ISSN 0361-7688, Programming and Computer Software, 2020, Vol. 46, No. 5, pp. 362–372. © Pleiades Publishing, Ltd., 2020.
Russian Text © The Author(s), 2020, published in Programmirovanie, 2020, Vol. 46, No. 5.

NOBRAINER: A Tool for Example-Based Transformation of C/C++ Code

V. V. Savchenko[a],*, K. S. Sorokin[a],, I. E. Bronshtein[a],***, A. S. Volkov[a],****, V. V. Kachanov[a],*****, G. A. Pankratenko[a],******, M. K. Ermakov[a],*******, S. I. Markov[a],********, A. V. Spiridonov[a],*********, and I. V. Aleksandrov[a],************

[a] *Ivannikov Institute for System Programming, Russian Academy of Sciences, Moscow, 109004 Russia*

**e-mail: vsavchenko@ispras.ru*
***e-mail: ksorokin@ispras.ru*
****e-mail: ibronstein@ispras.ru*
*****e-mail: asvolkov@ispras.ru*
******e-mail: vkachanov@ispras.ru*
*******e-mail: gpankratenko@ispras.ru*
********e-mail: mermakov@ispras.ru*
*********e-mail: markov@ispras.ru*
**********e-mail: aspiridonov@ispras.ru*
***********e-mail: ialexandrov@ispras.ru*

Received February 10, 2020; revised February 20, 2020; accepted March 15, 2020

Abstract—Refactoring is an integral part of the modern software development process. Often, the refactoring must be performed at the global level with modifications in a large number of files. Making these modifications is a long and painstaking work. However, users rarely employ automated tools for this purpose because they consider them unreliable and difficult to use. In this paper, a new tool for transforming the source code is described. It is based on the intuitively clear specification of transformation rules in the form of short code fragments in C or C++. These rules describe the code before and after the transformation. We believe that, due to the absence of additional abstractions (such as domain-specific languages), this approach can be easily used in practice. Even though the tool uses source code templates, it operates on the level of the abstract syntax tree. This enables the tool to better analyze the code and verify the validity of transformations.

DOI: 10.1134/S0361768820040052

1. INTRODUCTION

Any software product is evolving. The evolution in this case is not only the addition of new code implementing new functions but also continuous modification of the existing code. The excessive attention to the first aspect can result in the rapid accumulation of technical debt in the project. The *technical debt* [2] is a metaphor of software engineering that denotes simplifications in the architecture and code aimed at accelerating the initial development and deployment of the software product. If the technical debt is not paid off, it can, as time goes on, accumulate "interest"—the additional time needed for the developers to modify the program. In the worst case, the accumulated problems make the further development impossible.

A standard method of overcoming this difficulty is refactoring [1, 12], which is a modification of the internal program structure that does not affect its functionality [3]. Refactoring helps get rid of the existing architectural issues and simplify the program maintenance in future. According to Murphy-Hill estimates [8], software engineers spend 41% of their time on the refactoring-related activity. This paper also contains statistical data showing that developers prefer to make manual modifications of the code rather than use automated tools for program transformations even though the risk of making an error is higher in the former case. Another research performed on the site StackOverflow [9] showed that such tools are typically unreliable, difficult to use, and require a lot of additional actions from the user.

This enables us to formulate the minimal requirements for the refactoring tool that make it helpful. **It must be easy to use and ensure the correctness of transformations taking into account the available syntactic and semantic information.**

There are popular refactoring tools for C and C++, such as Proteus [4] and Eclipse C++ Tooling [5]. With these tools, the user can specify rules for code transformation using domain-specific languages (DSL). In these languages, it is possible to express both the required refactoring type and the syntactic and semantic structure of the target programming language.

However, there are difficulties in using DSL with C and C++. Studies show that learning C and C++ is more complicated than learning other popular programming languages [7], and it is easier to make errors in C and C++ [10]. As a result, taking into account the DSL, the difficulty of using refactoring tools drastically increases.

Thus, the definition of ease of use can be refined: **the refactoring tool should not require additional specific skills except for C/C++ knowledge.**

In this paper, we describe the tool Nobrainer designed for performing automatic transformations of the source code of C and C++ programs. It is based on the Clang/LLVM[1] infrastructure and meets the requirements described above. The name Nobrainer reflects its basic idea: this is a tool that allows users to easily create and apply their rules for code transformation.

Rules for Nobrainer are written in C/C++ without using DSL. They are indistinguishable from the ordinary code in the project, which allows the user to quickly master this tool.

Below, we describe the basic principles of Nobrainer, demonstrate the main architectural and technical solutions, and give examples of its use in industrial projects.

2. REVIEW OF EXISTING SOLUTIONS

This section is devoted to the existing approaches to code transformation and automated refactoring. In this review, we differentiate two key aspects—the form of the transformation description and the way the transformations are performed.

The majority of tools described in this section use a special syntax for specifying transformation rules. For example, in [4] a new language YATL is defined, and in [6] Java is extended to simplify the specification of such rules. We believe that DSLs can only confuse the user because of their complexity. The authors of ClangMR [14] propose another approach. Their tool uses Clang Abstract Syntax Tree (AST) Matchers [11]. They are needed for describing the parts of code that should be transformed. The user must describe the replacement of these parts in terms of the *abstract syntax tree* (AST). The authors assume that the users know syntax trees in general and how they are constructed for the C/C++ code in particular. We think that this is usually not the case, and the use of ClangMR can be difficult for ordinary users.

[1] https://clang.llvm.org/

Wasserman in [13] described the tool Refaster designed for refactoring Java codes that does not require any DSL. The author proposes to use the target programming language for specifying transformation rules. This makes it possible to include these rules into the project codebase, which simplifies the verification of their syntax and semantic constraints. Each rule is written in the form of a class that contains methods with one of the two annotations @BeforeTemplate or @AfterTemplate. Such a class describes a transformation; it must contain one or more methods with @BeforeTemplate and exactly one method with @AfterTemplate. The tool interprets such a description as follows: it uses the code in the methods with @BeforeTemplate for finding and matching, and then replaces the detected code with the code written in the method with @AfterTemplate. Since this approach seems to be most clear and convenient for end users, we accepted it as the basis for Nobrainer.

We also decided that the ClangMR approach is optimal for code search and matching. However, the direct use of AST matchers can be difficult. For this reason, we developed a higher level framework that in turn uses AST matchers.

In code transformation, the conventional solution is the construction of an AST, its transformation, and code generation. Such an approach is used in Proteus [4], Jackpot [6], and Eclipse C++ Tooling [5]. The main difficulty here is the preservation of the source code formatting at the stage of generation. However, the authors of ClangMR [14] managed to avoid this difficulty due to the use of the Clang/LLVM infrastructure for the code transformation. This allows the developers directly change the source code at the token level. We also use this infrastructure because we believe that this is the best solution for transforming the C/C++ codes.

3. ARCHITECTURE

In this section, we describe the general architecture and outline the Nobrainer operation.

The tool is based on using special examples—code fragments written in C/C++. Each example can describe a family of cases. This is why we denote/call them *templates*. First, the user provides an example of language constructs that should be replaced. It is called a Before template. Then, in an After template the user must provide the code that should replace the constructs matching the Before template. The descriptions of the Before and After templates may be added to any (e.g., a separate) file in the user project.

To run Nobrainer on a project, the user must do the following:

• add code transformation rules to the project;

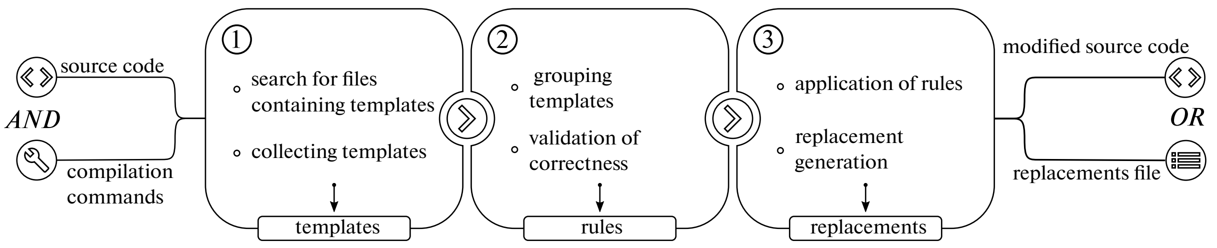


Fig. 1. Block diagram of Nobrainer operation.

• specify instructions for the project compilation (currently, instructions in the `JSON`[2] format from the `Clang/LLVM` infrastructure are used).

Figure 1 illustrates the `Nobrainer` internal structure. Each numbered block corresponds to a phase of the tool operation. The rectangular areas in the lower part of the figure show the data obtained at the output of each phase.

In the first phase, the `Before` and `After` templates are sought in the project code.

In the second phase, the list of transformation rules is made up by grouping the templates and validating their correctness.

In the third phase, the rules are applied and the replacements are generated. For each rule, the tool tries to match `Before` templates to the project source code. For the successful matches, replacements are generated using the `After` template.

`Nobrainer` makes it possible to immediately apply the replacements to the project files or save them in the `YAML`[3] format. In the latter case, the replacements can be later applied using the special tool `clang-apply-replacements` included in `Clang Extra Tools`.[4]

These phases are described in Section 4 in detail.

4. FORMAL DESCRIPTION AND IMPLEMENTATION DETAILS

`Nobrainer` provides API for writing templates. It is divided into the interface for C and the interface for C++. Both interfaces make it possible to write templates for transforming individual expressions and sequences of statements in the original files in the corresponding language.

To better explain the concept of a template, we consider an example of expression transformation. Assume that the user wants to find all calls of the function `foo` with two arguments. The first argument is an arbitrary expression of type `int`. The second argument is the global variable `globalVar`. These calls should be replaced by the calls of the bar function with the same arguments. Listing 1 shows an example of description of such a rule using the C interface.

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

Listing 1. Example of templates for expression transformation.

The expressions for matching and replacing are described within the `return` statement. This is done to delegate the verification of the compatibility of expression types in `Before` and `After` templates to the compiler.

In `Nobrainer`, transformations are based on the fact that two expressions of the same type are syntactically interchangeable. This is true except for certain situations in which the expressions must be parenthesized. There is a set of special rules for overcoming this issue; however, they are out of the scope of this paper.

To give a formal definition of template in a program, we introduce the following notation:

• Θ is the finite set of all types in the program;

• Σ is the finite set of all symbols (functions, variables, and types) declared in the program;

• $\mathcal{A}$ is the finite set of all nodes in the program's AST;

• $\mathcal{C}$ is the finite set of symbols that may be used in identifiers in C/C++;

• $\mathcal{P}$ is the finite set of all parameters of a function, $p \in \mathcal{P}$ and $p = \langle n_p, t_p \rangle$, where $n_p \in \mathcal{C}^*$ is the parameter name and $t_p \in \Theta$ is its type.

The *expression template* can be formally described as the 6-tuple

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

where

• $k \in \{$`Before`, `After`$\}$ is the template type;

• $n \in \mathcal{C}^*$ is the rule identifier used to link the `Before` and `After` templates;

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

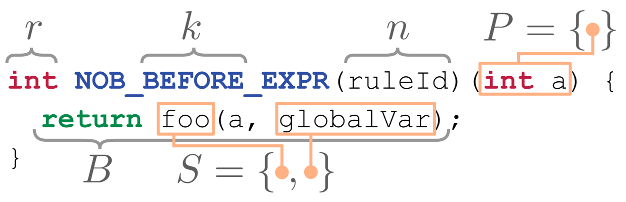

r
k
n
P = {●}
int NOB_BEFORE_EXPR(ruleId) (int a) {
return foo(a, globalVar);
}
B
S = {●, ●}

Fig. 2. Parsing the `Before` template.

- $r \in \Theta$ is the expression type;
- $B \subset \mathcal{A}$ is the template body;
- $P \subseteq \mathcal{P}$ is the set of parameters;
- $S \subseteq \Sigma$ is the set of symbols used in the body B.

The last two elements require a clarification.

The parameters of the template P are used to express special semantics. `Nobrainer` considers each parameter as an arbitrary expression of the corresponding type. For example, the parameter a in Listing 1 corresponds to any expression of type `int`.

The set of symbols S is used for correctness verification (see Subsections 4.2.3 and 4.3.2).

Figure 2 shows the detailed structure of the `Before` template in Listing 1.

In the following subsections, the phases of `Nobrainer` operation are considered in more detail.

4.1. Search for Templates

In the first phase, all the templates defined in the project are collected and their validity is verified.

4.1.1. Collecting templates. `Nobrainer` examines each file and tries to find the functions that were earlier defined using the Nobrainer API. This search can be made only in the parsed files. If this procedure were performed for the entire project, it would significantly decelerate the tool operation. This can be avoided by processing only the files containing the `#include` directives with `Nobrainer` API header files.

As a result, the set of all templates defined by the user is obtained. We denote this set by $\mathcal{T}$.

4.1.2. Verification of template validity. When all templates are collected, Nobrainer proceeds to verifying the validity of each template. It must also check that the templates in $\mathcal{T}$ have a correct structure. It is important that the syntactic validity of templates is guaranteed by the compiler. The template declarations are a part of the project code and, therefore, they are parsed during the search. This also includes the verification of accessibility of all symbols used in the template, type verification, etc.

Each template T_{expr} must define *exactly one* expression. Formally, this rule can be formulated as follows: The template body B must consist of a single non-empty `return` statement.

Currently, Nobrainer ignores the templates that use the functional style macros and C++ lambda expressions. This is a temporary limitation, and it will be later removed.

The result of this phase is the set of valid (from the viewpoint of rules described above) templates, which we denote by $\mathcal{T}_+$.

4.2. Matching the List of Transformation Rules

For an arbitrary identifier $id \in \mathcal{C}^*$, we define two groups 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 a unique code transformation because they include the `Before` and `After` templates with the same name. However, in order for B_{id} and A_{id} to specify a transformation rule, the two following conditions must be additionally satisfied:

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

where

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

Here $\prec$ is the *compatibility operator.* It specifies the relation between the templates x and y that indicates that the fragment of code matching y can be safely replaced by the code of x. The equality of the returned types r guarantees that the expression to be replaced has the same type as the original expression. At the same time, the condition $P_x \subseteq P_y$ ensures that the tool has a sufficient number of expressions for substituting the parameters in x.

Thus, we can define the *transformation rule* as a pair $R_{id} = \langle B_{id}, A_{id} \rangle$, where B_{id} and A_{id} satisfy conditions (4). The set of all rules defined in the project is denoted by $\mathcal{R}$.

4.2.1. Processing the `Before` templates. In this phase of `Nobrainer` operation, the Before templates are transformed into AST matchers. The latter are convenient for finding subtrees that satisfy the given conditions. Each AST matcher consists of predicates for the subtree root and predicates of all its nodes. Such a structure resembles the syntax tree itself, and `Nobrainer` uses this fact for the AST matcher generation. We recursively traverse all nodes of the tree constructed for the body of `Before`, and generate an AST matcher for each node type. In addition, special AST matchers are created for the type parameters; they will link the subtrees to be matched with the parameter name. As a result, the final "tree" consisting of AST matchers is constructed. Therefore, it is sufficient to

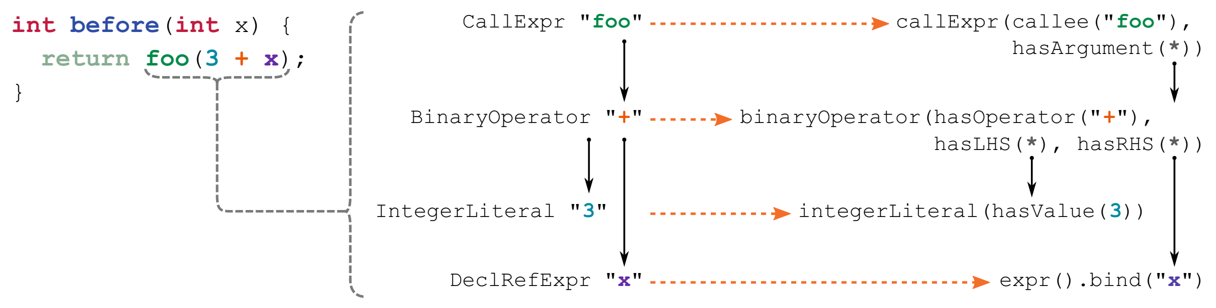

int before(int x) {
return foo(3 + x);
}
CallExpr "foo"
callExpr(callee("foo"), hasArgument(*))
BinaryOperator "+"
binaryOperator(hasOperator("+"), hasLHS(*), hasRHS(*))
IntegerLiteral "3"
integerLiteral(hasValue(3))
DeclRefExpr "x"
expr().bind("x")

Fig. 3. Recursive generation of an AST matcher.

organize the generation of an AST matchers for each node type.

Figure 3 illustrates a simplified example of such a transformation. In this example, we see three representations of `Before`: the original code, the syntax tree, and AST matcher. Solid arrows lead from the parent to the child AST node and the dashed arrows indicate the correspondence between syntax tree nodes and AST matchers. AST matchers must be constructed from the bottom to the top; therefore, the AST is parsed depth first.

4.2.2. Matching identical subtrees. Consider the `Before` template in Listing 2. It is unlikely that the user wants to find a call of foo with two arbitrary expressions of type int as its arguments. A more probable interpretation as follows: find a call of foo into which two identical expressions are passed.

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

Listing 2. Example of reusing a template parameter.

The `Clang/LLVM` infrastructure has no ready-to-use solution for matching identical subtrees. Using a mechanism available in `Nobrainer`, we dynamically generate AST matchers in the process of matching. This is done as follows. When the first argument of the `foo` call is matched, the corresponding subtree is associated with the parameter `x`. Next, using this subtree, an AST matcher is generated and then used for matching with the second argument of the call.

4.2.3. Processing the `After` *templates.* The aim of processing the `After` template is to construct the text that will be used as the replacement. For this reason, we transform the `After` templates into format strings. The template body *B* may contain elements for which it is impossible to simply take the text representation as is . Such parts are called *mutable.* During the traversal of the body of `After`, we extract the ranges corresponding to the mutable parts. Each range contains the initial and the final positions of a certain node of the syntax tree. There are two types of *mutable* parts.

The parts of the first type are the parameters in the body of `After`, which are filled during the generation of replacements (see Subsection 4.3.2).

The parts of the second type are symbols (identifiers that are not the parameters of the template). Inserting symbols into arbitrary places of the source code can be syntactically incorrect because this symbol can be undeclared in the place where it should be inserted. For this reason, we collect information about the symbols that is later used during the generation of replacements (see Subsection 4.3.2).

For example, for the `After` template in Listing 3, the format string `"${bar}($[x]) + 42"` will be generated. In this example, Nobrainer selects the symbol `bar` and the parameter `x` and labels them accordingly. The other parts of the string are considered as immutable by `Nobrainer`.

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

Listing 3. Example of an `After` template.

4.3. Application of Rules and Generation of Replacements

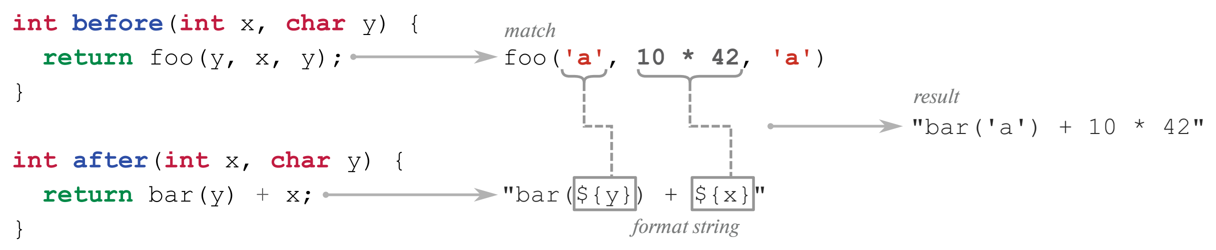
4.3.1. Application of rules. At the next step, we must detect the situations in which the rules $\mathcal{R}$ should be applied. In order to do this in the entire project, `Nobrainer` parses all the source files. Then, it applies the AST matchers generated for each rule.

Each time when an AST matcher finds a node in the syntax tree satisfying all the predicates, `Nobrainer` gets this node and the list of subtrees associated with the parameters of the corresponding `Before` template. Using this information and the `After` template, `Nobrainer` generates the transformation of the source code called *replacement.*

4.3.2. Generation of replacements. Each replacement includes:

• the file name to which the replacement is applied;

• the offset of the text to be replaced from the beginning of the file;



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

int after(int x, char y) {
  return bar(y) + x;
}
```

```
foo('a', 10 * 42, 'a')
"bar(${y}) + ${x}"
"bar('a') + 10 * 42"
```

Fig. 4. Replacement generation.

• the length of text to be replaced;

• the replacement text.

`Nobrainer` gets the first three elements from the associated node. The replacement text is constructed on the basis of the format string for `After` and the subtrees associated with the template parameters. For each such subtree, `Nobrainer` gets its text representation from the source code and then replaces the corresponding parameter by this substring in the format string. Figure 4 illustrates the replacement text generation for a real-life code fragment.

However, such a replacement can cause compilation errors because certain symbols appearing as the result of the replacement can be undeclared or miss namespace specifiers. To avoid these errors, `Nobrainer` can:

• add the directives `#include` for the header file in which this symbol is declared;

• specify the namespace or scope.

Therefore, the resulting code fragment is correct.

4.4. Type Parameters

The use of arbitrary expressions as parameters of the templates makes it possible to specify rules in a generalized form. However, this can be insufficient. The indication of specific types in a rule can significantly restrict its expressive power and narrow its field of application.

To overcome this drawback, we introduce the set of type parameters $\Phi \subset \mathscr{C}^*$ into the template syntax Thus, we extend the definition of template for replacing expression (1) to obtain

$$T'_{expr} = \langle k, n, r, B, P, S, \Phi \rangle, \qquad (6)$$

and the definition of compatibility $\prec$ (5) becomes

$$\forall x, y \in T_+ \to x \prec y \Leftrightarrow \begin{cases} \Phi_x \subseteq \Phi_y \\ P_x \subseteq P_y \\ r_x = r_y. \end{cases} \qquad (7)$$

Note that the type parameters Φ are completely analogous to the parameters P.

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

Listing 4. Example of a rule with a type parameter.

Listing 4 shows a rule in which type parameters are used.

4.5. Transformation of a Sequence of Statements

`Nobrainer` can be used to transform not only individual expressions but sequences of statements as well. To make such transformations correct, we extend the definition of the template T'_{expr} (6) to T_{stmt} as follows:

$$T_{stmt} = \langle k, n, r, P, B, S, \Phi, D \rangle; \qquad (8)$$

here D is the finite set of all declarations of local variables of the function, $d \in D$, and $d = \langle n_d, t_d \rangle$, where $n_d \in \mathscr{C}^*$ is the parameter name and $t_d \in \Theta$ is its type. We also strengthen the definition of the compatibility operator $\prec$ (7):

$$\forall x, y \in \mathscr{T}_+ \to x \prec y \Leftrightarrow \begin{cases} \Phi_x \subseteq \Phi_y \\ P_x \subseteq P_y \\ D_x = D_y \\ r_x = r_y. \end{cases} \qquad (9)$$

Thus, all local variables in the templates T_{stmt} are considered as parameters. We additionally require that the user is not allowed to add and remove declarations of variables. However, the scopes of local variables are ignored in definition (9), and the tool cannot guarantee that the code will successfully compile after the replacement. This is a temporary limitation. Listing 5 shows an example in which an uncompilable code can be produced if the counter is used outside the loop.

```
void before() {
  int i;
  for (i = 0; i < 10; ++i) {
    foo(i);
  }
}
void after() {
  for (int i = 0; i < 10; ++i) {
    foo(i);
 }
}
```

Listing 5. Example of an incorrect template with scope changing.

The templates T_{stmt} also allow an empty body in the `After` function, which allows the user to remove the code matching a `Before` template.

4.6. Matching an Arbitrary Statement

Using transformations of a sequence of statements, the code structure as a whole can be changed. For this reason, it would be convenient to generalize the transformation rules. To this end, we added a special function `anystmt` to the `Nobrainer` interface. Its call is matched to any statement. We demonstrate the application of `anystmt` by way of the following example (Listing 6).

```
void before(nobrainer::Name x) {
  if (foo())
    anystmt(x);
}
void after(nobrainer::Name x) {
  if (!foo())
    x;
}
```

Listing 6. Example of using `anystmt`.

Here, we want to invert the condition in the conditional statement calling the function `foo`.

It is seen that new parameters of the type `nobrainer::Name` now appear in the lists of arguments. This is a custom type that we included in the `Nobrainer` interface. In the `Before` template, the parameter of this type is associated with an arbitrary statement matching the call of `anystmt`. This makes it possible to use the statement matching the corresponding parameter in the `After` template.

To match an arbitrary statement in the body of `Before`, one must call the function anystmt with the single parameter of the type `nobrainer::Name`. To use the statement matching anystmt in `After`, the parameter name associated with it must be specified (Listing 6).

To ensure this functionality, we introduced an additional AST matcher that matches the call of the function `anystmt` to an arbitrary statement and associates the name of the unique parameter of this call with this statement. Note that, in the current implementation, the substitution of the statement associated with the parameter of the `nobrainer::Name` type into the `After` template causes a compiler warning *Unused expression result*. Both issues will be resolved in the near future.

4.7. Matching an Arbitrary Sequence of Statements

To extend the capabilities of the code structure transformation, we introduced two special functions `block` and `block_greedy`. Their calls are matched to an arbitrary (including empty) sequence of statements, and they work similar to the lazy (.*?) and greedy (.*) quantifiers in regular expressions, respectively. The functions `block` and `block_greedy` take a single parameter of the type `nobrainer::Name` as their argument and associate its name with the matched sequence of statements. In order to use this sequence in the `After` template, it suffices to write this name similarly to the mechanism for `anystmt`.

This functionality is implemented using a special AST matcher. The internal implementation of this AST matcher is based on an algorithm described in [15]; however, its description is out of the scope of the present paper. In the following example, we demonstrate the difference in using the lazy (`block`) and greedy (`block_greedy`) interfaces (Listings 7 and 8).

Note that currently only one use of each parameter of the type `nobrainer::Name` used with `block` or `block_greedy` in the body of `Before` is allowed.

```
void before(Name x) {
  block(x);
  foo();
}
void after(Name x) {
  x;
}
void example() {
                 // First match:
                 // <= block(x);
  foo();    // <= foo();
                // Second match:
  bar(); // <= block(x):
  foo(); // <= foo();
              // Third match:
  bar(); // <= block(x);
  foo(); // <= foo();
}
```

Listing 7. Example of using `block`.

```
void before(Name x) {
  block_greedy(x);
  foo();
}
void after(Name x) {
  x;
}
void example() {
         // Unique match:
         //
  foo(); // <=*
  bar(); // *
         // * block(x);
  foo(); // *
  bar(); // <=*
         //
  foo(); // <= foo();
}
```

Listing 8. Example of using block_greedy.

In the case when nobrainer::Name is associated with an empty sequence of statements, the empty string will be substituted for the corresponding parameter when the replacement for the After template is created.

4.8. Unsafe Transformations

Until now, we tried to ensure that all transformations are completely correct, which sometimes significantly restricts the capabilities of using Nobrainer. For this reason, we introduce one more type of templates T_{unsafe}. Its definition does not differ from the definition of the template T_{stmt} (8), but it has a weaker compatibility operator $\prec$:

$$\forall x, y \in \mathcal{T}_+ \to x \prec y \Leftrightarrow \begin{cases} \Phi_x \subseteq \Phi_y \\ P_x \subseteq P_y \\ r_x = r_y. \end{cases} \quad (10)$$

We no longer require that the sets D_x and D_y are identical. This, in turn, makes it possible to delete local variables and change their type in the After template. With the introduction of unsafe After, the implementation of the following replacement (Listing 9) becomes possible:

```
void before_unsafe(char *s) {
  for (int i = 0; i < strlen(s); ++i) {
    foo(s + i);
  }
}
void after_unsafe(char *s) {
  int len = strlen(s);
  for (int i = 0, i < len; ++i) {
    foo(s + i);
  }
}
```

Listing 9. Example of using an unsafe template.

However, the responsibility for the compilation errors related to changes of the local declarations as a result of replacements is with the user.

5. RESULTS

In this section, we describe an approach to testing Nobrainer; we consider examples of transformation rules and analyze the performance.

5.1. Testing

Tests for Nobrainer can be divided into two groups. The first group consists of unit and integration tests for each phase described in Section 3. They are mainly used for checking the correctness of automatically generated AST matchers for the templates Before (Subsection 4.2.1) and format strings for the templates After (Subsection 4.2.3).

The second group consists of regression tests, which include a number of open source projects. For each project, we manually created files with transformation rules. The regression testing system runs Nobrainer, measures its execution time, and checks that all the transformations have been successfully applied and the project compiles.

5.2. Examples of Rules

In this subsection, we discuss examples of the transformation rules supported by Nobrainer.

The first example (Listing 10) describes a rule that changes the order of arguments in all calls of the method compose. Nobrainer replaces each call of a.compose(x, y) for an arbitrary object a of the class Agent by a.compose(y, x).

This example demonstrates how the call arguments can be automatically exchanged.

```
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 10. Example of a rule for changing the order of arguments.

The second example (Listing 11) shows how Nobrainer can be used to simplify the source code.

```
using namespace nobrainer;
class EmptyRefactoring : public 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 11. Example of a rule for checking if a string is empty.

Recall that each rule may contain several `Before` templates but only one `After` template. In this case, the use of several `Before` templates helps group logically connected transformations.

The third example uses two types of parameters—expression and type parameters. Listing 12 shows the source code of the transformation rule.

```
using namespace nobrainer;
class CastRefactoring : public 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 12. Example of a rule for casting to a nonconstant type.

This rule finds all operations of casting to a nonconstant type in C and replaces them by equivalent `const_cast` expressions. In this case, the parameter `x` is an arbitrary expression of the type pointer to any nonconstant type. This expression should be replaced by the expression of the same type but without the `const` qualifier. `Nobrainer` takes into account all these requirements and makes the replacements correctly.

The fourth example illustrates the simplification of the `if` statement condition using `anystmt` (Listing 13).

```
using namespace nobrainer;
class SwapBranches : public StmtTemplate {
  void before(Name x, Name y, bool cond) {
    if (!cond)
      anystmt(x);
    else
      anystmt(y);
  }
  void after(Name x, Name y, bool cond) {
    if (cond)
      y;
    else
      x;
  }
};
```

Listing 13. Example of a rule with `anystmt`.

Table 1. Performance

Project	Size (thousands of lines)	Number of replacements	I phase (s)	II phase (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

This rule removes negation in the condition and exchanges the branches in `if`.

The fifth example implements an unsafe transformation (Listing 2). Here we change the type of the loop variable.

```
using namespace nobrainer;
class ChangeType : public UnsafeStmtTemplate {
  void before(const char *s, Name x) {
    for (int i = 0; i < strlen(s); ++i) {
      block(x);
    }
}
  void after(const char *s, Name x) {
    for (size_t i = 0; i < strlen(s); ++i) {
       x;
    }
  }
};
```

Listing 14. Example of a rule with an unsafe template.

5.3. Performance

The performance was measured by running the regression testing system five times. For this purpose, the work station with an Intel(R) Core(TM) i7-7700K CPU running at the clock rate of 4.20GHz with 64 Gb of RAM under OS Ubuntu 16.04 LTS was used.

Table 1 presents the results of performance measurements. For each project, its size (in thousands of code lines), number of replacements applied by `Nobrainer`, and its execution time for this project are specified. The execution time is presented for two phases measured individually. The first phase is the parsing of the project's source code. The second phase includes the main logics of `Nobrainer` (see Section 3).

It is seen from Table 1 that the execution time strongly depends on the project. In particular, this is the case for both phases. Figure 5 additionally shows the percentage of time spent on parsing for each project involved in regression testing. The results show that parsing takes 81% of the execution time on the average. The fact that the time needed to execute the remaining operations is about 20% indicates that the performance of `Nobrainer` is close to optimal.

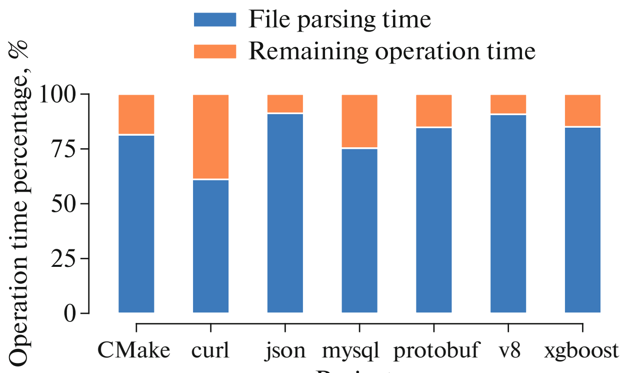


Fig. 5. Time percentage spent on parsing.

6. FURTHER RESEARCH

Currently, `Nobrainer` supports replacement rules for expressions and statements. In addition, it supports rules with type parameters. Nobrainer can already be used in continuous integration (CI) systems; however, its performance prevents using it interactively in large projects. On the whole, three basic directions of research can be distinguished:

1. support of the still unsupported AST nodes corresponding to C/C++ expressions and statements;

2. performance improvements;

3. usability improvements.

To improve the performance, we are going to research possible approaches to reducing the parsing time. There are two promising options. The first one is the optimization of the matching phase by ignoring the files not containing symbols from `Before` templates. The second one is the automatic generation of precompiled header files (PCH). These files should decrease the time needed to parse the header files in the project.

There is a number of approaches to improving the usability. Presently, `Nobrainer` processesall files in the project. It would be desirable to allow the user to specify the parts of the projects to be transformed. We also consider the integration with other developer tools. For example, Nobrainer can be used as an IDE plugin to enhance user experience and the convenience of usage. Another approach is to use `Nobrainer` as an auxiliary programming tool. For example, it is possible to try using it for automatically fixing bugs and defects found by a static analyzer.

7. CONCLUSIONS

In this paper, we presented Nobrainer—a tool designed for making automatic transformations of C and C++ programs. It is based on two principles—the simplicity of use and validity of transformation rules. Special attention was given to the description of the model, architecture, and implementation details with the justification of adopted solutions, results, and examples.

Ourresults show that Nobrainer can already be used in continuous integration systems for performing transformations in large projects. We also described the existing drawbacks , and directions for improvement. In the future, we are going to enhancethe usabilityof Nobrainer and integrate it with other software development tools.

REFERENCES

1. Brown, N., Cai, Y. Guo, Y., Kazman, R. Kim, M., Kruchten, P. Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., and Zazworka, N., Managing technical debt in software-reliant systems, *Proc. of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER'10,* New York, 2010, pp. 47–52.
2. Cunningham, W., The WyCash portfolio management system, *SIGPLAN OOPS Mess.,*1992, vol. 4, no. 2, pp. 29–30.
3. Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., Refactoring: Improving the design of existing code, Addison-Wesley Professional, 1999.
4. Waddington, D. G. and Yao, B., High-fidelity C/C++ code transformation, *Electron. Notes Theor. Comput. Sci.,* 2007, vol. 141, no. 1, pp. 35–56.
5. Graf, E., Zgraggen, G., and Sommerlad, P., Refactoring support for the C++ development tooling, *OOPSLA Companion,* 2007.
6. Lahoda, J., Bečička, J., and Ruijs, R.B., Custom declarative refactoring in NetBeans: Tool demonstration, *Proc. of the Fifth Workshop on Refactoring Tools, WRT'12,* New York, 2012, pp. 63–64.
7. Meyerovich, L.A. and Rabkin A.S., Empirical analysis of programming language adoption, SIGPLAN Notices, 2013, vol. 48, no. 10, pp. 1–18.
8. Murphy-Hill, E., Parnin, C., and Black, A.P., How we refactor, and how we know it, *ICSE,* 2009, pp. 287–297.
9. Pinto, G. H. and Kamei, F., What programmers say about refactoring tools?: An empirical investigation of stack overflow, *Proc. of the 2013 ACM Workshop on Refactoring Tools, WRT'13,* New York, 2013, pp. 33–36.
10. Ray, B., Posnett, D., Devanbu, P., and Filkov, V., A Large-scale study of programming languages and code quality in GitHub, *Commun. ACM,* 2017, vol. 60, no. 10, pp. 91–100.
11. The Clang Team. Clang documentation: Matching the Clang AST. https://clang.llvm.org/docs/LibASTMatchers.html
12. Tracz, W., Refactoring for software design smells: Managing Technical Debt by Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma, *ACM SIGSOFT Software Eng. Notes,* 2015, vol. 40, no. 6, p. 36.
13. Wasserman, L., Scalable, example-based refactorings with Refaster, *Proc. of the 2013 ACM Workshop on Refactoring Tools,* 2013, pp. 25–28.
14. Wright, H., Jasper, D., Klimek, M., Carruth, C., and Wan, Z., Large-scale automated refactoring using ClangMR, *Proc. of the 29th International Conference on Software Maintenance,* 2013.
15. Friedl, J.E.F., *Mastering Regular Expressions,* Cambridge: O'Reilly, 1997.

Translated by A. Klimontovich