Nobrainer: An Example-Driven Framework for C/C++ Code Transformations

Valeriy Savchenko[1(✉)], Konstantin Sorokin[1(✉)], Georgiy Pankratenko[1(✉)], Sergey Markov[1(✉)], Alexander Spiridonov[1(✉)], Ilia Alexandrov[1(✉)], Alexander Volkov[2(✉)], and Kwangwon Sun[3(✉)]

[1] Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow 109004, Russian Federation
{vsavchenko,ksorokin,gpankratenko,markov, aspiridonov,ialexandrov}@ispras.ru
[2] Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow 119991, Russian Federation
asvolkov@ispras.ru
[3] Samsung Electronics, Samsung GEC, 26, Sangil-ro 6-gil, Gangdong-gu, Seoul, South Korea
kwangwon.sun@samsung.com

Abstract. Refactoring is a standard part of any modern development cycle. It helps to reduce technical debt and keep software projects healthy. However, in many cases refactoring requires that transformations are applied globally across multiple files. Applying them manually involves large amounts of monotonous work. Nevertheless, automatic tools are severely underused because users find them unreliable, difficult to adopt, and not customizable enough.

This paper presents a new code transformation framework. It delivers an intuitive way to specify the expected outcome of a transformation applied within the whole project. The user provides simple C/C++ code snippets that serve as examples of what the code should look like before and after the transformation. Due to the absence of any additional abstractions (such as domain-specific languages), we believe this approach flattens the learning curve, making adoption easier.

Besides using the source code of the provided snippets, the framework also operates at the AST level. This gives it a deeper understanding of the program, which allows it to validate the correctness of the transformation and match the exact cases required by the user.

Keywords: Code transformation · Global refactoring · C/C++

1 Introduction

The lifecycle of any software project is a constant evolution. Not only does it mean writing new code while adding new features, but it also includes continuously modifying the existing code. Excessive focus on extending the system's

N. Bjørner et al. (Eds.): PSI 2019, LNCS 11964, pp. 140–155, 2019.
https://doi.org/10.1007/978-3-030-37487-7_12

functionality can lead to a rapid accumulation of the project's technical debt. The concept of *technical debt* is a widespread metaphor for design-wise imperfection that boosts initial product development and deployment. With time, however, the debt grows larger and can potentially stall the whole organization [3].

A common way to mitigate this problem is *refactoring* [2,12], which is a modification of the system's internal structure that does not change its external behavior [4]. It helps to eliminate existing architectural flaws and ease further code maintenance. Murphy-Hill et al. [9] have estimated that 41% of all programming activities involve refactoring. The same study also concluded that developers underuse automatic tools and perform code transformations manually despite the fact that a manual approach is more error-prone. Research performed on StackOverflow website data [10] found that corresponding tools can be too difficult and unreliable, as well as require too much human interaction. This reveals a few natural requirements for a beneficial refactoring tool—it should be easy to use, ask a minimal number of questions from the end user, and rely on syntactic and semantic information in order to ensure the correctness of the performed transformations.

Highly customizable refactoring tools typically utilize additional domain-specific languages (DSL) for describing user-defined transformation rules [5,7,14]. Such languages need to express both the intended refactoring and the different syntactical and semantical structures of the target programming language. Adopting a DSL can be too overwhelming in the case of C/C++ languages because the language itself is already complex. Studies show that C and C++ take longer to learn [8], and projects in these languages are more error-prone [11] compared to other popular languages.

This insight further qualifies the ease-of-use requirement: the tool should not introduce another level of sophistication on top of C/C++ nor expect additional knowledge from its user.

This study presents the Clang-based transformation framework `nobrainer`, which is built on such principles. The expression *a no-brainer* refers to something so simple or obvious that you do not need to think much about it.[1,2] This concept reflects the core idea behind `nobrainer`, the idea of providing an easy-to-use framework for implementing and applying a user's own code transformations.

Individual `nobrainer` rules are written in C/C++ without any DSLs. Each rule is a group of examples that represents situations that should be refactored and illustrates the way they should be refactored. They look exactly like developer's code, thus flattening the learning curve of the instrument.

In this paper, we describe the main principles behind `nobrainer`, illustrate the most interesting design and implementation solutions, and demonstrate the tool's application in real-world scenarios.

[1] https://www.merriam-webster.com/dictionary/no-brainer.
[2] https://dictionary.cambridge.org/dictionary/english/no-brainer.

2 Related Work

This section covers the various approaches on code transformation and automatic refactoring presented in the literature. We distinguish two key points in the current review. The first point is the form, in which the transformations are described. The second point is the way these transformations are performed. There are a few similar approaches that can be combined and compared.

Most of the tools reviewed here rely on a separate syntax for describing transformation or refactoring rules. For example, Waddington et al. [5] introduce their language YATL; whereas, Lahoda et al. [7] extend the Java language to simplify rule descriptions. We believe that various types of DSLs can confuse the user and introduce another layer of complexity. Wright and Jasper [14] describe a different approach. Their tool `ClangMR` adopts Clang AST matchers [1] as a mechanism for describing the parts of the user's code that should be transformed. The user must define replacements in terms of AST nodes. The authors imply that the user is familiar with the principles of syntax trees and how it is built for C/C++ programming languages. We believe that this requirement is rarely met, and that is why the adoption of `ClangMR` can be challenging for a regular user.

Wasserman [13], on the other hand, introduces a tool (`Refaster`) that does not involve any DSLs. He suggests using the target project's programming language for describing transformations. This allows the user to embed transformation rules into the project's code base, which leads to simpler syntax checks and symbol availability validations. Transformation rules are written in the form of classes and methods with either `@BeforeTemplate` or `@AfterTemplate` annotations. Each class represents a transformation and should contain one or more `@BeforeTemplate` methods and a single `@AfterTemplate` method. Then the tool treats the transformation as follows: match the code that is written in `@BeforeTemplate` method and replace it with the code written in `@AfterTemplate` method.

We consider Wasserman's tool design to be clear and user-friendly because it uses the language of the project's code base to define transformations. We use a similar approach in `nobrainer`.

We decided that the best method for matching the C/C++ source code is the approach used in `ClangMR`. However, because using Clang AST matchers directly can be challenging, we provide a higher level framework that utilizes AST matchers internally.

Regarding the code transformation, a common solution is to generate an AST, transform it, and restore the source code in the end. This kind of approach is used by `Proteus` [5], `Jackpot` [7] and Eclipse C++ Tooling Plugin [6]. The main problem of implementing such an approach is code generation. We should remember all the nuances of the original source code in order to replicate them when restoring the resulting code. This includes preserving comments, redundant spaces, etc. On the other hand, in `ClangMR` [14], the authors suggest

using the `Clang`[3] framework for code transformation because it allows developers to directly modify the source code token wise. We also use the Clang framework because we believe it is the best solution to transform C/C++ source code.

3 Design

In this section, we describe the overall design and the user's workflow. Running the tool on a real project involves the following list of actions:

- writing transformation rules as part of the target project
- providing compilation commands for the target project (the currently supported format is the Clang compilation database[4])

`Nobrainer` either applies all the replacements or generates a YAML file containing these replacements. In the latter case, replacements can be applied later with the `clang-apply-replacements` tool (part of the Clang Extra Tools[5]).

Figure 1 provides an insight into the internal `nobrainer` structure. Each numbered block represents a work phase of the tool. Boxes at the bottom correspond to each phase's output.

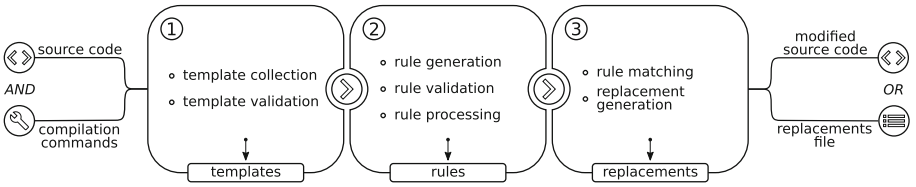

Fig. 1. Nobrainer workflow

During the first phase, the tool analyzes all of the translation units that are extracted from the given compilation commands. For each translation unit, it searches for and collects templates that represent our transformation examples. Then `nobrainer` filters invalid templates. The result of the first phase is a list of valid templates.

In the second phase, we group conforming templates into rules. `Nobrainer` also checks each rule for consistency and then processes each rule to generate internal template representation.

In the third phase, we work with the list of preprocessed rules. `Nobrainer` tries to match each rule against the project's source code. For each match, we construct a special data structure, which we use to generate a replacement. As a result, we obtain a set of replacements.

For more details on each phase, see Sect. 4.

[3] https://clang.llvm.org/.
[4] https://clang.llvm.org/docs/JSONCompilationDatabase.html.
[5] https://clang.llvm.org/extra/index.html.

4 Detailed Description

The core idea of `nobrainer` is the use of `examples`, which are code snippets written in C/C++ languages. Because each snippet may represent a whole family of cases, we call them *templates*. The user submits the situations she wants to change in the `Before` templates and the substituting code in the `After` templates. These templates can be defined anywhere in the project.

`Nobrainer` offers a special API for writing such examples, which is subdivided into C and C++ APIs. Both provide the ability to write `expression` and `statement` templates to match C/C++ expressions or statements respectively.

For a clearer explanation of what a template is, let us proceed with an example. Suppose the user wants to find all calls to function `foo` with an arbitrary `int` expression as the first argument and global variable `globalVar` as the second argument and replace the function with `bar`, while keeping all the same arguments. Listing 1 demonstrates how such a rule can be specified (using `nobrainer` C API).

```
int NOB_BEFORE_EXPR(ruleId)(int a) {
  return foo(a, globalVar);
}

int NOB_AFTER_EXPR(ruleId)(int a) {
  return bar(a, globalVar);
}
```

Listing 1: Expression template example

Expressions for matching and substitution reside inside of `return` statements. We force this limitation intentionally because it allows us to delegate the type compatibility check of `Before` and `After` expressions to the compiler.

`Nobrainer`'s transformations are based on the concept that two valid expressions of the same type are syntactically interchangeable. This statement is correct with the exception of parenthesis placement. In certain contexts, some expressions must be surrounded with parentheses. However, we introduce a simple set of rules that solve this issue and are not covered in this paper.

In order to properly define the term *template*, we first need to introduce the following notations (with respect to the given program):

- Θ is a finite set of all types defined
- Σ is a finite set of all defined symbols (functions, variables, types)
- $\mathcal{A}$ is a finite set of all AST nodes representing the program
- $\mathcal{C}$ is a finite set of characters allowed for C/C++ identifiers
- $\mathcal{P}$ is a finite set of all function parameters $p = \langle n_p, t_p \rangle$ where $n_p \in \mathcal{C}^*$ is the parameter's name and $t_p \in \Theta$ is its type.

An expression template can be formally defined as a 6-tuple

$$T_{expr} = \langle k, n, r, P, B, S \rangle \tag{1}$$

where

- $k \in \{\texttt{before}, \texttt{after}\}$ is the template's kind
- $n \in \mathcal{C}^*$ is the rule's identifier, it is used for pairing corresponding `before`/`after` templates
- $r \in \Theta$ is the return type
- $B \subset \mathcal{A}$ is the body
- $P \subseteq \mathcal{P}$ is the set of parameters
- $S \subseteq \Sigma$ is the set of symbols used in B.

The last two elements of the tuple require additional commentary.

Template parameters P represent generic placeholders for different situations encountered in the real source code. `Nobrainer` reads these parameters as arbitrary expressions of the corresponding type. For example, parameter `a` from Listing 1 corresponds to **any** expression of type `int`.

The set of symbols S is important for the correctness affirmation (see Sects. 4.4 and 4.6).

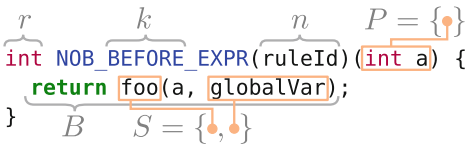

Fig. 2. `Before` template deconstruction

Figure 2 dissects the `Before` template from Listing 1.

The following subsections cover `nobrainer`'s phases in more detail.

4.1 Template Collection

The first phase is to collect all the templates from the project. `Nobrainer` scans each file and tries to find functions that were declared using the API. This can only be done for parsed source files. Doing so for the whole project can have a drastic impact on the tool's performance. In order to avoid checking all the files, we only process files that contain inclusion directives of `nobrainer` API header files.

As the output, this phase has a set of all templates defined by the user. We denote it as $\mathcal{T}$.

4.2 Template Validation

After the template collection phase, we validate each template individually. We need to check that the collected templates in $\mathcal{T}$ are structurally valid. First it is important to note that the syntactic correctness of a template is guaranteed by the compilation process. Templates are implemented as the part of the existing code base, which implies that they are actually parsed and checked during the

collection phase. This includes checks for the availability of all symbols, type checks, etc.

In every separate case, `nobrainer` replaces a single expression with another single expression. Considering this fact, each template T_{expr} should define *exactly one* expression. This requirement is transformed into a syntax form as: a template's body B should consist of a single return statement with a non-empty return expression. During the template validation stage, we check this constraint. Thus, `nobrainer` filters out templates without a body (i.e. declarations), templates with an empty body, and templates with a single statement `return;`.

Currently there are some limitations regarding the usage of functional style macros and the usage of C++ lambda expressions in template bodies. For this reason, we validate the absence of either of these language features.

Thus, if `nobrainer` encounters invalid templates, it filters them out and proceeds to the next phase with the set of valid ones $\mathcal{T}_+$.

4.3 Rule Generation

For an arbitrary $id \in \mathcal{C}^*$, we define two sets of templates B_{id} and A_{id} as follows:

$$B_{id} = \{T \in \mathcal{T}_+ | n_T = id, k_T = \texttt{before}\} \tag{2}$$

$$A_{id} = \{T \in \mathcal{T}_+ | n_T = id, k_T = \texttt{after}\} \tag{3}$$

These two groups describe exactly one user-defined transformation scenario because they include all of the *Before* and *After* examples under the same name. However, in order for B_{id} and A_{id} to form a transformation rule, the following additional conditions must be met:

$$\begin{cases} |B_{id}| \geq 1 \\ A_{id} = \{a_{id}\} \text{ (i.e. } |A_{id}| = 1) \\ \forall b \in B_{id} \rightarrow a_{id} \prec b \end{cases} \tag{4}$$

where

$$\forall x, y \in \mathcal{T}_+ \rightarrow x \prec y \Leftrightarrow \begin{cases} P_x \subseteq P_y \\ r_x = r_y \end{cases} \tag{5}$$

We refer to operator $\prec$ as the *compatibility operator*. It indicates that the snippet defined in x can safely replace the code matching y. The equality of return types r ensures that the substituting expression has the same type as the original one, while condition $P_x \subseteq P_y$ guarantees that `nobrainer` will have enough expressions to fill all of the x's placeholders.

As a result, we define *transformation rule* as a pair $R_{id} = \langle B_{id}, A_{id} \rangle$ where B_{id} and A_{id} meet conditions (4). Additionally we denote the set of all project rules as $\mathcal{R}$.

4.4 Rule Processing

Before Template Processing. As mentioned before, we convert `Before` templates to Clang AST matchers. These provide a convenient way to search for sub-trees that fit the given conditions. They describe each node, its properties, and the properties of its children. Thus, this structure resembles the structure of the AST itself. In order to generate matchers programmatically, we exploit this fact. Each node of the template's sub-tree is recursively traversed and paired with a matcher. As a result, we encapsulate the logic related to different AST nodes and avoid the necessity of supporting an exponential number of possible node combinations.

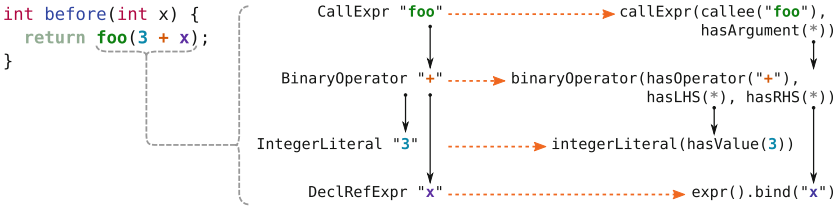

Fig. 3. Recursive AST matcher generation

Figure 3 demonstrates a simplified example of such a conversion. It depicts three stages of `Before` template processing: source code, AST, and AST matchers. Bold arrows reflect parent-child relationships, while dashed arrows stand for node-matcher correspondence. Because matchers are represented by a series of nested function calls, we construct the innermost matchers first, traversing the tree in a depth-first fashion.

Matching Identical Sub-Trees. Consider the `Before` template from Listing 2. It is unlikely that the user expects the system to match two arbitrary expressions as `foo`'s arguments. In fact, the most intuitive interpretation of this template is matching calls to function `foo` with identical arguments only.

```
int before(int x) {
  return foo(x, x);
}
```

Listing 2: An example of reusing a template parameter

Clang does not provide a matcher that can do the job. However, `nobrainer` already has a mechanism to find identical sub-trees for `Before` templates without parameters. During the matching process, we reuse this mechanism to dynamically generate a matcher. Thus, for the given example, we bind the first argument to `x`, generate a matcher, and check if the second argument fits.

After Template Processing. Our goal is to construct a text that represents the result of a replacement. Therefore, we convert *After* templates into plain strings. However, there are some parts of the *After* template's body that cannot be taken as is and placed into the desired location. We call such parts *mutable*. During the traversal of the *After* template's body, we extract the ranges that represent mutable parts. Each range consists of the start and the end locations of the certain AST node. There are two cases of *mutable* parts.

The first case is the use of a template parameter inside of an *After* template's body. We treat each template parameter as a placeholder that we fill during replacement generation (see Sect. 4.6).

The second case is the use of a symbol. Inserting symbols in arbitrary places in the source code can be syntactically incorrect. Indeed, in the location of insertion, the symbol may not yet have been declared. Thus, we collect symbol information that is used during replacement generation (see Sect. 4.6).

Given these points, for the *After* template from Listing 3, we construct the following format string: `"#{bar}(${x}) + 42"`. In this example, `nobrainer` distinguishes the symbol `bar` and the template parameter `x`, and handles them accordingly. The tool treats all the remaining parts of the string as immutable, and, with this in mind, constructs the resulting format string.

```
int after(int x) {
  return bar(x) + 42;
}
```

Listing 3: An example of an *After* template

4.5 Rule Application

The next step is to identify all situations, in which to apply rules $\mathcal{R}$. In order to do this across the whole project, `nobrainer` independently parses all the source files. After that, the tool applies AST matchers generated for each rule.

Each time a match is found, `nobrainer` obtains a top-level expression that should be replaced and a list of AST sub-trees bound to parameters from the corresponding *Before* template. Using this information and the *After* template, `nobrainer` generates an actual code change called a *replacement*.

4.6 Replacement Generation

Replacement is a sufficient specification for a complete textual transformation. It consists of four components:

- the file where current replacement is applied
- the byte offset where the replaced text starts
- the length of the replaced text
- the replacement text

`Nobrainer` extracts the first three components from the expression marked for substitution. The replacing text is composed from the *After* template and AST nodes bound to parameters. For each node, `nobrainer` gathers the corresponding source code and fills placeholders from the *After* template. This operation results in plain text for the substitution. Figure 4 demonstrates this procedure using a real code snippet.

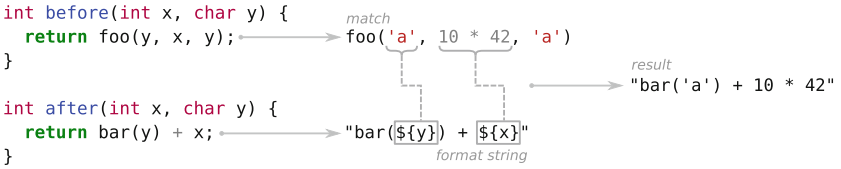

Fig. 4. Replacement generation

Such a transformation may nevertheless cause compilation errors due to symbol availability. `Nobrainer` should check that each symbol that comes with a substitution is declared and has all required name qualifiers. In order to ensure this, we:

- add inclusion directives for the corresponding header files
- add namespace specifiers.

The resulting code incorporates only the pieces of real source code that have been checked by Clang at different stages of the analysis.

4.7 Type Parameters

Parametrization with arbitrary expressions provides a flexible instrument for generic rule definition. However, this may not be enough. Exact type specification can significantly limit the rule's expressiveness and reduce the number of potential use cases.

In order to combat this shortcoming, we introduce a set of type parameters $\Phi \subset \mathcal{C}^*$ to a template syntax. This extends the template definition (1) to

$$T_{expr} = \langle k, n, r, P, B, S, \Phi \rangle \tag{6}$$

and compatibility operator $\prec$ (5) to

$$\forall x, y \in \mathcal{T}_+ \to x \prec y \Leftrightarrow \begin{cases} \Phi_x \subseteq \Phi_y \\ P_x \subseteq P_y \\ r_x = r_y \end{cases} \tag{7}$$

Note that type parameters Φ are fully symmetrical to parameters P.

```
template <class T> T *before() {
  return (T *)malloc(sizeof(T));
}

template <class T> T *after() {
  return new T;
}
```

Listing 4: An example of a type-parametrized rule

Listing 4 demonstrates a rule parametrized with type.

5 Results

In this section, we describe how we test `nobrainer`, provide some transformation rule examples and present the performance results.

5.1 Testing

Our tests can be divided into two main groups. First, we have a group of unit- and integration-tests for each phase described in Sect. 3. These are mainly used to check the correctness of AST matcher generation (Sect. 4.4) and format string generation (Sect. 4.4) for various AST nodes.

Second, we have a group of regression tests consisting of several open source projects.

For each project, we have created files with predefined `nobrainer` templates. Our testing framework runs the tool, measures the execution time, checks that all the predefined transformations have been performed as expected, and verifies that the project can be compiled afterwards.

5.2 Examples

In this section, we present three notable transformation rules that are supported by `nobrainer`.

The first example (Listing 5) shows the transformation rule that changes the order of arguments inside of the `compose` method call. Specifically, `nobrainer` will replace each call of the `compose` method of the `Agent` class `a.compose(x, y)` with the call `a.compose(y, x)`.

Thus, we demonstrate how to perform an argument swap automatically when method's signature changes.

```
int NOB_BEFORE_EXPR(ChangeOrder)(Agent a, char *x, char *y) {
  return a.compose(x, y);
}

int NOB_AFTER_EXPR(ChangeOrder)(Agent a, char *x, char *y) {
  return a.compose(y, x);
}
```

Listing 5: An example template for the argument swap

The second example (Listing 6) shows that `nobrainer` can be used to perform simplifying code transformations.

```
class EmptyCheckRefactoring : public nobrainer::ExprTemplate {
public:
  bool beforeSize(const std::string x) {
    return x.size() == 0;
  }

  bool beforeLength(const std::string x) {
    return x.length() == 0;
  }

  bool after(const std::string x) {
    return x.empty();
  }
};
```

Listing 6: An example template for a string emptiness check

Recall that each rule can have an arbitrary number of *before* templates, but only one *after* template. Writing several *before* expressions helps to group common transformations.

The third example contains type and expression parameters. Listing 7 shows the corresponding rule.

```
class ConstCastRefactoring : public nobrainer::ExprTemplate {
public:
  template <class T>
  T *before(const T *x) { return (T *)x; }

  template <class T>
  T *after(const T *x) { return const_cast<T *>(x); }
};
```

Listing 7: An example template for const casts

It detects the C-style cast that "drops" the `const` qualifier from the pointed type and replaces it with an equivalent C++-style cast. Parameter `x` should be of any pointer-to-const type and should be cast to exactly this type, but without a

`const` qualifier. `Nobrainer` captures all of these connections and processes them as expected.

5.3 Performance

To measure the performance we run our regression tests five times on a machine with Intel(R) Core(TM) i7-7700K CPU @ 4.20 GHz CPU, and 64 GB of RAM. The machine runs on Ubuntu 16.04 LTS. We perform the execution in eight threads.

Table 1 contains the results. For each project, we list its size in *lines of code*, the number of replacements `nobrainer` applies during the test, and our time measurements. We distinguish two stages of `nobrainer`'s workflow and measure them separately. The first stage incorporates the project's source code parsing. The second stage contains all the remaining computations up to replacement generation. We divide the whole process this way because the parsing process is performed by the *Clang* framework. For this reason, we can only minimize the time `nobrainer` spends in the second stage.

Table 1. Performance results

Project	KLOC	Replacements	Parsing time (s)	Remaining operation time (s)
CMake	493	24	31.36	7.13
curl	129	7	3.17	2.01
json	70	7	13.99	1.34
mysql	1170	10	9.54	3.12
protobuf	264	8	16.62	2.97
v8	3055	6	281.57	28.52
xgboost	43	14	6.75	1.18

It should be noted that the execution time does not directly correlate with the project's size. Other factors, such as translation unit sizes may also influence the overall performance.

As can be seen in Table 1, the elapsed time varies significantly between projects. In particular, this behavior applies both to the parsing time and to remaining processing time. Therefore, comparing the elapsed time of different projects offers few insights. Thus, in our evaluations, we consider the percentage of time that the file parsing takes from the whole process. Then, we compare this proportion among different projects. Figure 5 demonstrates the corresponding rates for the regression projects. Our results show that parsing takes up more than 81% of the whole execution time on average. For a global refactoring, all files must be parsed. The fact that the remaining procedure takes less than 20% of the execution time means that `nobrainer` has reached near-optimal performance.

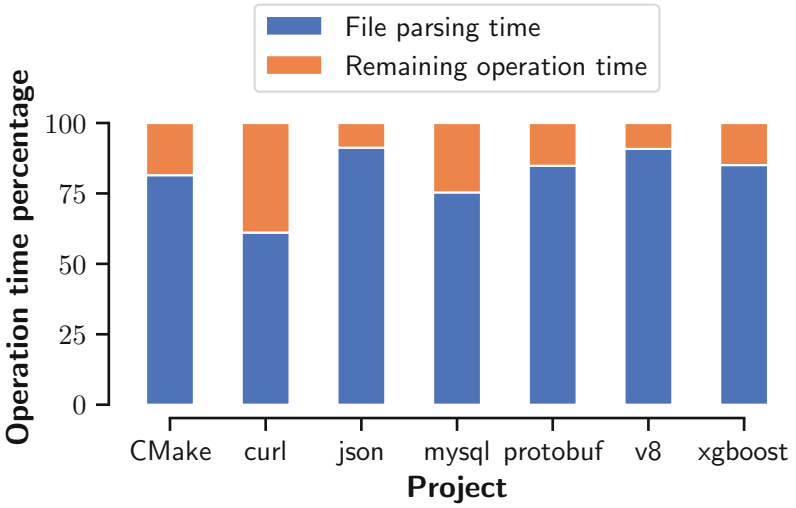

Fig. 5. File parsing percentage in `nobrainer` operation

Nevertheless, in certain cases, it is possible to avoid parsing files when it is sure that the file contains nothing to transform. The next section explains this and other directions in our future work.

6 Limitations and Future Work

Currently `nobrainer` supports expression templates and type parameterization. It can be used to perform transformations in continuous integration environments (CI). However, the execution time is still unsuitable for running it on large projects as a background task in IDE. There is still room for improvement. Thus, we consider three main directions for future work:

1. Full statement support
2. Performance improvements
3. Usability improvements

At the moment, we have already designed infrastructure for statement support. This includes API, validation and stubs for processing `Before` and `After` templates. We have also added support for `if` statements, `compound` statements, and variable declaration nodes. Our next task is to implement processing for each remaining statement node.

Regarding performance, we plan to research methods for reducing the parsing time. We are considering two directions. Firstly, we would like to improve the matching phase by skipping files that do not contain symbols used in `Before` templates. Secondly, we will explore automatic precompiled header (PCH) creation, which is expected to speed up the process of parsing the project's header files.

Further, the usability of our tool can be improved in two ways. Currently `nobrainer` performs found transformations only for the whole code base. We would like to add support for executing on user-defined parts of the project. We are also considering integrating with other developer tools. For example, `nobrainer` can be used as an IDE plugin to enhance user experience and the convenience of usage. Another possible scenario is to use `nobrainer` to assist static analyzers for fixing errors or defects.

7 Conclusion

In this paper, we presented `nobrainer`—a transformation and refactoring framework for C and C++ languages based on the Clang infrastructure. Its design is built on two main principles: ease-of-use and the extensive validation of transformation rules. A substantial part of this article includes describing its design and implementation, accompanied with examples and results.

Our results showed that `nobrainer` already supports real-world transformation examples and can be successfully applied to large C/C++ projects in continuous integration environments. We also highlighted the current limitations of the tool and some directions for later improvements. In the future, we plan to enhance the usability of `nobrainer` and integrate with other developer tools.

Acknowledgments. This work resulted from a joint project with Samsung Research. The authors of this paper are grateful to the colleagues from Samsung for their valuable ideas and feedback.

References

1. Clang documentation: Matching the clang AST. https://clang.llvm.org/docs/LibASTMatchers.html
2. Brown, N., et al.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010, pp. 47–52. ACM, New York (2010). https://doi.org/10.1145/1882362.1882373, http://doi.acm.org/10.1145/1882362.1882373
3. Cunningham, W.: The WyCash portfolio management system. SIGPLAN OOPS Mess. **4**(2), 29–30 (1992). https://doi.org/10.1145/157710.157715. http://doi.acm.org/10.1145/157710.157715
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (1999)
5. Waddington, D.G., Yao, B.: High-fidelity C/C++ code transformation. Electron. Notes Theoret. Comput. Sci. **141**, 35–56 (2007). https://doi.org/10.1016/j.entcs.2005.04.037
6. Graf, E., Zgraggen, G., Sommerlad, P.: Refactoring support for the C++ development tooling. In: OOPSLA Companion (2007)
7. Lahoda, J., Bečička, J., Ruijs, R.B.: Custom declarative refactoring in NetBeans: tool demonstration. In: Proceedings of the Fifth Workshop on Refactoring Tools, WRT 2012, pp. 63–64. ACM, New York (2012). https://doi.org/10.1145/2328876.2328886, http://doi.acm.org/10.1145/2328876.2328886

8. Meyerovich, L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. SIGPLAN Not. **48**(10), 1–18 (2013). https://doi.org/10.1145/2544173.2509515. http://doi.acm.org/10.1145/2544173.2509515
9. Murphy-Hill, E.R., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: ICSE, pp. 287–297. IEEE (2009). http://dblp.uni-trier.de/db/conf/icse/icse2009.html#Murphy-HillPB09
10. Pinto, G.H., Kamei, F.: What programmers say about refactoring tools?: An empirical investigation of stack overflow. In: Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools. WRT 2013, pp. 33–36. ACM, New York (2013). https://doi.org/10.1145/2541348.2541357, http://doi.acm.org/10.1145/2541348.2541357
11. Ray, B., Posnett, D., Devanbu, P., Filkov, V.: A large-scale study of programming languages and code quality in github. Commun. ACM **60**(10), 91–100 (2017). https://doi.org/10.1145/3126905. http://doi.acm.org/10.1145/3126905
12. Tracz, W.: Refactoring for software design smells: managing technical debt by Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. ACM SIGSOFT Softw. Eng. Notes **40**(6), 36 (2015). http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft40.html#Tracz15a
13. Wasserman, L.: Scalable, example-based refactorings with refaster. In: Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, pp. 25–28. ACM (2013)
14. Wright, H., Jasper, D., Klimek, M., Carruth, C., Wan, Z.: Large-scale automated refactoring using ClangMR. In: Proceedings of the 29th International Conference on Software Maintenance (2013)