ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 33–46

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.421.6

NOBRAINER: ИНСТРУМЕНТ ПРЕОБРАЗОВАНИЯ C/C++ КОДА НА ОСНОВЕ ПРИМЕРОВ

© 2020 г. В. В. Савченко[a,*], К. С. Сорокин[a,], И. Е. Бронштейн[a,***], А. С. Волков[a,****], В. В. Качанов[a,*****], Г. А. Панкратенко[a,******], М. К. Ермаков[a,*******], С. И. Марков[a,********], А. В. Спиридонов[a,*********], И. В. Александров[a,**********]**

[a] *Институт системного программирования РАН им. В.П. Иванникова*
109004 Москва, ул. Александра Солженицына, д. 25, Россия

**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*

Поступила в редакцию 10.02.2020 г.
После доработки 20.02.2020 г.
Принята к публикации 15.03.2020 г.

Рефакторинг – это неотъемлемая часть современного процесса разработки. Часто рефакторинг требуется выполнять на глобальном уровне с модификациями в большом количестве файлов. Внесение таких изменений вручную – долгая и кропотливая работа. Однако пользователи редко применяют автоматические инструменты для этих целей, так как считают их ненадежными и сложными в использовании.

В этой статье представлен новый инструмент для выполнения трансформаций исходного кода. Он основан на интуитивной схеме задания пользователем правил преобразования в виде коротких фрагментов кода на языке C или C++. Правила описывают то, как код должен выглядеть до и после трансформации. Мы убеждены, что благодаря отсутствию дополнительных абстракций (таких как предметно-ориентированные языки), указанный подход гораздо проще применить на практике.

Несмотря на использование примеров с исходным кодом, представленный инструмент оперирует на уровне абстрактного синтаксического дерева. Это позволяет ему лучше анализировать пользовательский код и проверять корректность трансформаций.

DOI: 10.31857/S0132347420040056

1. ВВЕДЕНИЕ

Любой программный продукт постоянно эволюционирует. Эволюция в данном случае – это не только появление нового кода при расширении функциональности, но и непрерывный процесс модификации существующего. Чрезмерное внимание к первому может привести к быстрому накоплению технического долга в рамках проекта. *Технический долг* [2] – это метафора программной инженерии, обозначающая упрощения в архитектуре и коде, позволяющие ускорить первоначальные разработку и развертывание программного продукта. С течением времени, если технический долг не “выплачивать”, он может обрастать “процентами” – дополнительным временем, которое разработчики потратят на изменение программы. В худшем случае накопленные проблемы могут привести к невозможности продолжить разработку.

Стандартным методом борьбы с этим является *рефакторинг* [1], [12] – изменение внутренней структуры программы, которое при этом не влияет на ее функциональность [3]. Он помогает избавиться от существующих архитектурных проблем и упростить сопровождение программы в будущем. По оценкам Мерфи-Хилла и др. [8], 41% времени программисты тратят на деятельность,

связанную с рефакторингом. В той же работе приведены статистические данные, которые показывают, что разработчики предпочитают править код вручную, а не использовать автоматические инструменты для трансформации программ несмотря на то, что риск допустить ошибку в первом случае выше. Другое исследование, проведенное на сайте StackOverflow [9], показало, что такие инструменты как правило ненадежны, сложны в использовании и требуют слишком большого количества дополнительных действий от самого пользователя.

Все это позволяет сформулировать минимальные требования к инструменту рефакторинга, который можно считать полезным: **он должен быть прост в использовании и обеспечивать корректность проводимых трансформаций с учетом имеющейся синтаксической и семантической информации**.

Для языков C/C++ существуют популярные инструменты рефакторинга, такие как `Proteus` [4] и `Eclipse C++ Tooling` [5]. С их помощью пользователь может задавать правила для трансформации кода, используя *предметно-ориентированные языки* (domain-specific languages, далее – DSL). На таких языках можно выразить и то, какой рефакторинг требуется, и синтаксическую/семантическую структуру целевого языка программирования.

Однако с применением DSL к C/C++ есть проблемы. Исследования показывают, что по сравнению с другими популярными языками программирования обучение языкам C/C++ труднее [7], при этом в C/C++ проще допустить ошибку [10]. Получается, что с учетом DSL сложность использования инструмента рефакторинга многократно возрастает.

Таким образом, определение простоты использования можно уточнить: **инструмент рефакторинга не должен требовать от пользователя дополнительных узкоспециализированных знаний помимо знания самого C/C++**.

В данной статье представлен инструмент `Nobrainer`, предназначенный для проведения автоматических преобразований исходного кода программ на языках C/C++. Он основан на инфраструктуре `Clang/LLVM`[1] и соответствует всем вышеописанным требованиям. Название `Nobrainer` отражает его ключевую идею: это инструмент, который позволяет пользователям легко создавать и применять собственные правила для трансформации кода.

Правила для `Nobrainer` пишутся на C/C++ без использования DSL. Они неотличимы от обычного кода из проекта, что позволяет освоить инструмент в короткие сроки.

Далее мы опишем базовые принципы Nobrainer, продемонстрируем основные архитектурные и технические решения, а также приведем примеры его использования на промышленных проектах.

2. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Эта глава посвящена существующим подходам к трансформации кода и автоматическому рефакторингу. В рамках обзора мы будем разделять два ключевых аспекта: форму описания трансформаций кода и то, каким способом эти трансформации производятся.

В большинстве описываемых здесь инструментов для правил трансформации кода используется собственный синтаксис. Например, в работе Уоддингтона (Waddington) и др. [4] вводится новый язык YATL, тогда как Лагода (Lahoda) и др. [6] расширяют язык Java, чтобы упростить написание таких правил. Мы считаем, что всевозможные DSL могут только запутать пользователя из-за своей дополнительной сложности. Авторы `ClangMR` [14] предлагают другой подход. Их инструмент используют *сопоставители абстрактных синтаксических деревьев* (Clang AST Matchers, далее – САСД) [11]. Они нужны для описания участков программного кода, которые необходимо трансформировать. Пользователь также должен составить замену для этих участков в терминах узлов *абстрактного синтаксического дерева* (AST, далее – АСД). Авторы предполагают, что пользователи разбираются в синтаксических деревьях вообще и в их построении для кода на C/C++ в частности. Мы же считаем, что это обычно не так, поэтому для рядового пользователя применение `ClangMR` может быть затруднительным.

Вассерман (Wasserman) [13] представил инструмент `Refaster`, предназначенный для рефакторинга кода на языке Java и не требующий никаких DSL. Автор предлагает использовать целевой язык программирования для описания правил трансформации. Это позволяет внедрить эти правила в кодовую базу проекта, что, в свою очередь, упрощает проверку их синтаксиса и семантических ограничений. Каждое правило пишется в форме класса, в котором есть методы с одной из двух аннотаций: `@BeforeTemplate` или `@AfterTemplate`. Такой класс описывает трансформацию и должен содержать как минимум один метод с `@BeforeTemplate` и ровно один метод с `@AfterTemplate`. Инструмент интерпретирует подобное описание следующим образом: он использует код в методах с `@BeforeTemplate` для поиска и сопоставления, после чего заменяет найденный код на то, что написано в

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

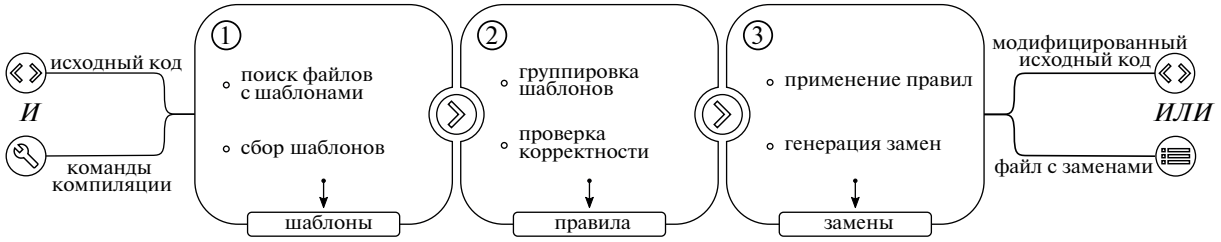

Рис. 1. Схема работы Nobrainer.

методе с `@AfterTemplate`. Поскольку указанный подход кажется нам наиболее понятным и удобным для конечных пользователей, он взят за основу нашего инструмента `Nobrainer`.

Мы также решили, что для поиска и сопоставления кода на C/C++ оптимальным является подход `ClangMR`. Однако использование САСД напрямую может быть затруднительным. По этой причине мы предоставляем более высокоуровневый фреймворк, который, в свою очередь, уже использует САСД.

В задаче трансформации кода стандартным решением является построение АСД, затем его преобразование и генерация кода. Такой подход используется инструментами `Proteus` [4], `Jackpot` [6] и `Eclipse C++ Tooling` [5]. Основной проблемой здесь является сохранение исходного форматирования на этапе генерации. Однако авторам `ClangMR` [14] удается избежать этой проблемы благодаря использованию для трансформации кода инфраструктуры `Clang/LLVM`. Это позволяет разработчикам напрямую править исходный код на уровне лексем. Мы также используем эту инфраструктуру, потому что полагаем, что это лучшее решение для трансформации программного кода на C/C++.

3. АРХИТЕКТУРА

В этой главе мы опишем общую архитектуру и схему работы `Nobrainer`.

Инструмент основан на использовании специальных примеров – фрагментов кода, написанных на языках C/C++. Каждый такой пример мы называем *шаблоном*, так как он может описывать целое семейство случаев. Сначала пользователь приводит пример языковых конструкций, которые он хотел бы заменить. Такой пример называется шаблоном `Before`. Затем в шаблоне `After` нужно написать код, который будет являться заменой для всех найденных конструкций. Описания шаблонов `Before` и `After` могут быть добавлены в любой, в том числе отдельный, файл пользовательского проекта.

Для запуска `Nobrainer` на каком-либо проекте необходимо сделать следующее:

• добавить в этот проект правила трансформации кода;

• указать команды компиляции для проекта (в настоящее время используются команды в формате `JSON`[2] из инфраструктуры `Clang/LLVM`).

На рис. 1 представлена внутренняя структура `Nobrainer`. Каждый пронумерованный блок соответствует одному этапу работы инструмента. Прямоугольные блоки в нижней части рисунка показывают, какие данные получаются на выходе каждого этапа.

Первый этап – поиск шаблонов `Before` и `After` в программном коде проекта.

Второй этап – составление списка правил трансформации путем группировки и проверки корректности шаблонов.

Третий этап – применение правил и генерация замен. Для каждого правила инструмент пытается сопоставить шаблоны `Before` с исходным кодом проекта. Для совпадений (успешных сопоставлений) по шаблону `After` формируются замены.

`Nobrainer` позволяет либо сразу применить к файлам проекта сформированные замены, либо сохранить их в формате `YAML`[3]. Во втором случае замены можно применить, используя специальный инструмент из набора `Clang Extra Tools`[4] – `clang-apply-replacements`.

Подробно все этапы описаны в главе 4.

4. ФОРМАЛЬНОЕ ОПИСАНИЕ И ДЕТАЛИ РЕАЛИЗАЦИИ

`Nobrainer` предоставляет программный интерфейс (API) для написания шаблонов. Он разделен на интерфейс для языка C и интерфейс для языка C++. Оба интерфейса дают возможность писать шаблоны для преобразования отдельных *выражений* (*expressions*) и последовательности

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

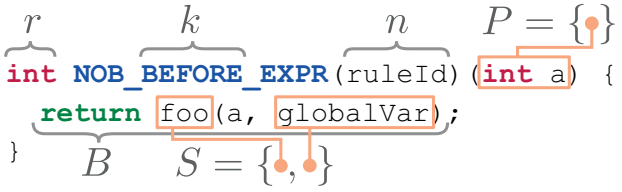

Рис. 2. Разбор шаблона Before.

операторов (*statements*) в исходных файлах на соответствующем языке.

Для более четкого понимания того, что из себя представляет шаблон, здесь и далее рассмотрим конкретный пример преобразования выражения. Предположим, что пользователь хочет найти все вызовы функции `foo` с двумя аргументами. Первый аргумент – это произвольное выражение типа `int`. Второй аргумент – это глобальная переменная `globalVar`. Указанные вызовы необходимо заменить вызовами функции `bar` с теми же аргументами. Листинг 1 содержит пример написания такого правила с использованием интерфейса для языка C.

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

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

Листинг 1: Пример шаблона для преобразования выражения

Выражения для сопоставления и замены описываются внутри оператора `return`. Это сделано специально для того, чтобы делегировать проверку совместимости типов выражений из `Before` и `After` компилятору.

Трансформации инструмента `Nobrainer` основаны на том, что два выражения одинакового типа синтаксически являются взаимозаменяемыми. Это утверждение верно за исключением определенных ситуаций, когда выражения должны быть взяты в скобки. Существует набор специальных правил, которые позволяют избежать эту проблему, но в этой статье подробно они не рассматриваются.

Для того чтобы формально определить термин *шаблон* для данной программы, необходимо ввести следующую нотацию:

- Θ – конечное множество всех типов в данной программе;
- Σ – конечное множество всех определенных символов (функций, переменных, типов) в программе;
- $\mathscr{A}$ – конечное множество всех узлов АСД, из которых состоит данная программа;
- $\mathscr{C}$ – конечное множество символов, разрешенных для идентификаторов в языках C/C++;
- $\mathscr{P}$ – конечное множество всех параметров функции, $p \in \mathscr{P}$ и $p = \langle n_p, t_p \rangle$, где $n_p \in \mathscr{C}^*$ – это имя параметра и $t_p \in \Theta$ – это его тип.

Шаблон для замены выражения (expression template) можно формально описать как шестерку

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

где

- $k \in \{\texttt{Before}, \texttt{After}\}$ – тип шаблона;
- $n \in \mathscr{C}^*$ – идентификатор правила, который используется для связывания шаблонов `Before/After`;
- $r \in \Theta$ – тип выражения;
- $B \subset \mathscr{A}$ – тело шаблона;
- $P \subseteq \mathscr{P}$ – множество параметров;
- $S \subseteq \Sigma$ – множество символов, которые используются в теле B.

Последние два элемента нуждаются в дополнительном пояснении.

Параметры шаблона P используются для выражения специальной семантики. `Nobrainer` рассматривает каждый параметр как произвольное выражение соответствующего типа. Например, параметр `a` в Листинге 1 соответствует **любому** выражению типа `int`.

Множество символов S используется для проверки корректности (см. Главы 4.2.3 и 4.3.2).

В следующих подглавах более подробно рассматриваются все стадии работы инструмента `Nobrainer`.

4.1. Поиск шаблонов

На первом этапе выполняется сбор и проверка корректности всех шаблонов, определенных в данном проекте.

4.1.1. Сбор шаблонов. `Nobrainer` обходит каждый файл и пытается найти функции, которые были объявлены с использованием программного интерфейса инструмента. Этот поиск можно выполнять только в синтаксически разобранных файлах. Если выполнять эту процедуру на всем проекте, это приведет к существенному замедлению работы инструмента. Этого можно избежать

путем обработки только тех файлов, которые содержат директивы `#include` с заголовочными файлами программного интерфейса `Nobrainer`.

В результате получается множество всех шаблонов, определенных пользователем. Обозначим это множество $\mathcal{T}$.

4.1.2. Проверка корректности шаблонов. Когда все шаблоны собраны, `Nobrainer` переходит к проверке корректности каждого отдельного шаблона. Здесь необходимо проверить, что шаблоны из $\mathcal{T}$ имеют правильную структуру. Важно отметить, что синтаксическая корректность шаблонов гарантируется компилятором. Их определения – это часть программного кода проекта, а значит для них проводится синтаксический разбор во время поиска. Это включает в себя проверку доступности всех используемых символов, проверку типов и др.

Каждый шаблон T_{expr} должен определять *ровно одно* выражение. Формально это правило может быть сформулировано следующим образом: тело шаблона B должно состоять из единственного непустого оператора `return`.

На текущий момент инструмент игнорирует шаблоны с использованием функциональных макросов (functional style macros) и лямбда-выражений языка C++. Это временное ограничение, которое мы планируем устранить в будущем.

Результатом работы данного этапа является множество корректных (с точки зрения вышеописанных правил) шаблонов, которое мы обозначим $\mathcal{T}_+$.

4.2. Составление списка правил преобразования

Для произвольного идентификатора $id \in \mathcal{C}^*$ определим две группы шаблонов B_{id} и A_{id} следующим образом:

$$B_{id} = \{T \in \mathcal{T}_+ | n_T = id, k_T = \texttt{Before}\} \quad (2)$$

$$A_{id} = \{T \in \mathcal{T}_+ | n_T = id, k_T = \texttt{After}\} \quad (3)$$

Две указанные группы описывают одну уникальную пользовательскую трансформацию кода, потому что они включают все шаблоны `Before` и `After` с одним и тем же именем. Однако для того чтобы B_{id} и A_{id} задавали правило преобразования, дополнительно должны выполняться следующие условия:

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

где

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

Оператор $\prec$ – это *оператор совместимости*. Он задает отношение между шаблоном x и шаблоном y, которое указывает на то, что фрагмент кода, сопоставленный шаблону y, может быть безопасно заменен кодом шаблона x. Равенство возвращаемых типов r гарантирует тот факт, что заменяемое выражение имеет тот же тип, что и исходное. В то же время, условие $P_x \subseteq P_y$ позволяет быть уверенным в том, что во время замены инструмент будет иметь достаточно выражений для подстановки на место параметров из шаблона x.

Таким образом, можно определить *правило трансформации* как пару $R_{id} = \langle B_{id}, A_{id} \rangle$, где B_{id} и A_{id} удовлетворяют условиям (4). Множество всех правил, определенных в проекте, обозначим $\mathcal{R}$.

4.2.1. Обработка шаблонов `Before`. На этой стадии работы инструмента `Nobrainer` шаблоны Before преобразовываются в САСД. При помощи последних удобно искать поддеревья, удовлетворяющие заданным условиям. Каждый САСД состоит из предикатов для вершины поддерева и предикатов всех его узлов. Такая структура напоминает само синтаксическое дерево, и этот факт используется для генерации САСД в `Nobrainer`. Мы рекурсивно обходим все узлы дерева, построенного для тела шаблона `Before`, и для каждого типа узла генерируем свой САСД. Кроме того, для параметров шаблона создаются специальные САСД, которые будут связывать сопоставляемые поддеревья с именем параметра. В результате конструируется итоговое “дерево” из САСД. Таким образом, нам достаточно правильно реализовать генерацию САСД для каждого типа узла.

Рисунок 3 содержит упрощенный пример такого преобразования. На нем можно увидеть три представления шаблона `Before`: исходный код, синтаксическое дерево и САСД. Сплошные стрелки ведут от родительского к дочернему узлу АСД, а пунктирные показывают соответствие между узлом синтаксического дерева и САСД. САСД необходимо конструировать снизу вверх, поэтому АСД обходится в глубину.

4.2.2. Сопоставление одинаковых поддеревьев. Рассмотрим шаблон `Before` из Листинга 2. Маловероятно, что пользователь хочет найти вызов foo с двумя произвольными выражениями типа int в качестве аргументов. Наиболее интуитивная интерпретация такая: нужно найти вызов foo, в который передали два одинаковых выражения.

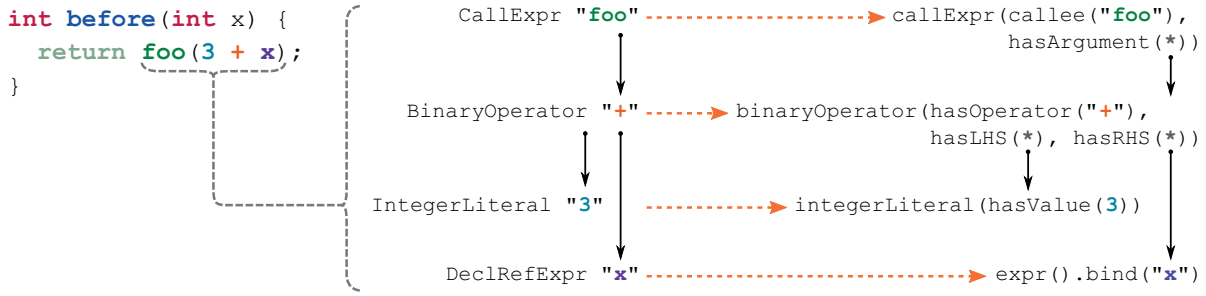

Рис. 3. Рекурсивная генерация САСД.

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

Листинг 2: Пример повторного использования шаблонного параметра

В инфраструктуре Clang/LLVM нет готового решения для сопоставления одинаковых поддеревьев. Используя уже имеющийся в Nobrainer механизм, мы динамически генерируем САСД во время процесса сопоставления. Это делается следующим образом: при сопоставлении первого аргумента вызова foo соответствующее поддерево связывается с параметром x. Затем по этому поддереву генерируется САСД, который и используется для сопоставления со вторым аргументом вызова.

4.2.3. Обработка шаблонов After. При обработке шаблона After наша цель – сконструировать текст, составляющий результат замены. Поэтому мы преобразовываем шаблоны After в форматные строки. Тело шаблона *B* может содержать элементы, у которых нельзя просто взять текстовое представление и использовать его в такой строке. Такие части мы называем *изменяемыми*. Во время обхода тела шаблона After мы извлекаем диапазоны, соответствующие изменяемым частям. Каждый диапазон содержит начальную и конечную позиции определенного узла синтаксического дерева. Есть два типа *изменяемых* частей.

Первый тип – это параметры шаблона в теле After, которые заполняются во время генерации замен (см. Подглаву 4.3.2).

Второй тип – символы (идентификаторы, не являющиеся параметрами шаблона). Вставка символов в произвольные места исходного кода может быть синтаксически некорректной, поскольку в том месте, куда символ вставляется, он может быть не объявлен. Поэтому мы собираем информацию о символах, которая затем используется во время генерации замен (см. Подглаву 4.3.2).

Например, для шаблона After из Листинга 3 сгенерируется следующая форматная строка: "${bar}($[x]) + 42". В этом примере Nobrainer выделяет символ bar и параметр x, помечая их соответствующим образом. Остальные части строки инструмент рассматривает как неизменяемые.

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

Листинг 3: Пример шаблона After

4.3. Применение правил и генерация замен

4.3.1. Применение правил. На следующем шаге нужно определить ситуации, в которых необходимо применять правила $\mathcal{R}$. Чтобы сделать это на всем проекте, Nobrainer проводит синтаксический разбор всех исходных файлов. После этого инструмент применяет САСД, сгенерированные для каждого правила.

Каждый раз, когда САСД находит какой-либо узел синтаксического дерева, удовлетворяющий всем предикатам, Nobrainer получает этот узел и список поддеревьев, связанных с параметрами соответствующего шаблона Before. Используя эту информацию и шаблон After, Nobrainer генерирует преобразование исходного кода, называемое *заменой*.

4.3.2. Генерация замен. Замена включает в себя:

- название файла, к которому применяется замена;
- смещение от начала файла для заменяемого текста;
- длину заменяемого текста;
- текст замены.

Первые три элемента Nobrainer получает из сопоставленного узла. Текст замены конструируется из форматной строки для шаблона After и поддеревьев, связанных с параметрами шаблона. Для каждого такого поддерева инструмент получает его текстовое представление из исходного

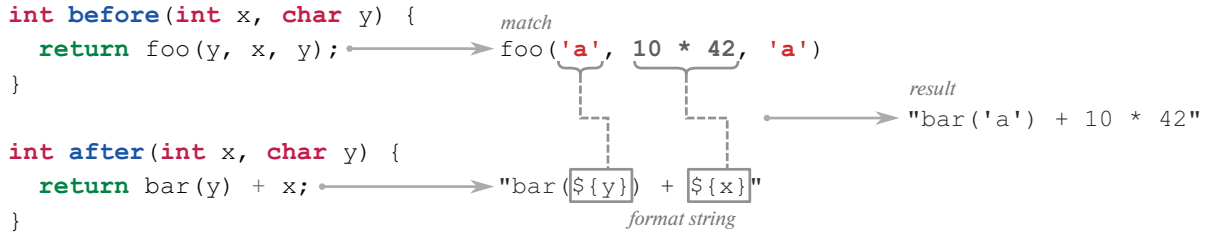

Рис. 4. Генерация замены.

кода, а затем подставляет эту подстроку в форматную строку вместо соответствующего параметра. На рис. 4 показана генерация текста замены для реального фрагмента кода.

Однако подобная замена может вызвать ошибки компиляции из-за недоступности каких-либо символов, появившихся в результате замены. `Nobrainer` должен обеспечить доступность этих символов. Для этого инструмент может добавлять:

• директивы `#include` для заголовочного файла, в котором определен данный символ;

• указание пространства имен или глобальной области видимости.

Таким образом, полученный фрагмент кода является корректным.

4.4. Типовые параметры

Использование произвольных выражений в качестве параметров шаблонов позволяет задавать правила в обобщенном виде. Однако, этого может быть недостаточно. Указание в правиле конкретных типов может сильно ограничить его выразительность и сузить круг применения.

Чтобы устранить этот недостаток, мы вводим в синтаксис шаблона множество типовых параметров $\Phi \subset \mathscr{C}^*$. Таким образом, мы расширяем определение шаблона для замены выражения (1) до следующего:

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

а определение оператора совместимости $\prec$ (5) до:

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

Заметим, что типовые параметры Φ полностью аналогичны параметрам P.

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

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

Листинг 4: Пример правила с типовым параметром

Листинг 4 демонстрирует правило, в котором используются типовые параметры.

4.5. Преобразование последовательности операторов

Инструмент `Nobrainer` может использоваться для трансформации не только отдельных выражений, но и последовательности операторов. Для обеспечения корректности таких преобразований мы расширяем определение шаблона T'_{expr} (6) до T_{stmt} следующим образом:

$$T_{stmt} = \langle k, n, r, P, B, S, \Phi, D \rangle, \tag{8}$$

где D – это конечное множество всех объявлений локальных переменных функции, $d \in D$ и $d = \langle n_d, t_d \rangle$, где $n_d \in \mathscr{C}^*$ – это имя параметра и $t_d \in \Theta$ – это его тип. Также мы усиливаем определение оператора совместимости $\prec$ (7) до:

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

Таким образом, в шаблонах T_{stmt} все локальные переменные рассматриваются как параметры. Дополнительно мы требуем от пользователя не добавлять и не удалять объявления переменных.

Однако в определении (9) не учитываются области видимости локальных переменных, и инструмент не может гарантировать, что после замены код останется компилируемым. Это является временным ограничением. В Листинге 5 приведен пример, который при неаккуратном использовании может породить некомпилируемый код в том случае, когда счетчик используется вне цикла.

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

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

Листинг 5: Пример некорректного шаблона с изменением области видимости.

Также в шаблонах T_{stmt} допускается пустое тело функции After, что позволяет пользователю удалить некоторый код, сопоставленный с шаблонами Before.

4.6. Сопоставление произвольного оператора

С помощью трансформаций последовательности операторов можно изменять структуру кода в целом, поэтому было бы удобно обобщать правила преобразования. Для этого мы добавили специальную функцию `anystmt` в интерфейс инструмента `Nobrainer`. Ее вызов сопоставляется с любым оператором. Продемонстрируем применение `anystmt` на следующем примере (Листинг 6).

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

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

Листинг 6: Пример использования `anystmt`

Здесь мы хотим инвертировать условие в условном операторе с вызовом функции `foo`.

Как можно заметить, в списках аргументов шаблонов появились новые параметры типа `nobrainer::Name`. Это служебный тип, который мы ввели в интерфейс инструмента `Nobrainer`. В шаблоне `Before` параметр этого типа связывается с произвольным оператором, сопоставленным с вызовом `anystmt`. Это позволяет использовать связанный с соответствующим параметром оператор в шаблоне `After`.

Для сопоставления произвольного оператора в теле шаблона `Before` необходимо написать вызов функции anystmt с единственным параметром типа `nobrainer::Name`. Чтобы использовать сопоставленный с anystmt оператор в шаблоне `After`, нужно указать связанное с ним имя параметра (Листинг 6).

Для обеспечения этой функциональности мы ввели дополнительный САСД, который сопоставляет вызов функции `anystmt` с произвольным оператором и связывает с ним имя единственного параметра этого вызова. Стоит отметить, что в текущей реализации подстановка оператора, связанного с параметром типа `nobrainer::Name`, в шаблон `After` порождает предупреждение компилятора “Неиспользуемый результат выражения”. Обе проблемы мы устраним в скором времени.

4.7. Сопоставление произвольной последовательности операторов

Для расширения возможностей трансформации структуры кода мы вводим две дополнительные специальные функции: `block` и `block_greedy`. Их вызовы сопоставляются с произвольной, в том числе и пустой, последовательностью операторов и работают аналогично ленивой (.*?) и жадной (.*) квантификации в регулярных выражениях соответственно. Функции `block` и `block_greedy` принимают в качестве единственного аргумента параметр типа `nobrainer::Name` и связывают его имя с сопоставленной последовательностью операторов. Чтобы использовать эту последовательность в шаблоне `After` достаточно написать это имя аналогично механизму для `anystmt`.

Рассматриваемая функциональность обеспечивается специальным САСД. Внутренняя реализация этого САСД основана на алгоритме из книги Фридла (Friedl) [15], но его описание выходит за рамки данной статьи. На следующем примере мы продемонстрируем разницу в использовании ленивого (`block`) и жадного (`block_greedy`) интерфейсов (Листинги 7 и 8).

Стоит отметить, что на данный момент допустимо только одно использование каждого параметра типа `nobrainer::Name`, используе-

мого с `block` или `block_greedy` в теле шаблона `Before`.

```
void before(Name x) {
  block(x);
  foo();
}

void after(Name x) {
  x;
}

void example() {
            // Первое сопоставление:
            // <= block(x);
  foo();    // <= foo();

            // Второе сопоставление:
  bar();    // <= block(x):
  foo();    // <= foo();

            // Третье сопоставление:
  bar();    // <= block(x);
  foo();    // <= foo();
}
```

Листинг 7: Пример использования `block`

```
void before(Name x) {
  block_greedy(x);
  foo();
}

void after(Name x) {
  x;
}

void example() {
          // Единственное сопоставление:
          //
  foo(); // <=*
  bar(); // *
          // * block(x);
  foo(); // *
  bar(); // <=*
          //
  foo(); // <= foo();
}
```

Листинг 8: Пример использования `block_greedy`

В случае, когда `nobrainer::Name` связывается с пустой последовательностью операторов, во время создания замены в шаблон `After` вместо соответствующего параметра будет подставлена пустая строка.

4.8. Небезопасные преобразования

До сих пор мы старались обеспечить полную корректность трансформаций, что иногда сильно ограничивает возможности использования инструмента `Nobrainer`. В связи с этим мы вводим еще один вид шаблонов – T_{unsafe}. Его определение не отличается от шаблона T_{stmt} (8), но он имеет ослабленный оператор соответствия $\prec$:

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

Мы больше не требуем совпадения множеств D_x и D_y. Это, в свою очередь, позволяет добавлять и удалять локальные переменные, а также менять их тип в шаблоне `After`. С ввдениеам небезопасных шаблонов стала возможна реализация, например, следующей замены (Листинг 9):

```
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);
  }
}
```

Листинг 9: Пример использования небезопасного шаблона

Однако ответственность за ошибки компиляции, связанные с изменением локальных объявлений в результате замены, ложится на пользователя.

5. РЕЗУЛЬТАТЫ

В этой главе описывается подход к тестированию инструмента `Nobrainer`, рассматриваются примеры правил преобразования и анализируется производительность.

5.1. Тестирование

Тесты для инструмента можно разделить на две группы. Первая группа состоит из модульных и интеграционных тестов для каждого этапа из Главы 3. В основном они используются для проверки корректности автоматически сгенерированных САСД для шаблонов `Before` (Подглава 4.2.1) и форматных строк для шаблонов `After` (Подглава 4.2.3).

Вторая группа – это регрессионные тесты, которые включают в себя несколько проектов с открытым исходным кодом. Для каждого проекта вручную были созданы файлы с заранее заданными правилами преобразования. Система регрессионного тестирования запускает `Nobrainer`, измеряет время выполнения и проверяет, что все заданные преобразования корректно применились и что проект остался компилируемым.

5.2. Примеры правил

В этой подглаве представлены некоторые примеры правил преобразования, которые поддерживаются инструментом `Nobrainer`.

Первый пример (Листинг 10) описывает правило, с помощью которого изменяется порядок аргументов во всех вызовах метода `compose`. `Nobrainer` заменит каждый вызов метода `a.compose(x, y)` у произвольного объекта `a` класса `Agent` на `a.compose(y, x)`.

Этот пример демонстрирует, как автоматически можно поменять местами аргументы вызова.

```
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);
}
```

Листинг 10. Пример правила для изменения порядка аргументов

Второй пример (Листинг 11) показывает, как можно использовать `Nobrainer` для упрощения исходного кода.

```
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();
  }
};
```

Листинг 11: Пример правила для проверки строки на пустоту

Напомним, что каждое правило может содержать несколько шаблонов `Before`, но только один шаблон `After`. Использование нескольких шаблонов `Before` в данном случае позволяет сгруппировать логически связанные трансформации.

Третий пример использует два вида параметров: обычные и типовые. Листинг 12 содержит исходный код правила преобразования.

```
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);
  }
};
```

Листинг 12: Пример правила для приведения к неконстантному типу

Указанное правило находит все операции приведения к неконстантному типу в стиле языка C и заменяет их эквивалентными конструкциями `const_cast`. В данном случае параметр `x` – это произвольное выражение, которое имеет тип “указатель на любой константный тип”. Указанное выражение должно замениться на выражение точно такого же типа, но без квалификатора `const`. `Nobrainer` учитывает все эти требования и корректно выполняет замены.

Четвертый пример описывает упрощение условия оператора `if` с помощью `anystmt` (Листинг 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;
  }
};
```

Листинг 13: Пример правила с использованием `anystmt`

Данное правило удаляет отрицание в условии и меняет местами ветви оператора `if`.

В **пятом примере** реализуется небезопасная трансформация (Листинг 2). Здесь мы меняем тип переменной цикла.

```
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;
    }
  }
};
```

Листинг 14: Пример правила с использованием небезопасного шаблона

5.3. Производительность

Измерения производительности проводились путем запуска системы регрессионного тестирования пять раз. Для этого использовалась рабочая станция на базе Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz с 64 Гб оперативной памяти и ОС Ubuntu 16.04 LTS.

Таблица 1 содержит результаты измерений. Для каждого проекта отдельно представлены его размер (тыс. строк кода), количество замен, которые применяет `Nobrainer`, и непосредственно время работы инструмента на данном проекте. Последнее состоит из времени работы двух фаз. Время каждой фазы измерялось отдельно. Первая фаза – это синтаксический разбор исходного кода проекта. Вторая фаза включает в себя всю основную логику работы инструмента `Nobrainer` (см. Главу 3).

Из табл. 1 видно, что затраченное время сильно варьируется в зависимости от проекта. В частности, это верно для времени работы как первой фазы, так и второй фазы. На рис. 5 дополнительно приведены процентные соотношения для времени синтаксического разбора на каждом проекте из системы регрессионного тестирования. Результаты показывают, что разбор в среднем занимает более 81% всего времени работы. Тот факт, что время выполнения оставшихся операций в среднем занимает менее 20%, гово-

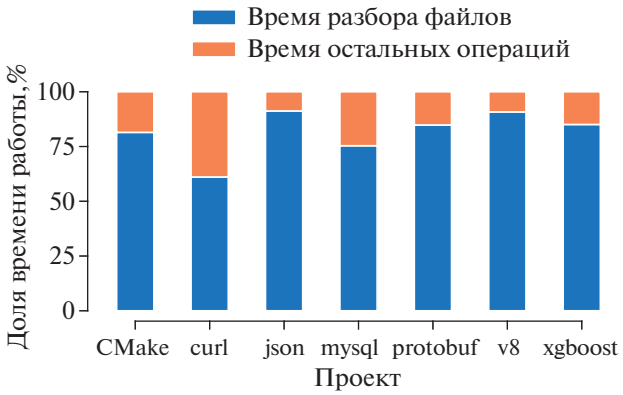

Рис. 5. Процент времени, затрачиваемого на синтаксический разбор.

Таблица 1. Производительность

Проект	Размер (тыс. строк)	Число замен	I фаза (сек)	II фаза (сек)
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

рит о том, что производительность инструмента Nobrainer близка к оптимальной.

6. ДАЛЬНЕЙШИЕ ИССЛЕДОВАНИЯ

На текущий момент Nobrainer поддерживает правила замены выражений и операторов. Кроме того, есть поддержка правил с типовыми параметрами. Инструмент уже можно использовать в системах непрерывной интеграции (CI), однако время работы все еще не позволяет применять его интерактивно на больших проектах. В целом, можно выделить три основных направления исследований для дальнейшей работы:

1. поддержку оставшихся узлов АСД, соответствующих выражениям и операторам C/C++;

2. улучшение производительности;

3. повышение удобства использования.

Для улучшения производительности предполагается исследовать возможные подходы к уменьшению времени, затрачиваемого на синтаксический разбор. Есть два возможных пути. Во-первых, это оптимизация фазы поиска совпадений за счет игнорирования файлов, не содержащих символов из шаблонов Before. Во-вторых, это исследование возможности автоматической генерации предкомпилированных заголовочных файлов (PCH). Эти файлы должны уменьшить время синтаксического разбора заголовочных файлов анализируемого проекта.

Для повышения удобства существует несколько направлений. Сейчас Nobrainer выполняет трансформации сразу на всех файлах проекта. Необходимо предоставить пользователю возможность указывать части проекта, для которых следует выполнять трансформации. Также важно рассмотреть варианты интеграции с другими инструментами для разработчиков. Например, упростить взаимодействие с пользователем путем создания расширений для интерактивных сред разработки (IDE). Другой возможный вариант – это использование инструмента Nobrainer в качестве вспомогательного программного средства. Например, с его помощью можно пытаться автоматически исправлять ошибки и дефекты, найденные статическим анализатором.

7. ЗАКЛЮЧЕНИЕ

В данной статье мы представили Nobrainer – инструмент для выполнения автоматических трансформаций программ на языках C/C++. Он основан на двух принципах: простоте использования и корректности правил преобразования. В работе мы уделили большое внимание описанию модели, архитектуры и деталей реализации с обоснованием принятых решений, результатами и примерами.

Наши результаты показывают, что Nobrainer уже может использоваться в системах непрерывной интеграции для проведения трансформаций на промышленных проектах. Мы описали текущие недостатки и обозначили возможные направления для улучшений. В будущем мы планируем повысить удобство использования нашего инструмента, а также интегрировать его с другими программными средствами для разработчиков.

СПИСОК ЛИТЕРАТУРЫ

1. *Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka*. Managing Technical Debt in Software-reliant Systems. In Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

2. *Ward Cunningham*. The WyCash Portfolio Management System. SIGPLAN OOPS Mess. December 1992. V. 4 (2). P. 29–30.

3. *Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts*. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, June 1999.

4. *Daniel G. Waddington and Bin Yao*. High-Fidelity C/C++ Code Transformation. Electronic Notes in Theoretical Computer Science. 01 2007. V. 141. P. 35–56.

5. *Emanuel Graf, Guido Zgraggen, and Peter Sommerlad*. Refactoring support for the C++ development tooling. In OOPSLA Companion, 2007.

6. *Jan Lahoda, Jan Bečička, and Ralph Benjamin Ruijs*. Custom Declarative Refactoring in NetBeans: Tool Demonstration. In Proceedings of the Fifth Workshop on Refactoring Tools, WRT'12, pages 63–64, New York, NY, USA, 2012. ACM.

7. *Leo A. Meyerovich and Ariel S. Rabkin*. Empirical Analysis of Programming Language Adoption. SIGPLAN Not., October 2013. V. 48 (10). P. 1–18.

8. *Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black*. How we refactor, and how we know it. In ICSE, pages 287–297. IEEE, 2009.

9. *Gustavo H. Pinto and Fernando Kamei*. 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 '13, pages 33–36, New York, NY, USA, 2013. ACM.

10. *Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov*. A Large-scale Study of Programming Languages and Code Quality in GitHub. Commun. ACM, September 2017. V. 60 (10). P. 91–100.

11. The Clang Team. Clang documentation: Matching the Clang AST. https://clang.llvm.org/docs/LibASTMatchers.html.

12. *Will Tracz*. Refactoring for Software Design Smells: Managing Technical Debt by Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. ACM SIGSOFT Software Engineering Notes. 2015. V. 40 (6). P. 36.

13. *Louis Wasserman*. Scalable, example-based refactorings with Refaster. In Proceedings of the 2013 ACM workshop on Workshop on refactoring tools, pages 25–28. ACM, 2013.

14. *Hyrum Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan*. Large-Scale Automated Refactoring Using ClangMR. In Proceedings of the 29th International Conference on Software Maintenance, 2013.

15. *Джеффри Фридл*. Регулярные выражения. Символ-Плюс. СПб., 2008.