.

3.3 The Abstract Syntax Tree

An abstract syntax tree is without doubt one of the most central concepts in Clang. To generate an AST, Clang first turns the translation unit into a token stream using the lexer. Afterwards, using a combination of syntactic and semantic analysis, it constructs the AST.

The incorporation of semantic analysis in the AST generation phase is due the ambiguity that can arise when purely investigating a stream of tokens. Consider for example the code shown in listing 5. It shows a seemingly simple source code snippet. However, without context, this snippet could mean either of two (or even more) things. Depending on the semantics of `A`, this could be a multiplication of two values if `A` and `B` are variables. However, if `A` is a type, this would be a declaration of a pointer variable `B` that points to a value of type `A`.

This is an example of the context-sensitivity of the C++ (and C) grammar. The lexer has no way of distinguishing between a variable identifier and a type identifier. Usually, compilers deal with this issue by establishing a bidirectional flow of information between the lexer and the parser. In this way, the lexer would send a stream of tokens to the parser, and when the parser encounters the *declaration* of `A` (not shown in the listing) and knows whether it is a type or a variable, it sends this information back to the lexer such that the latter knows what kind of identifier `A` is when it encounters it in the future. However, this is far from an elegant solution, which is why it is commonly referred to as the "Lexer Hack".

Clang uses a much more elegant solution to overcome this issue. It does so by performing semantic analysis while parsing the AST. This way, the semantic analyzer will know exactly what `A` signifies in the source code snippet shown, with the help of its symbol tables. Hence, the parser is able to correctly generate the AST. The full details of how this is managed falls outside of the scope of this introduction, we refer the interested reader to an article regarding this matter, written by E. Bendersky [Ben12].

```
1 A * B;
2 // Could be:
3 // Multiplication of variables A and B
4 // Or:
5 // Declaration of variable B whose type is a pointer to something of type A
```

Figure 5: An example of token ambiguity in C++.

While parsing, Clang will already perform extensive semantic checks and adds this information to the AST. This means that an AST contains a lot more than just syntactic information, it also contains a lot of type information. In order to group all of this information together, an `ASTUnit` is used. It encapsulates a full AST and contains references to the preprocessor, source manager and `ASTContext` for this AST. The AST context acts as a glue between different nodes in the AST. For example, given a node that represents a variable declaration of a certain user-defined type, the AST context can be used to retrieve the original declaration of that type. In essence, it holds long-lived AST nodes and information about them. Additionally, the AST context can be used to retrieve the parent of an AST node. Figure 6 shows a simplified class diagram for the AST unit and its components.

3.3.1 AST Nodes

Clang represents each different kind of AST node in a different class. This leads to tens of different classes and a large and complex class hierarchy. Due to the sheer amount of AST node classes, describing them in detail would be uninteresting, hence a short overview of the most important information is given.

There are four important base node classes, namely statements (`Stmt`), declarations (`Decl`), types (`Type`) and qualified types (`QualType`). This last type of AST node acts as a wrapper around a type, extending it with qualifiers, such as **`const`**. Three other base node classes exist,

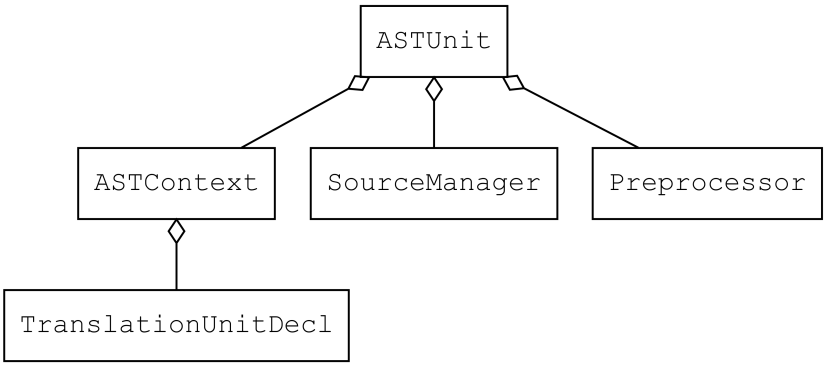


Figure 6: Class diagram depicting the structure of an AST unit.

but their use is limited and falls outside of the scope of this thesis. All other AST node classes inherit from one of these base classes and extends them with new behaviour, custom to the node type. In the end, a concrete node class has a list of children and a number of properties for this node type. Figure 7 shows a partial inheritance diagram for a few of these AST node classes.

Unfortunately, these four base node classes do not share a parent class, i.e. there is no single base class for AST nodes. In order to allow client code to be able to handle multiple types of nodes, such as both statements and declarations, Clang includes a dynamically typed node or `DynTypedNode` that acts as a wrapper around AST nodes. A dynamically typed node can contain any of the seven base classes or one of its derivations, and provides methods to retrieve the underlying node as a given type. It also keeps track of the exact type of node it contains.

Every AST node keeps track of its source range, its concrete node kind and its children. However, as they do not share a base class, they are not polymorphic. For example, all direct children of a statement node can be obtained by calling the `children()` method, however for a declaration node, inner declarations (i.e. child nodes of the AST node) are accessed via the `decls()` method, which is not implemented by every declaration. Each concrete node class also has its own accessors for specific child nodes, such as accessors for each of the operands in a binary operator expression.

It gets worse when dealing with AST nodes that bridge between different base types. For example, a compound statement can contain declarations, such as a local variable in a function body. However, statements can only contain other statements as children, hence, a special bridge node exists, a “declaration statement”, which simply wraps around a declaration node. This node has a child, but as this child is a declaration, it is not contained in the child list obtained by calling `children()` and needs to be accessed via a special method. The same is the case for declarations, where, for example, a function declaration’s body is a statement, or where a variable declaration can have an initializer, which is an expression (a subclass of a statement).

From these examples, we can clearly see that navigating through an AST can quickly become very complicated and that the AST structure makes it difficult to see the forest for the trees. Hence, our tool will need to make use of abstractions to navigate through the AST in a clear manner.

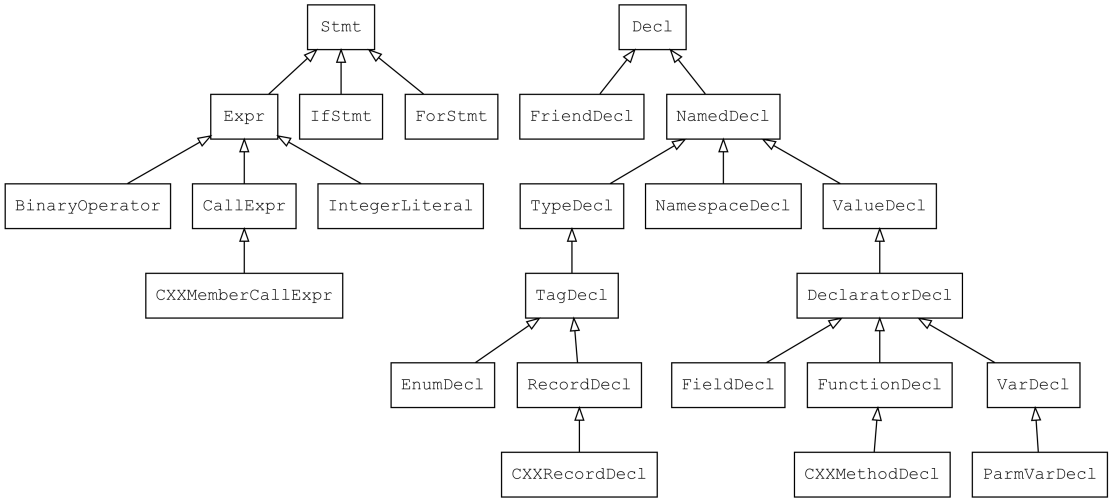


Figure 7: A partial inheritance diagram of AST node classes.

3.4 AST Traversal

As was already mentioned, the `RecursiveASTVisitor` is the main class to be used when traversing an AST.[2] It provides three sets of methods, `Visit*`, `Traverse*` and `WalkUpFrom*`, each with different variants for different AST nodes. The AST node under consideration is passed to these methods as an argument. For example, the `VisitStmt` method will be called for each AST node that derives from a statement, while the `VisitForStmt` will only be called when the recursive AST visitor encounters a for-statement.

As is clear from the names of the methods, `Visit*` is called whenever a certain type of node is visited and `Traverse*` is called to traverse down to a node's children. However, the `WalkUpFrom*` methods do not backtrack to the parent of a node, as the name would suggest[3]. They "walk up" the class hierarchy of the given node recursively, dispatching to `Visit*` methods along the inheritance chain afterwards. This means that `Visit*` methods are called in top-down order, starting from the base class. For example, when encountering a `CallExpr` (a function call), first a `VisitStmt` call will be made, after which the `VisitExpr` and finally the `VisitCallExpr` methods will be called.

To traverse the AST, users need to derive the `RecursiveASTVisitor` class and override the methods for the nodes they are interested in. In the most general case, overriding the appropriate `Visit*` methods will suffice. However, sometimes we want to exert more control over the traversal, such as when we know we do not need to visit certain AST node, for example AST nodes resulting from header inclusion. In order to prevent the AST visitor from traversing down these unwanted subtrees, we can override the `Traverse*` methods. For nodes whose children we want to visit, we can forward the call to the base recursive AST visitor class that we derived from by calling the same method on the base class, which will then take care of traversing the children of the given node. In case we do not want to continue down this subtree, we can simply refrain from making this call, which will prevent further traversal down to the children

[2]Contrary to what the name would suggest, this class does not implement the conventional visitor pattern. Instead of using a double-dispatch mechanism, where an AST node would accept a visitor, notify the visitor of itself and dispatch the traversal to its children; the recursive AST visitor implements the traversal itself through a number of methods in the base class which dispatch to the derived class.

[3]As it is a *recursive* AST visitor, backtracking to a parent is simply returning from a `Traverse*` call.

of the node.

3.5 Source Code Rewriting

Finally, an important part of source-to-source transformations is replacing the original source code ranges with our newly instantiated right-hand side templates. One option to apply replacements is by using Clang's `Rewriter`. It provides methods to delete source code ranges, to insert new source code on a certain location, or to replace ranges with new source code, which is simply a combination of the first two operations.

The rewriter keeps a rewrite buffer for each source file. All source code modifications are applied onto this buffer and are only made permanent when the buffer is written to a file. In order to support multiple consecutive rewrites without breaking the source code, the buffer also keeps a list of deltas which are used to map a location in the original source code to the corresponding location in the rewritten buffer. This means we can safely replace source code in non-overlapping ranges. However, this does not necessarily imply that overlapping source ranges can be correctly edited, as this depends on multiple other factors. For example, it does not make sense to replace text in a source range when that source range was previously deleted. Hence, care must be taken to ensure that the matches we return from the matching algorithm do not overlap in conflicting ways.

4 Overview of Framework X

This section will describe the process of applying a transformation in Framework X on a high level. To do this, we employ the same example as seen in figure 1 in section 1. The example has been repeated in figure 8. Listing 8a shows the source code we want to transform in this example, while listing 8b shows the output we expect after the transformation. In this example, we want to remove a temporary variable in a function definition and perform the computation in the return statement.

In Framework X, a transformation is defined by two source code templates: a left-hand side template declaring what the transformation candidates are, and a right-hand side template describing how these should be rewritten.

Let us first create a left-hand side template for the transformation in the example. A left-hand side template is represented as a partial AST extended with metavariables. The partial AST can contain multiple roots such that it can cover a number of consecutive statements or declarations. Metavariables are used to parameterize a part of the AST that can correspond to any sequence of nodes in the source files, as long as the surrounding context is consistent with the rest of the template.

We will start from the `square` function to define our left-hand side template. However, we also want to transform the `area` function. To do so, our template must be generalized by adding metavariables to it. Inspecting the differences between our two functions, we can see that we need introduce three metavariables to our left-hand side template to make it general enough to cover both functions: the function's name, the name of its parameter and the computation whose result gets assigned to the `result` variable. Hence, these parts must get replaced by metavariables. The result of this is shown in listing 8c. More details on how the templates should be defined can be found in section 5.2.

To find the transformation candidates, Framework X runs the left-hand side template and the input files through its matching algorithm. The matching algorithm will traverse the template AST and compare it to the ASTs of the input files. Its output is a list of transformation candidates, sorted by their order of occurrence in the source files. Each of the candidates has

```
1  int square(int x) {
2      int result = x * x;
3      return result;
4  }
5
6  int area(int length) {
7      int result = square(length);
8      return result;
9  }
```

(a) The source code that needs to be transformed.

```
1  int square(int x) {
2      return x * x;
3  }
4
5  int area(int length) {
6      return square(length);
7  }
```

(b) The intended output of the transformation.

```
int ?functionName(int ?param) {
    int result = ?computation;
    return result;
}
```

(c) The left-hand side template for this transformation.

```
int ?functionName(int ?param) {
    return ?computation;
}
```

(d) The right-hand side template for this transformation.

Figure 8: An example of a template-driven transformation in Framework X.

one or more AST nodes bound to each of the metavariables in the template. It is guaranteed that the candidates do not overlap. This is to prevent the replacements from corrupting the source code. When matches overlap, the largest of the two matches or the one that occurs first in the source file is retained. The transformation candidates are then given to the right-hand side for replacement. A detailed description of the matching algorithm can be found in section 5.3.

A right-hand side template is specified by a combination of literal source code and the metavariables from the left-hand side. For each match obtained by the matching algorithm, the right-hand side template is instantiated by replacing metavariables with the source code represented by the node to which the metavariable is bound. The result of this is a string of replacement source code. The tool will then replace the full source range of the transformation candidate with this replacement text. The right-hand side of the transformation is explained further in section 6.

Let us now create the right-hand side template for our example transformation. As the full function definition will be replaced, we need to provide a right-hand side template that results in source code for this function definition after the transformation. To do this, we simply need to change our left-hand side template to remove the temporary variable. The resulting right-hand side template is shown in listing 8d.

5 Left-Hand Side Templates

Generating and matching left-hand side templates are two of the main components of the tool and as such, it is important to describe how it performs this. First, we will look into the most important challenges the tool must solve in order to achieve its goal. Afterwards, we describe how left-hand side templates are generated, and finally the matching algorithm will be explained.

5.1 Challenges of Left-Hand Side Template Generation and Matching

In order to generate a left-hand side template, we need to map from a flat, textual source code representation onto an AST representation with metavariables. Normally, a parser handles the task of converting textual source code to an AST, however for left-hand side templates, the Clang parser will fail to do so because of a few reasons.

First of all, a template can contain metavariables. If we were to specify these metavariables directly in the source code of the template, we would need specialized syntax used to indicate that it is indeed a metavariable and not some other native language construct. The addition of syntax would mean that a parser would need to be extended to allow and recognize this syntax correctly.

Another problem is that a template essentially is a code snippet (which allows metavariables). While a snippet is valid C++ syntax, it often lacks the context that is required by Clang's parser to perform any semantic analysis. Consider for example the left-hand side template shown in figure 9 and assume the parser is able to correctly recognize the template metavariable. This is a perfectly legal left-hand side template, but because of the lack of context, the parser will not be able to resolve the identifiers `foo` and `bar`. As we saw in section 3.3, it needs to be able to identify these in order to generate a correct AST. As a result, the parser will raise an error and will fail to parse the template.

```
if (?conditionVar) {
    foo(bar);
}
```

Figure 9: An example of a left-hand side template.

Finally, figure 9 shows another issue. In order for C++ code to be valid, a translation unit may only contain top-level declarations. In other words, writing statements outside of function definitions is not allowed. If we were to feed this template into the parser, it would refuse to generate an AST as it expects a top-level declaration but receives a statement instead.

To generate a left-hand side template, we need to find a way to overcome these challenges instead of limiting how a template can be defined. One option to allow such templates is by adapting Clang's parser such that it is capable of interpreting them. However, Clang's parser is very large, consisting of thousands of lines of code, so changing its behaviour correctly is quite a difficult task. Hence, we opted to start from a legal source file, one that Clang can parse correctly. We allow the user to specify which part of the source code represents the template, and the parts of this source code that need to be parameterized with a metavariable. Afterwards, we decorate the original source file's AST with the necessary information. Subsection 5.2 describes how this is achieved in more detail.

Matching of ASTs also imposes some challenges. As we already mentioned, AST nodes do not share a base class. Hence, in order to support multiple types of AST nodes, such as for example both statements and declarations, we need to use special constructions such as dynamically typed nodes. Furthermore, because of the complexity of the AST structure, some abstractions are necessary to facilitate traversal. Finally, as Clang is not designed to compare ASTs, we will need to implement our own comparators to match AST nodes. How the tool handles these challenges is explained in subsection 5.3.

5.2 Generating Left-Hand Side Templates

Left-hand side templates are defined by a combination of a normal C++ source file and a configuration file. The source file acts as a basis for the eventual left-hand side template, while the configuration file contains information regarding the metavariables and other options of the transformation. The tool will generate an AST from the source file, which prevents it from running into the challenges laid out in the previous subsection. Afterwards, it will use the configuration file to decorate the obtained AST with metavariables and other important information.

5.2.1 The Configuration File

We chose to represent the configuration of a transformation as a JSON file, as this allows the user to clearly but compactly define how a left-hand side template should be constructed using key-value pairs. In the current version of the tool, the only way to specify how the template should be constructed is by creating this configuration file manually. Figure 10 shows an example of such a configuration file. It corresponds to the transformation illustrated in section 4.

The configuration file contains the following mandatory information:

- The source file to generate the template AST from;
- The range in this source file which specifies the template's start and end.
- A (potentially empty) list of metavariables in the template, with each entry consisting of:
  - The identifier of this metavariable;
  - The range in the source file that specifies which piece of code should be replaced by a metavariable;
  - An optional flag indicating whether the full AST node residing behind this source range should be parameterized, or only the name of a declaration of a variable, function, class, or other named entities. This allows us to for example parameterize the name of a class but still require that its members are matched between the template and the potential matches in the source code to be transformed.
- The right-hand side template file (see section 6).

Additionally, the configuration file may define flags specifying whether the original source files need to be overwritten or that the changes need to be written to a new file; and whether or not the template source file should be included as one of the files to be transformed. The former option is useful when testing transformations or when the user is unsure whether or not the changes should be kept. The latter is especially useful when creating general transformations that are applicable to many codebases. This way, when it is unknown on which source files the tool will be applied, a minimal source file could be distributed which the tool could use to parse an AST and generate the left-hand side template from.

The ranges in the configuration file are specified as a pair containing the begin and end location of the range. Locations are itself specified as a pair of line and column numbers. This facilitates defining template and metavariable ranges as users can simply look up the required numbers in their IDE or their text editor, which would not be the case if we used file offsets or source code lengths.

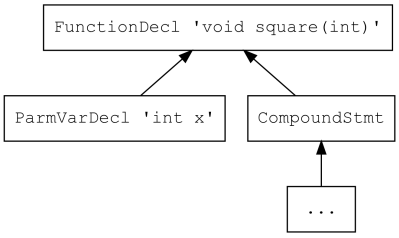
The template range indicates the start and end of the template in the original source code. A metavariable range indicates the start and end of an AST node or multiple AST nodes that should be parameterized in the template, i.e. the nodes will not be compared to the source

```
{
    "templateSource": "Example.cpp", // Example source code (figure 8a) is written
    ↪  in Example.cpp
    "templateRange": [[1, 1], [4, 2]],
    "metaVariables": [{
                      "identifier": "functionName",
                      "range": [[1, 1], [4, 2]],
                      "nameOnly": true
                      },
                      {
                      "identifier": "param",
                      "range": [[1, 12], [1, 16]],
                      "nameOnly": true
                      },
                      {
                      "identifier": "computation",
                      "range": [[2, 18], [2, 23]]
                      }],
    "rhsTemplate": "replacement.tmpl", // Replacement template (figure 8d) is
    ↪  written in replacement.tmpl
}
```

Figure 10: A configuration file corresponding to the transformation defined in figure 8.

code to be transformed and the metavariable will simply be instantiated with any node whose surroundings match the rest of the template.

As Clang's AST represents identifiers as a property of an AST node rather than as an AST node itself, the name-only metavariables are necessary to parameterize the names of classes, functions or variables when we still want to match the children and properties of their nodes. Consider for example again the source code shown in listing 8a. The corresponding AST subtree is shown in figure 11a. If the metavariable parameterizing the function parameter `x` was not a name-only metavariable, the whole AST node for the function parameter would be replaced by a metavariable instead of only its name. This would result in the left-hand side template depicted in listing 11b. This template is much more general than the one shown in figure 8c and as such, could result in incorrect matches.



(a) The AST subtree for the `square` function depicted in listing 8a.

```
void ?functionName(?param) {
    int result = ?computation;
    return result;
}
```

(b) The left-hand side template for the example if **`?param`** is not marked as name-only.

Figure 11: An example showing the importance of name-only metavariables.

Because an identifier is a property of an AST node, we cannot parameterize it using the source range of the identifier and must use the source range of the whole AST node instead, while marking the corresponding metavariable as a name-only parameterization. For example

for a class, this source range would be the start and end of the whole class definition, i.e. from the position where the **`class`** keyword is written, up until the position of the brace ending the class definition. However, while matching, the class node will still be matched as if it was not parameterized and only differences in the name of the class will be ignored.

A JSON schema is used in order to verify that the configuration file is formatted correctly. Such a schema allows us to define which parts of the configuration file are mandatory as well as the allowed types for the various entries in it. We can also specify the allowed length of an array in a JSON file, which is especially useful for ranges and locations, as both of these must consist of exactly two elements.

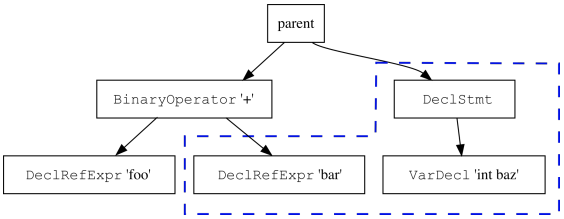
5.2.2 Mapping Configuration Ranges to the Template Source AST

In order to create the left-hand side templates, the information provided in the configuration file is used to find the template AST and the metavariables in the AST of the template source file. This is done by mapping source ranges provided by the user to nodes in the source AST. As such, the ranges in the configuration file must pass a number of requirements:

1. The template range must fall within the bounds of the source file, and every metavariable range must fall within this template range.
2. Metavariable ranges may not overlap, unless the metavariable range falls within the range of a name-only metavariable.
3. Each range, both the full template range or a metavariable range, must map to either:
   (a) A single statement or declaration node;
   (b) Multiple statement or declaration nodes, provided they are siblings, i.e. they share the same direct parent.

   Note that for metavariables, these nodes may be any nodes in the template AST, both one of the AST roots or descendants of the roots.

Another way to formulate rule 3b is that a range may not partially span an AST subtree. Consider for example the AST in figure 12a. It shows a range, depicted as a blue dashed frame, that partially spans a subtree. The corresponding source code snippet is shown in listing 12b. The source code corresponding to the range is depicted in bold text. As we can see from the figure, this range makes little sense because it only partially covers the addition operation on the first line. Hence, ranges that partially cover an AST subtree are disallowed.



(a) A representation of an example AST with a range partially covering a subtree.

```
foo + bar;
int baz;
```

(b) The source code of the example AST with the portion covered by the range in bold text.

Figure 12: An illustration of why metavariable ranges may not span multiple subtrees.

In order to map the ranges in the configuration to the AST nodes in the template source file, we use a recursive AST visitor that traverses the AST of this file. Our visitor gets called on each statement or declaration node in the AST and performs its task in two stages. During the first phase, it compares the source ranges of these nodes to the template range while assuring that no partially spanned subtrees are added to the template. In case such a subtree is encountered, it simply throws an exception and stops the traversal, as this means that the configuration file is malformed. When it encounters a node that is part of the template, it adds this node to the template and to an internal queue containing subtrees that should later be processed to find the metavariables.

The second phase consists of traversing the subtrees contained in this queue and looking for the AST nodes corresponding to the metavariable ranges. When such a node is found, it is again added to the template. Similar to the first phase, it throws an exception when one of the metavariable ranges partially spans an AST subtree.

Our visitor implements its own traversal strategy and descends only into a subtree if the source range of this subtree contains a range that is being looked for. There is no need to descend into subtrees that do not overlap with this range, their children cannot possibly overlap with it either. Additionally, once we find that a node is part of a template or a metavariable, there is no need to descend into its children, as metavariable ranges cannot overlap. An exception to this rule is when the metavariable is a name-only metavariable, in which case we still need to descend. The result of this custom traversal is that only the relevant AST subtrees are traversed, which prevents the tool from wasting time searching in AST nodes far away from the relevant ones.

One difficulty in this algorithm is the fact that we allow ranges to cover multiple nodes. The recursive AST visitor provides little context besides the node that is currently being visited: it does not keep track of its children, parents or siblings. However, we need this information to assure that the ranges do not partially cover AST subtrees, and to determine which nodes are all part of a single source range. Hence, we need to keep track of this information ourselves.

Another issue lies in the fact that trailing semicolon locations are not part of the source range of a node, and that literal values are collapsed to their most compact representation and as a result, its source range is inconsistent. As we have seen in section 3.2, Clang's lexer allows us to find the real end locations in cases such as these. As a result, when comparing source ranges, we also need to compare these extended source ranges to the template or metavariable range under consideration.

5.2.3 Left-Hand Side Template Representation

Finally, we need a representation of the left-hand side template. Our template representation keeps a list of AST nodes, making up the root list of the template. These are exactly the AST nodes that were found during the first phase described in the previous section.

The template representation also needs to keep track of which AST nodes in the template are parameterized by metavariables. Unfortunately, Clang limits our ability to dynamically create new AST nodes or to change existing ones. Node constructors are often locked away behind private or protected access modifiers and are only accessible using custom allocation techniques, and many properties of AST nodes are marked constant and can only be set using these custom allocators. Because of this, we cannot extend the nodes with metavariable information or change one of the children of a node to a metavariable node. As a solution to this problem, our representation also contains a map from AST nodes to metavariables. This map contains all AST nodes that have been found to be parameterized during the second stage described earlier. As a result, when traversing the template during the matching phase, we need only to look up the

AST node in this map to determine whether or not it is parameterized by a metavariable.

When adding metavariables to the template, an interesting special case is encountered when a name-only metavariable covers a class declaration. As was already mentioned, this parameterizes the name of the class, but the matching algorithm will still need to match all of its children. However, the class' constructors will carry the same name as the class itself and as such, its names will need to be parameterized as well. To achieve this, the left-hand side template will add implicit name-only metavariables linked to the nodes of these constructors.

5.3 Matching Left-Hand Side Templates

After generating the left-hand side template, the tool's next task is to match it to the given source files to be transformed. In short, the way this is performed is by first gathering all potential matches from the ASTs from these source files. Afterwards, the template AST is traversed and at each step of the traversal, the current node in the template is compared to the corresponding node in each of the potential matches. Potential matches that cause a mismatch at this step are removed and will not be considered in the future. Once we have finished traversing the entire template AST, all matches in the potential match list are actual matches to the template. Finally, the tool cleans up the potential match list by filtering out overlapping matches and sorting them. Algorithm 1 depicts the matching algorithm in pseudocode. In the next subsections, we will look into each of these steps in more detail.

5.3.1 Gathering the Potential Matches

In order to populate the initial list of potential matches, the tool uses a recursive AST visitor which visits each statement and declaration, and adds the node to the potential match list if its node kind matches the node kind of the root of the template. It performs no other checks: it does not compare the children, as the template traversal will already handle this, nor does it compare specific properties of the nodes, as this will be done while matching the template to each of the potential matches.

For example, if the first node in the template is a function declaration, the initial potential match list will contain all function declarations throughout the codebase that needs to be transformed as these can all potentially match the template. Determining whether or not they actually match is performed by comparing the nodes in the template to the corresponding nodes in these potential matches, which is done in the next step of the matching algorithm.

5.3.2 Abstracting the AST Traversal

In order to facilitate the matching, the tool uses a simple, custom AST representation to traverse the AST. As was mentioned in section 3.3, Clang's AST structure is quite complex and accessing children differs for some nodes. Hence, the tool uses its own custom AST node wrappers which encapsulate the real nodes and generalize the process of accessing children. This wrapper class uses lazy initialization for the child list: the first time the children are accessed, it retrieves the child list of the underlying node and saves it. In other words, it only creates the child list once it is really needed, preventing expensive operations for nodes which are not visited by the matching algorithm because earlier nodes in the potential match caused a mismatch.

Such AST nodes can also be virtual, which indicates that they do not directly represent a real underlying AST node. This is for example useful when grouping together multiple template roots if the template covers multiple AST subtrees, or for example in case of a function declaration, which has a variable length list of parameters and a function body. Such function declarations are represented by our custom AST nodes as a node with two children. The first of these children

Algorithm 1 Framework X's Left-Hand Side Matching Algorithm

Input: *ASTs* ▷ a list of ASTs
Input: *template* ▷ a left-hand side template
Output: *matches* ▷ a list of transformation candidates

```
1: potentialMatches ← all AST nodes whose class equals that of the template's root
2: while template is not fully traversed do
3:     current ← current node in template
4:     if first time entering current then
5:         if current is non-name-only metavariable then
6:             Create new potential matches for all possible bindings for the metavariable
7:         else if current is name-only metavariable then
8:             for all p in potentialMatches do
9:                 if p and current differ in more than their name then
10:                    Remove p from potentialMatches
11:                else bind the current node in p to the metavariable
12:                end if
13:            end for
14:        else ▷ current is not a metavariable
15:            Filter out any potential match whose current node does not match with current
16:        end if
17:        if current has children then descend to them
18:        else if current has next sibling then traverse to the next sibling
19:        else backtrack to parent
20:        end if
21:        Filter out potential matches whose traversal state does not equal the template's
22:        Perform the same traversal step in each of the potential matches
23:    else ▷ We have visited this node before, so we backtracked from a child
24:        If current has a next sibling, traverse to it, otherwise backtrack to parent
25:        Filter out potential matches whose traversal state does not equal the template's
26:        Perform the same traversal step in each of the potential matches
27:    end if
28: end while
29: return potentialMatches
```

is a virtual AST node, containing each of the function parameter declarations as its own children, while the last child of the function declaration is the function body.

As a second abstraction, the tool uses an AST traversal state class in order to coordinate the traversal between the template and the potential matches and to control the traversals themselves. This class encapsulates the custom AST node for the AST being traversed, and contains at any point a "current node". Each of its operations are applied with this node in mind. The class provides methods to consult the current state of the traversal, such as checking whether or not the current node's children have been accessed in the past, whether or not the node has children, checking if the node is the last of the siblings, and so on. It also provides methods to influence the traversal, mainly descending to the children of the current node, continuing to the next sibling of the node, and backtracking to the current node's parent.

5.3.3 Matching the AST Structure

The tool traverses the template only once. It does this iteratively with the help of the AST traversal state class. Both the template and each of the potential matches are represented by such an AST traversal state, meaning we can navigate through their ASTs in an abstract way. In essence, this traversal is an iterative pre-order depth-first tree traversal.

There are two possible cases at every step in the traversal, depending on whether or not the current node in the template has been visited before. If it has not been visited before, the traversal is currently moving downwards or to the right. If it has been visited before, it means that we backtracked from a child in the previous traversal step.

Downwards Traversal The first case we consider is when the traversal enters a previously unvisited node in the template, thus travelling downwards or to the right in the tree. Since the current template node is a new node, it still needs processing. Depending on whether or not the current node is a metavariable, different actions need to be performed:

- If the current template node is not a metavariable, it needs to be compared to the current node in each of the potential matches. If this comparison determines that the current node of a potential match is not equivalent to the template node, this potential match is a mismatch and should not be considered anymore in the future.
- Otherwise, if the current template node is a metavariable that performs a name-only parameterization, we still need to compare this node to each of the potential matches in a similar way as the previous case. However, while doing this comparison, differences in the names of the template node and the potential match node should be ignored. If they are still not equivalent, the potential match is again a mismatch, otherwise, the metavariable should be bound to the current node in the potential match.
- Finally, if the current template node is a non-name-only metavariable, no comparison should be performed and in each of the potential matches, the metavariable should be bound to the current node in this potential match.

Afterwards, the matching algorithm should proceed to the next template node. Which node is the next node depends on the structure of the AST subtree at the current template node. As the matching algorithm implements a pre-order depth-first traversal, it descends into the children of the node if there are any, proceeds to the next sibling if there is any, or backtracks to the parent node in any other case. Note that if the current node is parameterized by a non-name-only metavariable, descending into the children should not be done, as the children should not be matched any further. Hence, in this case, traversal proceeds to the next sibling or backtracks to the parent.

In any case, the same traversal step should be made in each of the potential matches. For example, if the matching algorithm proceeds to the first child of the current node in the template, it should do the same in each of the potential matches. However, before doing this, the matching algorithm must make sure that the traversal state in each of the potential matches is consistent with the traversal state in the template. For example, if the current node in a potential match has children, but the current template node does not, the potential match is also a mismatch and should be ignored in the future.

Backtracking In the other case, when we already visited the current template node, it means that we have backtracked in the previous traversal step. Hence, this node has already been

processed in the past and should not be compared anymore. We should simply proceed to the next sibling if there is any sibling left, or backtrack to our own parent if we are its last child.

As in the downwards traversal, we need to make sure that the traversal state is consistent between the template and each of the potential matches. If the current node of any of the potential matches still has a sibling while the current template node is the last sibling, the potential match is again a mismatch. Afterwards, the traversal in the potential matches should perform the same step as is done in the template.

Remarks on Matching As was mentioned before, the matching algorithm only traverses the template once, which is made possible by gathering all potential matches into a single list and narrowing down this list while traversing through the template. This leads to an efficient matching strategy, contrary to a naive solution which would traverse the entire template again for each potential match.

However, this strategy has an additional benefit. We allow a metavariable to match multiple consecutive sibling AST nodes in the source file. For example, figure 13 shows an example of why matching multiple AST siblings to a single metavariable should be allowed. The downside to allowing this is that it makes instantiating metavariables in the matching algorithm more difficult. When we encounter a metavariable during matching, and that metavariable could potentially match multiple sibling nodes, we need to know how many of these sibling nodes we should assign to this metavariable and at which point the traversal should continue. To determine this, we would need to be able to look ahead in the template, which would significantly complicate the matching algorithm.

Fortunately, since we use a potential match list, we can use a more brute-force way to instantiate this metavariable. We can simply create new potential matches for every possible instantiation of the metavariable and add all of these to the potential match list. The future checks in the traversal will then remove any inconsistent instantiations and in the end, only correct ones are kept. This means that we do not have to look ahead in the template, which retains our simple pre-order depth-first traversal through the tree.

```
bool foo() {
    ?stmts
    return true;
}
```

```
bool foo() {
    1 + 2;
    return true;
}
```

```
bool foo() {
    string bar;
    int baz;
    return true;
}
```

Figure 13: An example of a metavariable matching multiple AST nodes. The listing on the left shows the an example template in which the metavariable **?stmts** can match multiple sibling nodes. The other listings show matches of this template, with the instantiation of **?stmts** shown in bold.

5.3.4 Comparing AST Nodes

As Clang does not provide us with a way to compare AST nodes by value, we need to create comparators ourselves. Fortunately, for most of the AST node classes, it suffices to verify that both AST nodes are of the same kind.

However, not every AST node defines all their characteristics as children. For example, nodes representing a literal value, such as an integer literal or a string literal, need to have these values compared as well. For declarations, depending on the kind of declaration, we may need to

compare the names of the declarations, the types defined or used in this declaration, and other specific properties such as access specifiers and linkages.

This results in quite a lot of comparators which all share the same basic format. In order to reduce the amount of code being repeated, a number of functional macros have been written that allow us to easily specify comparators with a low amount of code redudancy.

5.3.5 Sorting Results and Filtering Out Overlapping Matches

As was briefly explained in section 3.5, in order to prevent the source code from being corrupted after rewriting, we need to assure our matches do not overlap in conflicting ways. To remedy this, we simply do not allow matches to overlap in any way, which assures us that no conflicting rewrites take place. However, this may limit the amount of matches returned from the tool. Fortunately, this can often be fixed by simply running the tool again over the previously transformed source file.

In order to filter out the overlapping matches, we inspect their source ranges and remove any match which falls inside of another match's source range. This way, only the largest matches are kept in the match list. During this process, the matches are also grouped by which file they belong to and sorted in order of appearance in the source file.

Although this technique is successful in eliminating conflicting matches, it may be somewhat too restrictive, as it may also eliminate overlapping matches which would not corrupt the source code. However, determining when two overlapping matches would conflict requires a lot of analysis and as Framework X is only a proof of concept, we decided not to implement this analysis and simply eliminate overlapping matches in the naive way described above.

5.3.6 Closing Remarks

Finally, we list some closing remarks on the left-hand side matching algorithm.

First of all, we do not use the recursive AST visitor to traverse the AST since it cannot be used to traverse multiple ASTs at the same time, which is being done in our matching algorithm. Furthermore, it does not provide us some important information regarding the traversal, such as whether or not a certain node is the last child of its parent. This information is vital to assure the traversal states are consistent between the template and its potential matches.

Secondly, since a metavariable can match multiple nodes, a special case can be encountered when two metavariables immediately follow each other, such as in the template depicted in figure 14. Because there is no other AST node in the template that separates the two metavariables, there is no way for the tool to know which of the statements in a function body should belong to the first metavariable, and which to the second. As such, although this template will match a function that has at least two statements in its body (metavariables cannot be bound to nothing), the tool's behaviour is undefined on such a template. It has no way of knowing what the user expects the metavariables to be bound to, and therefore cannot make guarantees that they will be bound correctly.

6 Right-Hand Side Templates

The right-hand side of a transformation specifies how the matches obtained by the left-hand side should be replaced in the source code. Since we are developing a fully template-driven tool, our right-hand side consists of source code templates as well. We will first look at how right-hand side templates are defined. Afterwards, we describe how they are parsed and represented in

```
void foo() {
    ?firstStmts
    ?lastStmts
}
```

Figure 14: A template containing a special case of two metavariables directly following each other.

the program. Finally, we explain how right-hand side templates are instantiated and how the replacements are applied.

6.1 The Structure of a Right-Hand Side Template

A right-hand side template is specified in its own file. As a convention, this file has the `.tmpl` extension, however this is not a hard requirement, as the tool allows any file with any extension to serve as the template definition. Users can specify which right-hand side template file to use in the transformation configuration file, as explained in section 5.2.1.

The contents of the right-hand side template file are used as the replacement text in the transformation. As such, it is recommended that the template consists of legal C++ code. However, the tool currently does not verify that the replacement text is indeed legal C++ code due to the way templates are parsed and represented. As a result, the user has a lot of freedom in designing transformations. In the future, such verifications should be added, or at least some restrictions on the right-hand side template should be placed.

A right-hand side template can contain metavariables as well. The syntax used for these metavariables is the same as the syntax we used in previous examples of the left-hand side template. For example, **`?meta`** is a valid metavariable. Since a question mark is legal syntax in C++, used as part of the ternary conditional operator `?:`, care must be taken not to confuse both uses of the symbol. Hence, we require that metavariables contain no spacing between the question mark and the identifier. This way, a ternary conditional operator can be used in the right-hand side template, provided that its question mark and the second operand *do* have a whitespace separating them.

Since we do not restrict the template to be only legal C++ code, a metavariable may occur anywhere in the replacement text. When instantiating the template, its occurrence will simply be replaced with the source code represented by the AST node(s) bound to the metavariable in a match. In case of a name-only metavariable, the metavariable will be instantiated with the name of the matched AST node. When all metavariables are instantiated, the source code of the full match will be substituted with the replacement text.

6.2 Representation of the Right-Hand Side Template

Since we do not need to perform complex traversals or comparisons at the right-hand side of the transformation, a very simple representation of right-hand side templates can be used. The tool represents a right-hand side template as a list of template parts, which can either be literals or metavariables. Literal template parts represent the literal pieces of source code written in the template file, and metavariable parts contain the identifier of the metavariable whose source text should be used in its place. Although a very simple representation, it allows us to easily parse and instantiate templates. Parsing can be done by simply splitting the template at the right

places (section 6.3), while instantiation can be performed by concatenating the template parts (section 6.4).

6.3 Parsing Right-Hand Side Templates

In order to split the template into literal parts and metavariables, we use Clang's lexer to turn the template file into a token stream. This way, metavariables can be detected by inspecting a sequence of tokens: when a question mark token followed by an identifier token is found, and there is no whitespace separating them, these two tokens identify a metavariable. We can retrieve the identifier of the metavariable from the contents of the identifier token.

We iterate through the token stream while keeping track of the source range of the literal part. Each time a token is encountered that is not part of a metavariable, the literal range is extended until the end of this token. When we encounter a metavariable, the literal range is read from the source file as a string and its text and the metavariable are added to the template parts list. When the full token stream is processed, the final literal range is read as well, and added to the list.

6.4 Instantiating Right-Hand Side Templates and Applying Replacements

Instantiating the right-hand side template is done by concatenating the literal parts with the source text of the AST nodes bound to a metavariable. In case of a name-only metavariable, only the name of the declaration node is retrieved and used as an instance value. Otherwise, the source range of the node(s) bound to a metavariable is read and used as an instance value. As multiple sibling nodes can be bound to a metavariable, the source range must be computed to be the range covering all of these siblings. When reading the source range, care must again be taken to include trailing semicolons and full literals.

There are a two options when it comes to applying replacements. One option is to use the rewriter introduced in section 3.5. To apply replacements, we use its text replacement method, replacing the source range of the full match with the instantiated replacement text. Another option is to use Clang's `Replacements` class. This class internally uses the rewriter, but its main benefit is that it keeps a set of replacements and verifies that there are no conflicts before applying them. As we already filter out conflicting matches, there is no added benefit to using this class over the rewriter. Furthermore, since version 4 of Clang, this class has received a poorly-documented update and it is unclear how it should be used. Therefore, the tool uses the rewriter to perform its replacements.

7 Evaluation of Framework X

Now that we have described how the tool transforms source code, we must evaluate how well it performs this task. There are a few ways to measure this and the ones we will focus on are the expressiveness and ease-of-use of the transformations. For expressiveness, we will implement a few example transformations and focus on what can be parameterized in the left-hand side template, what the tool can successfully match, and if the right-hand side template succeeds in correctly replacing the matches.

7.1 Expressiveness of Transformations

In order to evaluate the expressiveness of transformations, five examples of transformations have been implemented. We will analyse each transformation individually, describing their purpose and discussing the three focus points mentioned earlier. For cases where the tool fails to correctly apply a transformation, we will investigate why it does this and propose a way to resolve this. As we will see, a lot of these issues can be remedied by the introduction of **directives**, which are used to specify information that cannot be represented by literal source code and metavariables alone. Examples of such information are additional constraints on AST nodes or changing the default behaviour of the matching algorithm.

7.1.1 Example 1: Rearranging Class Members

In C++, class members are by default private, unless specified otherwise. Hence, a lot of class definitions list the private members as their first members, which pushes the publicly accessible members to the bottom of the definition. However, when designing interfaces, this is a bad practice, as it is more convenient for the users of the interface to have the public members listed at the top of the class definition such that they can easily be found.

Our first transformation example will rearrange the public and private members of a class to place the public members at the top of the class definition. Figure 15 shows the left and right-hand side templates for this transformation, along with an example source code snippet before and after applying it. Note that the **`?classname`** metavariable is a name-only metavariable.

```
class ?classname {
    ?private_members
public:
    ?public_members
};
```

(a) The left-hand side template.

```
class ?classname {
public:
    ?public_members
private:
    ?private_members
};
```

(b) The right-hand side replacement template.

```
class A {
    int _b;
    int _c;
public:
    A(int b, int c) : _b(b), _c(c) {}
    int multiply() { return b * c; }
};
```

(c) Example code before transformation.

```
class A {
public:
    A(int b, int c) : _b(b), _c(c) {}
    int multiply() { return b * c; }
private:
    int _b;
    int _c;
};
```

(d) Example code after transformation.

Figure 15: Templates and example code to rearrange class members.

When we run this transformation on the example, it gives the output as expected. However, if the class had ordered its members in any other way than is expected in the left-hand side template, such as for example the class depicted in listing 16, the transformation would fail to match the left-hand side template. The reason for this is that the AST structure is different to

the AST structure of the left-hand side template because of the extra `private` access specifier at the bottom of the class definition.

```
class B {
private:
    // Private fields
public:
    // Public interface
private:
    // Private helper methods
};
```

Figure 16: Source code example of a class which the first transformation example fails to transform.

We can see from the transformation example that we are able to parameterize class members and class names correctly. The left-hand side matching algorithm also correctly matches the template to the source file, except for when the class organizes its members in a way different to what the template was designed to match. However, this is correct behaviour, as the matching algorithm should not be able to correctly match this case. To resolve this, we could introduce directives to match any public or private member, such that these exceptional cases can be handled correctly.

7.1.2 Example 2: Redundant Void Parameters

In C, when a function accepts no parameters, its parameter list must consist of a single `void` keyword. In C++, this requirement is not necessary, hence the parameter list may simply be empty. However, it also allows an empty parameter list to be denoted in C-style. The use of a `void` parameter is nonetheless discouraged as it is redundant.

Experienced C programmers who start programming in C++ often make the mistake of including such a `void` parameter. Additionally, when migrating C code to C++, such function declarations are often left unchanged. Hence, we create a transformation to automatically remove the redundant `void` parameter. Figure 17 again shows the left and right-hand side templates for this transformation, and example source code before and after applying the transformation. The **`?functionName`** metavariable is a name-only metavariable matching the name of a function.

We can see from this example that we can correctly parameterize the name of a function, and that this is also correctly instantiated in the replacement. The same is the case for the function body. However, there are some limitations to our transformation that can be seen in this example.

Type Information First of all, the template cannot parameterize the return type of a function, hence in this case, it will only pick up functions that do not return any value. The reason for this is that we can currently only parameterize declarations and statements, not types.

Methods and Functions Secondly, the source code example contains a function in the `Baz` class which declares a `void` parameter list. However, this function is not transformed. This is because it is actually a method, not just a function. Functions and methods are represented by different AST node classes and as such, our matching algorithm does not consider them a match. Whether or not this behaviour is correct relies on the intent of the creator of the transformation.

```
void ?functionName(void) {
    ?functionBody
}
```

(a) The left-hand side template.

```
// Void params are redundant!
void ?functionName() {
    ?functionBody
}
```

(b) The right-hand side replacement template.

```
1  void foo(void) {
2      1 + 2;
3      cout << "Foo called\n";
4  }
5
6  void bar() {
7      cout << "Bar called\n";
8  }
9
10 class Baz {
11     void baz(void) {
12         cout << "Baz::baz called\n";
13     }
14 };
```

(c) Example code before applying the transformation.

```
1  // Void params are redundant!
2  void foo() {
3      1 + 2;
4      cout << "Foo called\n";
5  }
6
7  // Void params are redundant!
8  void bar() {
9      cout << "Bar called\n";
10 }
11
12 class Baz {
13     void baz(void) {
14         cout << "Baz::baz called\n";
15     }
16 };
```

(d) Example code after applying the transformation.

Figure 17: Templates and example code to remove redundant void parameters.

In this specific case, we consider this behaviour to be incorrect, as the `void` parameter should also be removed from the method. However, in some cases, we may only want to match normal functions which are not part of a class. In order to resolve this, a directive should be added that specifies whether or not the tool should match class methods to functions.

Counterintuitive AST Representation Finally, the observant reader may have already noticed that the `bar` function has had a comment from the replacement template prepended to it. This is not a bug in the right-hand side template, but signifies that the matching algorithm may have made a mistake. What happened here is that the tool has also matched the `bar` function to the template, even though its parameter list is empty and does not contain a `void` parameter and as such, should not have matched. Hence, given this match, the right-hand side simply instantiated and replaced the source code as it is supposed to do. The reason why this false positive has been picked up is because Clang does not represent the `void` parameter in the AST. This means that a function which has no parameters is represented by an AST node whose parameter declaration list is empty, *regardless of whether or not a `void` argument is present.* As a result, the template picks up *any* function that accepts no arguments and returns nothing. Although this does not break the transformation in this case, the results can be surprising.

From this example, we can generally conclude that we are unable to parameterize type information and that the matching algorithm distinguishes between function and method declarations in a way that is sometimes undesired. These issues can be fixed by introducing directives.

7.1.3 Example 3: Comparisons to Boolean Literals

Another bad coding practice is comparing variables or function return values to boolean literals. It not only leads to code redudancy, but can also limit readability of the source code. Therefore, we created a simple transformation that eliminates such comparisons. Figure 18 shows the templates for this transformation, along with example source code before and after the transformation has been applied.

```
?lhs == true
```

(a) The left-hand side template.

```
?lhs
```

(b) The right-hand side replacement template.

```
int foo(bool b) {
    if (b == true) {
        return 1;
    } else {
        return 2;
    }
}

bool condition(int i) {
    return i < 5 == true;
}

void doUntil() {
    for(int i; condition(i) == true;
    ↪  i++) {
        // ...
    }
}
```

(c) Example code before applying the transformation.

```
int foo(bool b) {
    if (b) {
        return 1;
    } else {
        return 2;
    }
}

bool condition(int i) {
    return i < 5
}

void doUntil() {
    for(int i; condition(i) i++) {
        // ...
    }
}
```

(d) Example code after applying the transformation.

Figure 18: Templates and example code to remove comparisons to boolean literals.

This example shows a number of shortcomings when it comes to expressiveness and correctness.

Multiple Cases First of all, the transformation only matches equivalence comparisons to `true`. However, there are three more kinds of comparisons to boolean literals that can be transformed, namely equivalence to `false` and non-equivalence to both literals. To transform all of these comparisons, we would need to create four distinct transformations, each with their own left-hand side template. A better alternative would be to introduce left-hand side directives that can handle distinct cases and right-hand side directives that perform differently for each case.

Semicolons Secondly, in this example, only the condition of the `if`-statement is correctly transformed. The replacements for the `return`-statement and the condition of the `for`-statement unfortunately break the code, as not only the boolean comparison is eliminated, but also the semicolon is dropped. This is because when replacing the source code, the tool naively includes the trailing semicolon in source code that is to be replaced and expects the creator of

the right-hand side template to include a semicolon in the replacement text if necessary. It does this regardless of how the left-hand side or right-hand side templates are specified. However, the right-hand side template of this transformation does not include a semicolon, as this would break the `if`-statement, and as such, the semicolon that is removed from the other two matches is not inserted back into the code afterwards.

There are two ways to solve this issue. One option is to use more fine-grained left-hand side templates, such as only matching an `if`-statement. An adjusted template is shown in listing 19. However, this would require us to create many more templates and is not the most optimal solution. A better option would be to make right-hand side template instantiation smarter by either adding directives that can selectively disable the inclusion of the semicolon, or by automatically deriving whether or not it should be replaced.

```
if (?lhs == true) ?thenBranch
else ?elseBranch
```

Figure 19: An updated template to solve semicolon issues with the third template example.

7.1.4 Example 4: Converting Pointer Parameters to References

Another interesting idea for a transformation was proposed by OM Partners. Some time ago, they migrated their codebase from C to C++. In their software, they often make use of pointers to pass large data objects efficiently without having to copy the data. As such, they have a lot of functions that accept a pointer parameter and whose body contains an assertion that verifies that this pointer is not a null-pointer. They suggested creating a transformation that would find such functions and migrate the pointer parameters to use references, as references are never allowed to be null. Hence, the assertion could be dropped, and the code would be more idiomatic to C++.

Figure 20 shows a first attempt at such a transformation and an example source code snippet before and after the transformation. The left-hand side template attempts to match functions which take a pointer to a `LargeData` value and whose body asserts that this data is not a null-pointer. However, it again shows important issues with its expressiveness.

Types and Names First of all, we can again only match functions that have a specific return type, as was the case in example 2. Furthermore, it will only be able to match functions that take exactly one parameter whose type is a pointer to a `LargeData` value and whose name is `data`. If we wanted to match functions that take multiple parameters, or a pointer parameter to an abritrary type, directives and type parameterization are needed. Although we could represent the parameter as a name-only metavariable, we would also need to parameterize the variable used inside the function body and verify that the node bound to this metavariable refers to this parameter. As with a lot of the restrictions on expressiveness, we need a directive to represent this constraint.

Comments Another issue is that some of the comments in the function body are removed after the replacement. This is because comments are not represented in the AST, which results in them not being included in the source range of the bound metavariable. We can however fix this by retaining comments during preprocessing, which will result in them being part of the AST.

```
void ?name(LargeData *data) {
    assert(data != NULL);
    ?relevantBody
}
```

(a) The left-hand side template.

```
void ?name(LargeData &data) {
    ?relevantBody
}
```

(b) The right-hand side replacement template.

```
1  void processData(LargeData *data) {
2      assert(data != NULL);
3
4      // Do things with data
5      return;
6  }
7
8  void compute(LargeData *data) {
9      assert(data != NULL);
10
11     // Do things with data
12     int foo = 5;
13     data + foo;
14 }
```

(c) Example code before applying the transformation.

```
1  void processData(LargeData &data) {
2      return;
3  }
4
5  void compute(LargeData *data) {
6      assert(data != NULL);
7
8      // Do things with data
9      int foo = 5;
10     data + foo;
11 }
```

(d) Example code after applying the transformation.

Figure 20: Templates and example code to attempt to migrate pointers to references.

Macros A third issue lies in the fact that `assert` is implemented as a macro. This leads to an AST subtree which is not directly apparent from the source code. Although the tool correctly expands and matches this macro, which can be seen in the first function being transformed, a specific caveat hides in the macro implementation of `assert`. When an assertion fails, the program is stopped immediately and a diagnostic is printed that shows what caused the assertion to fail, along with the name of the function in which the assertion failed. Hence, somewhere in the AST resides a string literal that contains the name of the function, but since the name of the function is parameterized, the tool will be unable to match it correctly. The reason it still matches the first function is because the left-hand side template is derived from this function's AST subtree. Recall that a template is generated from an existing AST. Therefore, the nodes in the template and the nodes in the first function's subtree are identical, which explains why they match correctly. However, the second function has a different name and therefore carries a different string literal, which cannot be matched to the template's string literal node.

In order to fix this, we need a better way to handle macros while matching. One solution would be to prevent matching nodes that are the result of a macro expansion and instead attempt to match the macro as it is written in the source file. However, the original source code as a macro call is not represented in the AST, which means we cannot use our normal AST traversal for this purpose and another matching strategy would need to be used for these specific cases.

Lack of Verification A fourth and final issue is that this transformation in its current state completely breaks the source code. Not every pointer can simply be converted to a reference as references have a few restrictions. For example, once a reference has been instantiated, it

cannot be changed to refer to a different object. Hence, the left-hand side template would need to verify such behaviour does not occur, which is difficult to check, even with directives. Additionally, a reference acts as a normal value in the function body and should not be dereferenced before usage. Our transformation would need to identify all uses of the pointer parameter and change them accordingly. For example, the second function in the example source code performs pointer arithmetic on the parameter. If we change this parameter to a reference, the addition operation would represent something completely different. Hence, before replacing the code, the transformation must be able to assure that such behaviour-changing replacements do not occur. Finally, any call to a transformed function must be edited to pass the argument by reference instead of by pointer. To do all of this, we would again need some advanced directives.

7.1.5 Example 5: Renaming Namespaces

The final example we will consider is renaming the main namespace used in the tool's own source code. The tool's classes are defined in a namespace `X`. However, when our prototype reaches a production-grade quality, its name will likely change and therefore this namespace would most likely be renamed too. Hence, we will already design a transformation that can be used for this purpose.

When renaming a namespace, we also need to rename all references to this namespace throughout the source code. In order to do this, we need to edit all `using` directives, which are used to tell the compiler that it should search in a given namespace when it cannot resolve certain identifiers. This spares us from having to explicitly mention the namespace on each use of one of its members. However, the tool sometimes still explicitly signifies that a certain function or class belongs to namespace `X` in order to resolve nameclashes with identically named members from the `clang` namespace. Hence, three kinds of edits need to be performed: renaming the namespace itself, updating `using` directives to use this new name, and updating explicit mentions of the namespace. Figure 21 shows an overview of these edits before and after the transformation has been applied.

```
1  namespace X {
2      class MatchResult;
3  }
4
5  using namespace X;
6
7  X::MatchResult res;
8  X::Lexer lex;
```

(a) The source code before the transformation.

```
1  namespace Tool {
2      class MatchResult;
3  }
4
5  using namespace Tool;
6
7  Tool::MatchResult res;
8  Tool::Lexer lex;
```

(b) The source code after the transformation.

Figure 21: An overview of the edits that need to be performed in order to rename the namespace from `X` to `Tool`.

Earlier examples already showed that we cannot parameterize the type of a declared variable or function. Hence, we cannot perform the third edit, the updating of explicit namespace indications in declarations, as we would need a directive that parameterizes a type to only match types from namespace `X`. However, we can perform the first two edits. The transformations for these edits are shown in figure 22. The upper templates are used to rename the namespace itself, while the bottom templates are used to update the `using` directives.

```
namespace X {
    ?namespaceDecls
}
```

(a) The left-hand side template used to rename the namespace.

```
namespace Tool {
    ?namespaceDecls
}
```

(b) The right-hand side replacement template used to rename the namespace.

```
using namespace X;
```

(c) The left-hand side template used to update `using` directives.

```
using namespace Tool;
```

(d) The right-hand side replacement template used to update `using` directives.

Figure 22: Templates for renaming namespaces and updating `using` directives.

The transformations defined by these templates are fairly straightforward and work as expected when we run them on the tool's source code individually. However, in order for the full transformation to work correctly, these transformations need to be carrier out together. If we were to first rename the namespace, the `using` directives in the partially transformed code would refer to a now unknown namespace and the tool would fail to parse them when applying the second transformation. Similarly, if we were to first transform the `using` directives, they would refer to an unknown namespace as the namespace itself still carries its original name. Hence, to correctly transform the source code, both transformations must be applied on the same source ASTs without parsing the source files again in between them.

Unfortunately, the tool can currently run only one transformation at a time. To make this migration work, it needs to be able to run multiple transformations at once. It is possible to implement this, but care must be taken to ensure a replacement made by one transformation does not overlap with a replacement made by another. A possible alternative would be to allow a left-hand side template to match multiple cases, as was proposed in section 7.1.3. This way, both parts of the migration can be expressed as one transformation and can be carrier out together.

7.1.6 Topics Not Covered in the Examples

These five example transformations already provide a good representation of what can and cannot be expressed in a transformation, but there are certain interesting topics that have not been covered yet.

The first of these topics is the tool's ability to transform header files. We applied the transformation in example five on the tool's header files as well, and found that it correctly carries out its task as if they were normal source files. This even works when mixing header files and normal source files in its input list. As a result, the tool can be used to transform entire codebases, including publicly accessible header interfaces, or even completely header-only libraries.

Secondly, certain important C++ features have not been considered while developing the tool. Concepts such as lambdas and generic templates[4] may fail to be parameterized or matched. However, using these concepts in a right-hand side replacement is possible, as it uses a string-based representation that can contain any source code. In the future, full support for these concepts should be added.

[4]With templates, we mean C++'s native `template` construct used for generic programming, not the left-hand side or right-hand side templates used by the tool.

Finally, although the tool is designed to transform C++ code only, there is a possibility that it can also transform C or even Objective-C code. As Clang can parse both of these languages and uses the same classes for their AST representation, the tool may be able to correctly generate and match left-hand side templates for certain source code written in these languages. However, it will certainly not be able to correctly handle any source code as it is intended for C++ only. However, towards the future, support for these languages could be introduced.

7.2 The Tool's Ease-of-Use

Ease-of-use is a very subjective metric and as such, measuring it objectively is difficult. However, we can raise some objective arguments as to why the tool allows more developers to use it compared to the other transformation strategies outlined in section 2.

Framework X does not require the user to have an extensive knowledge of Clang or its internals in order to create transformations. Users do not have to be aware of how Clang structures its AST as our left-hand side template generator maps source locations to AST nodes automatically. However, there are some caveats to the Clang AST structure that could lead to unexpected results, but we can enhance the tool to handle these situations correctly. For example, name-only metavariables allow the user to parameterize the name of a function, variable or class, which would normally be impossible due to the name being a property of an AST node and not an AST node of itself.

Additionally, we argue that template-driven transformations are more intuitive to design and understand. Because both sides of the transformation are in essence source code containing wildcards and placeholders, it is easy to reason about which source code snippets it will replace and what these snippets will be replaced with. As such, the learning curve is substantially lower than those of the Clang transformation strategies, where figuring out what will be matched and how it will be replaced requires a thorough investigation of the source code implementing a transformation. Because of this relatively low learning curve, we believe more developers will be enticed to let an automated tool transform their source code, instead of doing so manually.

Unfortunately, the main obstacle is the way transformations are currently defined. Creating and editing the configuration file manually is more difficult than parameterizing source code directly. However, the addition of a GUI to interactively create and adjust transformations will make this process easier and more intuitive. This approach also lends itself well to be integrated into an IDE, which would make the task of transforming source code even faster and easier.

8 Future Work

As we have seen in the previous section, the tool can be used to apply simple but useful transformations. However, there exists a lot of room for improvement. Most of these improvements relate to introducing directives to the transformations. Subsection 8.1 lists important categories of directives together with some examples of where they can be used. Subsection 8.2 describes other important functionalities that can be added to the tool. Finally, subsection 8.3 describes features that could be added once our prototype reaches a production-grade quality.

8.1 Directives in Transformations

One of the most important ways to make a transformation more powerful is by adding directives to them. As seen in the evaluation of the expressiveness of the tool, a lot of facts cannot be described by metavariables and literal source code alone. Directives can help to express such facts. They can be added to both the left-hand side templates and right-hand side templates.

At the left-hand side of a transformation, a directive influences the matching algorithm to verify additional constraints on potential matches and to traverse its AST in different ways. At the right-hand side, a directive changes how replacements are made.

8.1.1 Left-Hand Side Directives

There are many possible directives that can influence the matching algorithm. In general, these can be divided into five categories.

Syntactic Directives Syntactic directives are primarily intended to change the way the matching algorithm traverses the AST of its potential matches. For example, a directive could specify that a certain template node is optional. For instance, the `else`-branch of an `if`-statement is optional. However, we cannot create a template that can match `if`-statements with or without an `else`-branch using only metavariables and literal source code because all AST nodes should match. A template with an `else`-branch cannot match a potential match without one, and similarly a template without an `else`-branch cannot match a potential match with one. Hence, a directive is needed that specifies that the matching algorithm should try to match this part of the AST, but may still consider the potential match to be valid if the matching fails in this part. This can also be applied to other nodes, such as optional statements in a function body.

Another example of a syntactic directive is one that specifies that a certain AST node must match any of the descendants of the current AST subtree. This match can be a direct child of the current AST node, but can also be nested in its direct children. This is useful to for example find functions which perform a certain computation somewhere in their body.

Semantic Directives A semantic directive is used to represent semantic information that cannot be expressed by metavariables or simple source code. A prime example of such a directive is matching type information. For example, we may want to match all variable declarations whose type is a numeric type, regardless of whether this type is signed or unsigned, a long integer or a floating point numeral. Another example of where we need semantic information is presented in section 7.1.5, where we need to transform all variable and function declarations that mention a certain namespace in its declaration. A final example is when we want to match variables that may or may not be declared a constant value.

Another scenario where we need semantic information is to verify that a certain AST node refers to another node. For example, recall from section 7.1.4 that we could not parameterize the name of the pointer parameter as we needed to assure that it was indeed this parameter that was used in the `assert` call. Directives can help us parameterize this, as we can express a constraint that the variable used in the `assert` call indeed refers to the pointer parameter.

Structural Directives Structural directives are used to express constraints on the class hierarchy of the source code. Consider for example the classes depicted in listing 23. Class `B` derives class `A` and as such, inherits from it. Similarly, class `C` inherits from `B`. Since inheritance is a transitive relation, this means that `C` also inherits from `A`, although `C` does not directly derive `A`.

```
1 class A {};
2 class B : public A {};
3 class C : public B {};
```

Figure 23: A simple class hierarchy exemplifying the transitive nature of inheritance.

If we were to write a transformation that gathers all classes that derive `A`, class `C` would be expected to match too. However, since there is no direct inheritance relation from `A` to `C`, we cannot express this without directives. Hence, an example of a structural directive is one that analyses the *whole* derivation chain to find deriving classes. Other examples are a directive that only matches abstract classes, or a directive that matches classes which override specific behaviour of a specific base class.

Control Flow Directives The fourth category of directives contains those that can express facts regarding the control flow of a program. Examples of the purpose of such directives are finding "dead code", i.e. code that will never be executed; matching all functions called within a specific block; or matching all blocks that call a specific function.

For instance, in the example presented in section 7.1.4, we created a transformation that updates function declarations to use references instead of pointer parameters. We mentioned that in order not to break the code, the transformation should also update all calls to the transformed functions to pass the argument by reference instead of by pointer. To do this, a control flow directive may be useful to gather all calls to the transformed functions.

Dataflow Directives The final and most advanced directives we can add to the left-hand side templates are dataflow directives. Consider for example the code shown in listing 24. It shows the flow of a data value in a C++ program. There are four active references to this value: `a` carries its initial definition, `b` and `c` point to this value, and `d` is a variable containing a copy of this value. Furthermore, we say that `b` and `c` alias each other: they are distinct pointer variables that point to the same value.

```
1 int a = 123;
2 int *b = &a;
3 int *c = b;
4 int d = *c;
```

Figure 24: An example of the flow of a data value in C++.

Dataflow directives can be used to inspect this flow of data in the program. Examples of such directives are matching pointers that may alias each other, matching instances where a value is copied or moved, or even matching instances where values are assigned, initialized, created or destructed. Hence, dataflow directives are very powerful and can be used for very advanced purposes, such as even finding potential memory leaks. However, the downside to this power is the fact that such information is often very difficult to derive.

8.1.2 Right-Hand Side Directives

Directives at the right-hand side are used to instruct the tool on how to perform replacements. For example, paired with a left-hand side directive that indicates that an AST node should optionally be matched, as described in section 8.1.1, a right-hand side directive can specify that the replacement text should be instantiated differently when this node has in fact been matched.

Other examples of right-hand side directives are distinguishing between different cases based on what has been matched in the left-hand side, which can be useful for the transformation in section 7.1.3; transforming only specific metavariables instead of the whole match; or specifying where a certain replacement text should be inserted into the original source code. For instance,

we may want to add new class members to the bottom of a class definition. A replacement template can then specify where these new members should be added in the class.

8.2 Various Enhancements

There are a number of other enhancements that should be made to the program in the future. First of all, we need to add support for all other C++ concepts, such as lambdas and the language's native generic template mechanisms. This is because our tool must at least support all major C++ concepts in order to be powerful enough to handle more advanced transformations.

We must also perform further work on supporting macros and comments. The tool should at least be able to retain comments, prevent rewriting matches that occur inside of a macro expansion, and correctly match code that contains a macro expansion. Additionally, we should investigate what happens to whitespace when rewriting nested source code, which may present an annoyance if the replacement text is not properly indented.

Another important enhancement would be the addition of a GUI to create and edit transformation templates. As was mentioned in section 7.2, this would make the tool significantly easier to use.

Finally, it would be beneficial to use an AST representation on the right-hand side of the transformation. When adding right-hand side directives, an AST representation will help us to correctly perform rewrites. Fortunately, Clang provides us with conventient abstractions to perform this. It provides a class that can be used to create and replace AST subtrees, and provides methods to render an AST back to source code. Metavariable instantiation would then be performed by replacing the metavariables with the actual nodes that were bound to the metavariable during the matching. Afterwards, these template instances could then be injected back into the source file's original AST, replacing the subtree of the full match. After replacing all matches, the original ASTs would then be written back to the source files.

8.3 Future Applications

Once our tool is able to represent and apply more advanced transformations, we can look into how we could extend its functionality further. One interesting and achievable extension is to add support for C or Objective-C. As Clang already supports parsing and compiling these languages and uses the same parser and the same AST node classes to represent them, adding support for these specific languages would not pose a big challenge.

We could also look into integrating the tool into an IDE. This would allow for a seamless integration of a template-driven transformation tool into the daily workflow of developers.

Finally, we could look into enhancing the process of creating and editing transformations itself. Often, a template matches too many or too few results, which means it needs to be specialized or generalized, respectively. We could guide the user through the process of generalizing or specializing templates by automatically recommending changes. Ekeko/X already uses such an approach [MDR16], hence we could try and port these functionalities over to Framework X. We could even try to take this a step further by monitoring the source code itself for recurrent changes and suggest templates for representing this transformation.

9 Conclusion

Transforming C++ code is a laborious task and as such, can benefit from automatisation. The Clang compiler project already includes a number of frameworks which can be used to perform transformations automatically, however, they require a lot of expertise for a developer to use

them. The reason for this is that an extensive knowledge of Clang and its libraries is necessary in order to correctly match and replace source code.

In this thesis, we presented a tool prototype called "Framework X", which serves as a proof of concept of template-driven program transformations in C++. Its goal is to simplify creating and interpreting source code transformations by representing them as source code templates. Although it is built on top of Clang's libraries, using a template representation allows us to require less Clang expertise of our users.

The tool performs two phases. First, it uses the left-hand side template to match the template to the source code that needs to be transformed. Afterwards, it takes these matches and replaces them according to the right-hand side template. Both of these templates may contain metavariables. At the left-hand side, these act as wildcards to the matching algorithm and will be bound to the corresponding node in the source file's AST. At the right-hand side, these act as placeholders which eventually get instantiated by the original source code residing behind the node bound to the metavariable.

Parsing left-hand side templates to an AST poses a number of challenges. Not only should a parser be able to detect metavariables and correctly represent them in an AST, it should also be able to cope with the lack of context that is typical of a source code template. In order not to have to parse the left-hand side templates, we instead chose to generate a left-hand side template starting from an existing source file, selecting a certain part of this file's AST to use as the template and replacing certain AST nodes with metavariables. The user can supply this information by creating a configuration file.

To find transformation candidates, the tool uses a matching algorithm that traverses through the left-hand side template's AST and compares it to the ASTs of the input files. While doing this, it instantiates the metavariables and uses custom comparators to determine the equivalence of AST nodes. Afterwards, the matching algorithm sorts the matches it found and eliminates conflicts to prevent the right-hand side from corrupting the source code.

The right-hand side templates are represented as a list containing metavariables and strings representing literal snippets of source code. For each match obtained during the matching phase, the right-hand side template is instantiated by reading the original source code residing behind the AST node for each metavariable. Afterwards, the full match gets replaced by the instantiated replacement text.

The tool has been evaluated based on expressiveness and ease-of-use. Although it can be used to apply simple but useful transformations, it lacks the expressive power to represent more advanced transformations. The key reason for this is the lack of directives. Such directives can be used in both the left-hand side, where it influences the matching algorithm, as well as in the right-hand side, where it dictates how matches should be replaced.

The tool's ease-of-use is difficult to measure objectively, but we argue that it is easier to use than Clang's transformation frameworks because it does not require intricate knowledge of Clang. Furthermore, using source code templates to define transformations makes creating, editing and interpreting transformations easier. However, ease-of-use is limited by the way left-hand side templates are generated. The requirement of manually creating or editing a configuration file is cumbersome.

A number of ways have been proposed to improve the tool's expressiveness and ease-of-use. These range from adding support for more C++ concepts to developing a GUI for the tool.

References

[Ben12] Eli Bendersky. How Clang handles the type / variable name ambiguity of C/C++. Web, July 2012. Accessed: 2017-05-26.

[Car11] Chandler Carruth. Clang MapReduce – Automatic C++ refactoring at Google scale. `http://llvm.org/devmtg/2011-11/#talk2`, November 2011. Accessed: 2017-05-26.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DRI14] Coen De Roover and Katsuro Inoue. The Ekeko/X program transformation tool. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 53–58. IEEE, 2014.

[DRS14] Coen De Roover and Reinout Stevens. Building development tools interactively using the Ekeko meta-programming library. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 429–433. IEEE, 2014.

[MDR16] Tim Molderez and Coen De Roover. Automated generalization and refinement of code templates with Ekeko/X. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 669–672. IEEE, 2016.

[Opd17] Ruben Opdebeeck. Transforming entire C++ codebases using Clang, January 2017.