Large-Scale Automated Refactoring Using ClangMR

Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, Zhanyong Wan
Google, Inc.
Mountain View, California 94043
Email: {hwright,djasper,klimek,chandlerc,wan}@google.com

***Abstract*—Maintaining large codebases can be a challenging endeavour. As new libraries, APIs and standards are introduced, old code is migrated to use them. To provide as clean and succinct an interface as possible for developers, old APIs are ideally removed as new ones are introduced. In practice, this becomes difficult as automatically finding and transforming code in a semantically correct way can be challenging, particularly as the size of a codebase increases.**

In this paper, we present a real-world implementation of a system to refactor large C++ codebases efficiently. A combination of the Clang compiler framework and the MapReduce parallel processor, ClangMR enables code maintainers to easily and correctly transform large collections of code. We describe the motivation behind such a tool, its implementation and then present our experiences using it in a recent API update with Google's C++ codebase.

I. Introduction

As software systems evolve, existing code often must be updated to allow for future feature development, removal of old or incompatible interfaces, and further maintenance tasks. Large software systems often suffer from an inability to evolve to meet new demands [1], largely due to increasing amounts of technical debt [2], and the inability of code maintainers to automatically update large portions code in a semantically safe way. Even automatic changes which may appear safe in isolation may introduce semantic conflicts that cause faults which are difficult to detect [3].

Google addresses this challenge by using ClangMR, a tool that uses semantic knowledge from the C++ abstract syntax tree (AST) to make editing decisions. ClangMR programs are written in C++ and provide a wide variety of different refactoring capabilities. ClangMR also takes advantage of the independent nature of most refactoring work by parallelizing its analysis across many computers simultaneously by using the MapReduce framework [4]. This combination of knowledge, flexibility and speed allows code maintainers to perform complex transformations across millions of lines of C++ code in a matter of minutes.

While this specific implementation of ClangMR is dependent upon Google's infrastructure, a significant portion of the system is available as open source software as part of the LLVM project [5]. The Clang AST and its node traversal and matching infrastructure are all readily available for public consumption and improvements continue to be publicly released.

In the following pages, we present the basic implementation details of the ClangMR system in use at Google. We also discuss a sample large-scale refactoring effort recently completed which modified over 35,000 function call sites across 100 million lines of code.

A. Existing Tools

ClangMR is not the first tool designed to do complex refactorings over C++, but it is unique in its flexibility, speed, and use in industrial applications.

While useful in simple cases, traditional regular-expression-based matching tools lack the semantic knowledge that complex transformations often require. For instance, these tools can not distinguish calls to similarly-named methods which are members of different classes, making them unsuitable for large-scale semantically-safe refactoring. Such naming conflicts also frequently increase as the size of a codebase grows, leading to a lack of scalability.

Many integrated development environments, such as Eclipse, provide a limited set of refactoring tools. While these tools take advantage of compile-time knowledge, they are generally limited to a single file or package, and only support the operations built into the tool. Since these tools run on a developer's local workstation, processing large collections of source code is often intractable.

Other tools, such as Pivot [6], may be versatile, but have difficulty scaling to many millions of lines of code. Some tools that do scale, such as those described by Kumar, et al in [7], perform specific types of transformations at scale, but lack the versatility of ClangMR. While these are useful in theory, we have yet to see existing tools demonstrated on a large, real-world corpus of production-quality code.

II. Motivation

Google maintains a large mixed-language codebase in a single repository, with a significant portion written in C++. Like many large software systems, this codebase continues to evolve as new APIs, idioms and standards are introduced. While new code may use the improved techniques, large amounts of legacy code still uses old APIs and standards.

To help maintain as small an API surface as reasonable, the developers and teams responsible for introducing new core APIs are also tasked with removing old ones and migrating existing callers to the new abstractions. Google engineers are also actively engaged in source code rejuvenation efforts to migrate custom implementations to the C++11 standard [8].

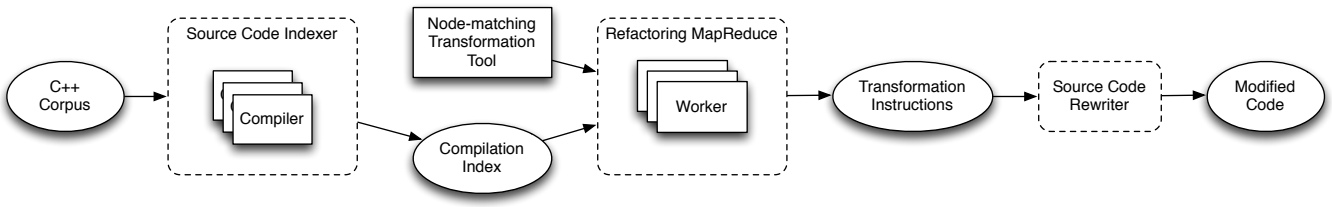

Fig. 1: ClangMR processing pipeline

ClangMR has seen wide use within Google for the past two years, and it continues to evolve into a more powerful and easy-to-use tool. Recent refactoring projects include:

- Updating legacy C++98 code to take advantage of features in the C++11 standard.
- Removing redundant explicit type conversions.
- Updating callers to improved APIs, such as string manipulation and file handling.

While these may seem trivial in isolation, performing these migrations on millions of lines of code would not be practical without ClangMR. In general, ClangMR helps reduce the accumulated technical debt of a diverse codebase built over the course of more than a decade. An example of a specific migration effort is discussed in depth below.

III. Implementation

The ClangMR implementation consists of three parts:

- an indexer which describes how to compile the C++ codebase into a collection of ASTs
- a transformation-specific node matching tool which builds ASTs from the index, matches applicable AST nodes, and outputs editing instructions
- a source code refactorer which consumes editing instructions and modifies the files in a local version control client to effect the desire transformations

Of the three separate components, only the transformation-specific node matching tool is specific to an individual transformation. The compilation index can be consumed by multiple node matching tools, and the refactorer can operate on their standardized output. This architecture means a single code maintainer needs only implement an appropriate matcher for the desired transform, rather than the entire pipeline. An overview of the pipeline is shown in Figure 1, and the individual components are discussed below.

A. Source Code Indexer

A daily task builds a denormalized index of the acts of compilation—the commandlines used, the various files read, and the filesystem layouts of those files. Storing the entire precomputed ASTs would be space-prohibitive, but this index serves to provide a way to quickly construct those ASTs from a snapshot in source code history. Determining these compilation steps is largely independent for individual files, so this process can be parallelized across a number of different machines in Google's standard build cluster.

B. Node Matcher

The node matcher uses the most recent compilation index to produce ASTs for the entire collection of source code. Because this intermediate AST is only required for node traversal, it can be stored in memory for the duration of the traversal. Even though the size of the AST can be quite large, traversing the AST demonstrates high memory locality, so it is quite fast. Experience shows that it is roughly as fast to compile C++ code into a memory-held AST as it is to read a completely annotated AST out of distant storage.

Because each translation unit has a separate root node in the AST, and each source file is generally an independent translation unit, the node matcher can operate on separate translation units in parallel. At Google, we use the MapReduce framework [4], but ClangMR could be adapted to use other suitable parallelization systems.

Most of the node matching executes outside the care or control of the programmer performing the refactoring. As input, a developer provides an appropriate node matching expression, and a callback to be invoked when that expression is matched. In practice, these tend to be relatively small: a few hundred lines of C++ code. The ClangMR infrastructure handles running the node matching algorithm and invoking the callbacks on the appropriate nodes.

The node matching infrastructure first reads the index from bigtable, uses it to recursively traverse the AST nodes, and then invokes any callbacks that have been registered for matched nodes. Any output produced by the callbacks is then serialized to disk for use by the refactorer.

1) Node Matching Expression: Developers use node matching expressions to register a callback with the ClangMR processor. These expressions may match a variety of node types, and can be qualified with various logical filters and traversal operations. Examples of node matching expressions used at Google are shown in Figure 2, and a complete reference is available on the Clang website [9].

2) Callbacks: When the preprocessor matches a node in the AST, it invokes the provided callback with the node and some context about where it was found, such as the source location. The callback is written in C++, allowing it to query the properties of the node and its context and make complex decisions about what edits, if any, can be applied. They may also decide to not make any edits. This technique allows for much more powerful transformations than pure textual substitution.

```
StatementMatcher matcher =
  callExpr(allOf(
    argumentCountIs(1),
    callee(functionDecl(hasName(
      "::Foo::Bar")))))
  .bind("call");
```

(a) Match all calls to `Foo::Bar` which have a single argument

```
TypeLocMatcher matcher =
  loc(qualType(hasDeclaration(
    recordDecl(hasName(
      "::scoped_array")))))
  .bind("loc");
```

(b) Match all `scoped_array` typed variable declarations

Fig. 2: Example node matching expressions

As output, each callback may generate a set of instructions on how to transform the code on a textual level. Similar to a text-based patch, these instructions describe edits to the target source file as a series of byte-level offsets and additions or deletions. These instructions are then serialized to disk, and used as input to the source code refactorer.

An example callback implementation is shown in Figure 3. Much of the error checking and boilerplate has been omitted, but combined with the matcher in Figure 2a, this example renames all calls to `Foo::Bar` to `Foo::Baz`, independent of the name of the instance variable, or whether is is called directly or by pointer or reference.

C. Refactorer

The source refactorer reads the list of edit commands generated by the node matcher callbacks and filters out any duplicate, overlapping or conflicting edit instructions before editing the source code in the local version control client. Each edit is processed serially in the version control client on the local workstation of the developer, which is synchronized to the version of code stored in the compilation index.

Even though it is local and serial, in practice, this step is relatively quick, and edits spanning thousands of files are performed on the order of tens of seconds. A final pass with ClangFormat, a Clang-based formatting tool [10], ensures the resulting code meets formatting and style guide recommendations.

D. Limitations

While ClangMR enables a large class of refactoring operations at scale, it does have limitations. ClangMR can only refactor changes which are self-contained within translation units. Large sets of changes still require tedious manual review—though review tools are improving to allow faster automated review of large changes. Finally, ClangMR requires learning some nuances of the Clang AST, which requires developer investment.

```
void Refactor(const MatchFinder::MatchResult& res) {
  Clang::CXXMemberCallExpr* call =
    res.Nodes.getStmtAs<clang::CXXMemberCallExpr>(
      "call");

  const clang::MemberExpr *member_expr =
    llvm::dyn_cast<clang::MemberExpr>(
      call->getCallee());

  EditState state(res, call);
  state.ReplaceToken(member_expr->getMemberLoc(),
    "Baz");

  Report(&state);
}
```

Fig. 3: Example node matcher callback

In spite of these limitations, ClangMR enables engineers to make significant semantically-correct changes to large C++ codebases.

IV. Practical Application

In this section, we present an actual large-scale transformation done at Google using ClangMR and demonstrate the technical advantages to this approach. While not the largest or most complex refactoring performed done at Google, this example demonstrates the versatility of the AST-based refactoring approach.

A. Splitting Strings

Google's internal software libraries have historically provided a number of methods for splitting strings. One of the most common APIs is known as `SplitStringUsing`, shown in Figure 4. As the name implies, `SplitStringUsing` parses a string of characters using a set of delimiters, and inserts the resulting substrings into the provided vector of strings.

Google engineers recently introduced a single method to unify and consolidate the various split-related APIs. Known as `strings::Split`, it is shown in Figure 5. This new API is appropriately parameterized to handle the use cases of most existing split functions in a single interface. While a complete discussion of `strings::Split` is out-of-scope for this paper, suffice it to say that the new API was well received by Google engineers, led to reduced numbers of bugs and has been proposed for the next iteration of the ISO C++ standard [11].

After the new API debuted, most Google developers were not anxious to invest the effort to migrate their currently-functioning code to `strings::Split`. Due to the semantic difference between the APIs, any automatic transformation would require semantic knowledge, not just a strict textual substitution. At the time, there were roughly 45,000 callers of `SplitStringUsing`, and migrating them by hand was infeasible. One software engineer was attempting to convert these callers manually using a combination of inspection and editor macros, but his efforts could not keep up with an

```
void SplitStringUsing(const string& full,
                      const char* delimiters,
                      vector<string>* result);

void foo() {
  string input;
  vector<string> output;
  ...
  SplitStringUsing(input, "-", &output);
}
```

Fig. 4: `SplitStringUsing` example

```
namespace strings {
template <typename Delimiter, Predicate>
Split(StringPiece text, Delimiter d, Predicate p);

struct SkipEmpty {
  bool operator()(StringPiece sp) const {
    return !sp.empty();
  }
};
}

void foo() {
  string input;
  ...
  vector<string> output =
    strings::Split(input, "-",
                   strings::SkipEmpty());
}
```

Fig. 5: `strings::Split` example

evolving codebase. ClangMR allowed us to migrate the bulk of existing callers and encourage engineers to use the new API in new code.

1) Node Matching Implementation: Instead of using a node matching expression to decide which kinds of calls were transformable, the expression simply matched *all* calls to `SplitStringUsing`, and relied upon logic in the callback to determine if the transformation was safe or not. A "safe" transformation meets the following criteria:

- The output variable declaration was in the same scope as the call to `SplitStringUsing`.
- The output variable was not referenced between its declaration and the call to `SplitStringUsing`.

Both of these criteria were easily examined using the context available in the AST provided by ClangMR. In some complex situations, we chose to defer the edits to be done manually.

Because ClangMR allows the examinations of literal values known at compile-time, we were also able to determine how to rewrite the actual function calls themselves. The default behavior of `SplitStringUsing` is to use any of the provided characters as a delimiter, but doing so with `strings::Split` this requires a separate `Delimiter` argument, which could be omitted in the case of only one delimiter character. By examining the literal delimiter arguments to the old function call, we could simplify a large number of common cases when rewriting to the new code.

2) Experiences: The initial ClangMR program transformed about 35,000 callers of `SplitStringUsing` to `strings::Split`, and these changes were mailed for review in 3,100 separate chunks, though not all simultaneously. These were often reviewed quite quickly, with an 80th-percentile review time of just over two minutes. The bulk of reviews were completed over two months, with a small number requiring another month to complete.

One benefit of using ClangMR for this work was that the transformation could be repeated. During the course of this effort, we frequently re-ran the tool to find any additional uses which had been added since the initial run. This made it easy stay current with an ever-changing codebase.

V. Conclusion

In this paper, we presented the design and implementation of ClangMR, a highly parallelized, semantically-aware refactoring tool based upon the Clang compiler running on MapReduce. We also discussed an example application of a real-world large-scale transformation using the ClangMR system to update callers of deprecated APIs.

ClangMR allows for fast and versatile refactoring of large C++ codebases, and has been applied to many problems within Google, enabling maintainers to keep millions of lines of C++ code nimble and accessible to thousands of engineers.

References

[1] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 1–12, 2001.

[2] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 35–38.

[3] G. Thione and D. Perry, "Parallel changes: detecting semantic interferences," in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, vol. 1, 2005, pp. 47–56 Vol. 2.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[5] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[6] B. Stroustrup and G. Dos Reis, "Supporting sell for high-performance computing," in *Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 458–465.

[7] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating c++ programs through demacrofication," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 98–107.

[8] P. Pirkelbauer, D. Dechev, and B. Stroustrup, "Source code rejuvenation is not refactoring," in *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, ser. SOFSEM '10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 639–650.

[9] (2013) AST matcher reference. [Online]. Available: http://clang.llvm.org/docs/LibASTMatchersReference.html

[10] (2013) ClangFormat. [Online]. Available: http://clang.llvm.org/docs/ClangFormat.html

[11] G. Miller. (2013) std::split(): An algorithm for splitting strings. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3510.html