北京大学

博士研究生学位论文

题目：测试文档

姓　　名：某某

学　　号：0123456789

院　　系：某某学院

专　　业：某某专业

研究方向：某某方向

导　　师：某某教授

某年某月

版权声明

摘要

pkuthss 文档模版最常见问题:

在最终打印和提交论文之前，请将 pkuthss 文档类选项中的 `colorlinks` 改为 `nocolorlinks`，因为图书馆要求电子版论文的目录必须为黑色，且某些教务要求打印版论文的文字部分为纯黑色而非灰度打印。

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，[**test-zh**]、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 `openany` 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

关键词：其一，其二

Test Document

Test (Some Major)

Directed by Prof. Somebody

Abstract

Test of the English abstract.

Keywords: First, Second

目录

序言

0.1 Introduction

现在，许多程序分析工具都涉及代码修改功能。在这些工具中，有许多都是代码修复工具 [1–4]。通常来说，修复工具的输入是一段代码和一组测试，并不断修改代码直至代码能通过测试。另一些程序分析工具是API升级工具 [5–7]。当API升级时出现了不兼容情况时，这些工具可以自动更新相应的API调用让程序与API契合。我们把这类直接修改代码的工具称作*程序编辑工具*。

另一方面，许多程序语言的实现都带有预处理器 [8–10]。最常见的预处理器是C预处理器（C++）。许多程序语言也接受C++，包括C，C++和Objective-C。同时，程序员也时常使用C++来写一些零散的小工具。这时就会使用到预处理器。例如Korpela [11]曾在文章中描述过用C++写一个HTML编辑工具：这个工具会把页面间相同的HTML代码转换成C的宏，而不是直接生成HTML页面。然后页面再利用这些宏最终生成HTML文件。

程序编辑工具通常不会去修改程序的预处理指令。但是，只有能够把修改反映射到预处理之前的代码的工具才算有用。只在预处理后的代码中修复错误会导致原有程序再次编译的时候错误依然存在，这样毫无意义。这个问题具有挑战性，因为工具必须同时能理解预处理命令和目标程序语言，同时保证修改在两边能保持一致。事实上，现有的程序编辑工具往往无法正确处理预处理指令、或是直接不处理预处理指令，例如现有的C语言工具：GenProg [1, 2]，RSRepair [3]，和SemFix [4]。这三个工具都只在预处理后的代码上工作。用户需要手动检查预处理后的代码变化，并自行修改源代码——而这又增加了新bug的可能。

代码重构是一个密切相关的领域 [12, 13]。在代码重构中，程序编辑工具有时需要直接修改预处理指令。比如：用户有时想重命名一个宏，或者需要提取一

个宏作为重构的一部分。在这种情况下，工具开发者别无选择，只好修改预处理指令。典型情况中，工具开发者会定义一种新的C语法使得原有的C语法和预处理指令能兼容。但是，如果我们考虑更一般的程序编辑工具，这种方法就捉襟见肘了。首先，工具开发者需要在真正设计工具之前把精力花费在学习语言的细节上。其次，学习新语言的努力并不能在其他语言中复用。

本文中我们提出了一个轻量级的支持C++的程序编辑工具实现方法。该系统时一个双向的C++预处理器：原有的预处理过程可以背看作一个正向变换，在此基础上我们添加一个能够把预处理后代码上的修改反响映射回去的反向变换。于是，程序编辑工具现在可以关注于预处理后的代码，并把映射修改的交给我们的自动工具[1]。

在这里我们列举一些例子：（1）上文中所提到的三个学术届认可的错误修复系统现在可以处理预处理前的代码；（2）API升级软件现在可以在有预处理代码的情况下更好地实现；（3）所有并不需要关心预处理过程的程序编辑工具都能够被改善。

虽然现在存在着几种双向变换的技术 [15–17]，但是它们都是为数据的转换设计的。给定一个源数据集`s`，一个变换程序`p`，和一个目标数据集`t = p(s)`，这些方法试图将`t`上的变化描述成`s`的变化。然而，C的预处理器与数据的转换不同，因为C的源代码不仅包含了作为数据的代码，还包含了座位变换程序的预处理指令。这就要求反向变换的机制能处理更复杂的情况：当目标数据发生变化时，我们可能要变化源数据、转换程序、或者是二者都变化。

本系统的一个创新处是在能够处理上文提到的复杂双向变换情况的同时，尽量最小化对变换程序`p`的修改。首先，该设计中从不引入新的宏定义或修改现有的宏定义，有效地限制反向变化的影响。其次，该设计只会移除/修改宏调用而并不会创造新的宏调用。再有，该设计只会在必要的时候移除宏调用。这样，我们就能尽可能保证源代码和修改之前的相似度。

实现这样的算法设计也是十分具有挑战的。典型的这类双向变换的方法 [15, 16, 18]是顺着程序的抽象语法树AST拆分双向变换。每一个子树对应着一个小的双向变换，组合起来就成为整个程序的双向变换。但是，一个未预处理的C++的程序并不能简单地解析成树状结构。比方说，在下列代码中，

[1] 这个过程并不是全自动的。因为我们的工具现在支持程序编辑的基本操作。尽管这些步骤可以用通用代码差分方法实现 [14]，但是如果工具能直接提供基本编辑操作可以有最好的效果

```
#define inc(x) 1+x
#define double(x) 2*x
inc(double) 2
inc(double) (x)
```

第一个宏调用`inc(double)`会自动展开成为一个单独的`statement`。但是第二个宏调用不能单独展开成一个`statement`。它需要连上之后的`(x)`，循环展开后才算完整的句子。因此，我们不能把第二个宏调用`inc(double)`当作独立的一段并直接对他做双向变换的分析。为了克服这个困难，我们为描述类似C++程序这样的情况设计了新的模型。我们把预处理看作是对代码数据的重写规则（*rewriting rules*）集，而不是直接把代码解析成抽象语法树。这样的模型把程序的双向变换看作是重写规则的双向变换。

另外，该设计也可以被证明满足双向变换的正确性：（1）如果预处理后的程序没有变化，那么源代码也不会变化；（2）源代码在接受了反向映射的变化后，预处理后会得到相同的变化后的预处理后的代码。这两个性质在双向变化领域被称作GETPUT和PUTGET [19]。

总之，该项目的贡献如下：

- 我们提出了一个轻量级的在程序编辑工具方面的双向C预处理器。我们分析了不同可能的设计并提出了五个反向变换应有的性质，包括GETPUT和PUTGET (Section 1.1)。
- 我们提出了一个能够符合五个性质的算法。该算法把C的预处理看作是重写规则的集合，并结构性地把预处理的双向变换转换到重写规则的双向变换上 (Section 1.2)。
- 我们在Linux内核上验证了我们的方法，并和另外两种基本的做法进行比较：一个是直接把整个修改的文件映射回去；另一个是只把修改了的代码行映射回源代码。实验的结果显示相较于其他方法，我们的方法破坏了相当少的宏调用，并且总是可以给出正确的修改，而其他方法有时不行(Section 1.3)。

Finally, we discuss related work in the paper in

第一章　背景介绍

1.1　Problem

1.1.1　The C Preprocessor

表 *1.1:* 主流预处理器指令与操作

Directives	Functionality	Example	Result
`#progma`	Compiler options	`#progma once`	removed from the preprocessed file
`#include`	File Inclusion	`#include <stdio.h>`	the content of "stdio.h"

`#if, #ifdef, ...`	Conditional compilation	`#ifdef FEATURE1`
		`  x = x + 1;`
		`#endif`
			`x = x + 1;`

`#define X`	Object-like macro definition	`#define X 100` `a = X;`	`a = 100;`
`#define X(a, b)`	Function-like macro definition	`#define F(x) x * 100` `F(10);`	`10 * 100;`
`a ## b`	Concatenation	`#define X a_##100` `X`	`a_100`
`#b`	Stringification	`#define F(x) #x;` `F(hello);`	`"hello";`
`__FILE__, __DATE__, ...`	Predefined macros	`__FILE__`	main.c

表 1.1显示了主流预处理器的指令与操作。一条预处理指令在行首以#开头，在行末结束。宏可以被预处理指令定义，但是它们自身并不是预处理指令。本质上，我们认为现在有四种主要的预处理指令： `#progma` 提供了编译选项，`#include` 描述了包含的头文件， `#if` 提供了条件编译选项，`#define` 是宏定义指令。另外，在一个宏定义重，我们可以使用类似`##` 和`#` 的指令来连接两个变量活着字符化一

个变量。最后，还存在一些预定义的宏，如 __FILE__ ，会随着上下文的不同而被替换。

当C预处理器处理一个源文件的时候，它会依据以下的方法来转换源程序文件：

- 首先展开 #include 和 #if 指令，然后再重复扫描展开后的代码词序列
- 对于每个宏调用，预处理器先处理参数的展开，然后再展开宏调用。
- 对于含有 # 和 ##的参数，预处理器并不会处理这类参数。相反，预处理器会把参数直接文本拷贝到展开项中。
- 当一个宏调用被展开之后，这个被展开的程序词序列会被再次扫描。如果这时还有宏没有展开，预处理器会把宏调用展开。
- 为了避免循环转开，如果一个宏定义已经在展开过程中被展开，那么它就不会被再次展开。
- 如果在展开宏的过程中生成了新的预处理指令，该指令并不会被预处理器执行。

```
#define SAFE_FREE(x) if (x) vfree(x);
#define FREE(x) vfree(x);
#define RESIZE(array, new_size, postprocess) \
  g_resize_times++; \
  postprocess(array); \
  array = vmalloc(sizeof(int)*(new_size));
#define GARRAY(x) g_array##x;

RESIZE(GARRAY(2), 100, FREE);
```

图 *1.1: An example of code preprocessing*

这里举一个实例，让我们来考虑在Figure 1.1里的代码片段。这个例子向我们展示了许多实际项目中的宏定义与宏调用。这段代码中含有四段宏定义和一个宏调用。前两个宏定义被包含在一个用户自定义的空间释放函数里。第三个宏定义是为了用户自定义的内存空间管理和日志记录而重新调整数组的大小。最后一个宏是为了定义一组特殊的全局变量。当预处理器扫描这段代码时，第一个参数 RESIZE 将会被处理。此时， GARRAY(2) 会被展开成 g_array2 。尽管第三个参数 FREE 已经被定义成一个函数状的宏（*function-like macro*），但是并没有能够提供给 FREE 的参数，因此此时预处理器并不会把它当作一个宏调用来处理。然后 RESIZE 的宏调用被展开，于是我们得到了以下的代码：

```
g_resize_times++;
FREE(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

现在我们能看到在展开的宏中，我们已经给 FREE 提供了一个参数列表。因此接着系统会展开 FREE(g_array2)，我们得到以下代码：

```
g_resize_times++;
vfree(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

换句话说，RESIZE 实际上是一个高阶宏（*high-order macro*），因为他的第三个参数也是宏。

1.1.2 反向变换的设计

现在让我们来考虑一下输入为预处理后代码的程序编辑工具们。比方在下面的例子中，程序编辑工具会发现 vfree 可能存在内存泄漏的可能，所以工具会在这句代码前加入一个保护语句，如下：

```
g_resize_times++;
if (g_array2) vfree(g_array2);
g_array2 = malloc(sizeof(int)*(100));
```

现在让我们来考虑一下反向变换。反向变换一般来说输入是预处理后代码上的修改，然后产生预处理前代码上的修改。当生成的修改作用在预处理前的代码上后，新的代码在预处理后会得到与作用输入修改的预处理后代码相同的结构。对于之前例子里面程序编辑工具做出的修改，我们有两种处理拌饭：（1）我们可以修改 RESIZE 的宏定义，把加入的 IF 语句加入到宏定义中；（2）我们可以展开 RESIZE 的宏调用，并按照预处理后的修改吧保护语句添加到原程序中。第二个选项只影响到了这一小段局部的代码，而第一个选项有会影响到全局的其他宏调用。但是，因为反向变换并不知道应该把修改作用到局部还是全局，一个更安全的做法是选择只影响局部的代码。在本例中，有很大可能我们并不需要对每一个vfree的调用都加上保护语句，因为这样会带来大量不必要运行时间。我们有理由相信程序编辑程序会根据需要选择是否在vfree前加上保护语句。进一步讲，局部的选项会尽可能小地修改原有代码，因为全局会影响程序的许多部分。这些可能性让我们想到一个双向预处理器的第一个性质。

Requirement 1 *反向变换不应该改变任何宏定义。*

根据我们定义的第一个性质，我们应该展开宏调用后再在源代码中加入保护语句。然而现在在映射修改时我们又面临着以下两个展开的选择。我们可以把所有的宏都依次展开：

```
g_resize_times++;
if (g_array2) vfree(g_array2);                    (1.1)
g_array2 = malloc(sizeof(int)*(100));
```

活着我们也可以只把宏展开一层：

```
g_resize_times++;
if (GARRAY(2)) FREE(GARRAY(2));                   (1.2)
GARRAY(2) = malloc(sizeof(int)*(100));
```

我们认为code piece (1.2)比code piece (1.1)更好，因为它保存了更多原有的结构，这使得代码更加易懂，重用或是维护。这就引入了我们的第二条性质。

Requirement 2 *反向变换应该尽可能保存现有的宏调用。*

也许有人会提议我们进一步地把这个保护语句缩减成一个新的宏，或者复用现有的某一个宏来实现保护语句的功能。在本例子中，我们的保护语句被 `SAFE_MACRO` 这个新宏定义包含。代码可能如下：

```
g_resize_times++;
SAFE_FREE(GARRAY(2));
GARRAY(2) = malloc(sizeof(int)*(100));
```

但是，这样做是十分危险的。因为宏定义并不是用来替换所有语义相同的代码片段的。比方说，Ernst等人 [8]就在文章中描述过某一个宏定义， `#define ISFUNC 0`，定义了一个在系统调用中时常用到的常数。很明显，我们并不能把整个系统里的 `0`都替换成 `ISFUNC`。这就引入了我们的第三个性质。

Requirement 3 *反向变换不应该引入新的宏调用。*

除了已经提到的三个性质之外，我们还有两条从双向变换领域借鉴过来的性质，叫做 GETPUT 和 PUTGET [19]。令s是预处理前的源程序，t是预处理后代码，c_t是作用在t上的变化，c_s是完全的反向变换所提供的作用在s上的变化。

Requirement 4 (GETPUT) 如果c_t是空的，那么c_s也是空的。

Requirement 5 (PUTGET) 令$s' = c_s(s)$为新的预处理前的源代码。意为把生成的c_s作用在s上。令$t' = c_t(t)$为作用了修改操作的预处理后代码。对s'做预处理会得到t'。

以上五条性质一起定义了反向变换的行为：它应该通过修改宏调用参数、修改普通代码、展开宏调用并且展开尽可能少的宏调用的方法来把预处理后的修改映射到源代码上。

1.1.3 简单的方法

我们将在本节中讨论为什么简单的方法并不能满足我们提出的五条性质。

简单的方法 I (per-file). 第一种简单的方法是直接把修改过的文件不加处理地拷贝覆盖源代码。这种方法十分容易实现，我们称之为*per-file*，但是他有两大缺点。首先，原有的未预处理源代码可能含有宏定义，然而现在的做法会让这些宏定义都丢失掉，那么在其他地方，比如包含了该文件并调用了相应宏的其他代码，可能就会出现错误。其次，这会导致把文件中所有的宏调用都展开，甚至包括所有的 `#include` 指令。整个代码面目全非，破坏了完整性。

简单的办法 II (per-line). 第二个简单的办法是利用代码的性质，只把预处理后代码中被修改的行反向映射回源代码。我们把这个方法称作*per-line*。这个做法看起来是可行的，因为现代的预处理器会记录下预处理前后行间的对应关系。比方说，当GCC预处理我们之前的例子时，它会把1-7行替换成空行。同时它会把从第9行开始的几行展开压缩成一行。可以看到，现代预处理器都记录下了源代码与预处理代码行间的一一映射。

虽然第二种方法相较于第一种有了不删除宏定义的好处，它依然存在隐患。首先，依然有不少宏被不必要地展开了。在我们的例子中，如果我们把修改的那

一行复制回去，那么code piece (1.1)就会时结果，但这与我们定义的第二条性质不符。甚至，如果我们考虑有的工具会在代码中挑选代码复制插入来修改，比如GenProg 1 就会在代码中拷贝不同地方的代码来修改程序的错误。这样一来拷贝的行中所有的宏调用都会被展开，代码的完整性还是被破坏。另外，这还并不是最严重的。如果源文件中的宏调用是一个多行宏（在我们的调研与实验中也确实发现了这样的情况Section 1.3），那么只替换宏调用的第一行会直接带来错误的结果，反而引入了新的bug。比方说，如果在源代码的宏调用中插入了一个断行，如下：

```
RESIZE(GARRAY(2),
  100, FREE);
```

在GCC中，这个宏调用会被展开成两行。其中第一行时全部的宏展开语句，而第二行是空行。因此，修改操作只会作用于第一行。如果反向变换仅仅把第一行替换成新的代码，那么就会留下不正确的程序。

1.2 Approach

As mentioned before, the basic idea of our approach is to interpret CPP as a set of rewriting rules, where the backward transformation reverses each application of the rules. In this section, we first present this interpretation of CPP (Section 1.2.1). Then we describe the change operations we support and first focus on one type of change: replacements. (Section 1.2.2). Since we need to reverse each application of the rules, we first record the applications as rewriting steps (Section 1.2.2), and then derive a backward transformation for each rule, respectively (Section 1.2.3). We also discuss how this process satisfies the requirements (Section 1.2.4) and an optimization of the algorithm (Section 1.2.4). Finally, we discuss how to convert other types of changes into replacement (Section 1.2.6).

For the ease of presentation, we shall consider a subset of C preprocessor first: there is no # operator nor ## operator, no #include directive, nor does any macro expansion contain any self invocation. We will later discuss how to extend this simplified model to a full model (Section 1.2.5).

1.2.1 Modeling forward preprocessing

We view a CPP program as a sequence of tokens. To recognize preprocessor directives from the sequence of tokens, we rely on two special tokens: # at the beginning of a line, and the end of line following from a # at the beginning of the line. We also assume all macro definitions in scope are stored in a *context*.

We deem the semantics of CPP as a set of rewriting rules. Each rewriting rule is taken the form of `guard` $\hookrightarrow$ `action`. When `guard` is true, `action` is performed to replace some tokens at the beginning of the current remaining token sequence, and set the position for the next rule application.

Both `guard` and `action` are modelled as functions. Function `guard` takes the currently remaining token sequence and the current context, and returns a Boolean value to indicate whether the rule can be applied to the current token sequence or not. Function `action` takes the currently remaining sequence and the current context, and returns four components `(finalized, changed, restIndex, newContext)`. This result indi-

cates that the tokens before `restIndex` are replaced by two consecutive subsequences, `finalized` and `changed`, where `finalized` is a sequence that does not need to be further scanned, and `changed` is a sequence that needs to be scanned. The last component, `newContext`, is the updated context. The algorithm for forward preprocessing is given in Algorithm 1. It repeatedly applies the rules in `R` until the whole token sequence has been processed.

```
Input: token sequence src, rule list R
Output: new token sequence res
ctx ← {};
while src.length > 0 do
    for r ∈ R do
        if r.guard(src, ctx) then
            (finalized, hanged, rest, ctx′) ← r.action(src, env);
            break;
        end
    end
    res ← res + finalized;
    src ← changed + src.sub(rest);
    // sub(l) returns a subsequence starting from l
    ctx ← ctx′;
end
```

Algorithm 1: Algorithm for forward preprocessing

There are totally four rules in the rule list `R`. One processes conditionals such as `#if`, `#ifdef`, one processes all other preprocessor directives as we basically need to remove them and updates the context, one processes macro invocations, and the last one processes normal text. These rules are described below.

- R1: This rule processes conditional compilations. Function `guard` determines whether the token sequence starts with a respective directive such as `#if`, `#ifdef`. Function `action` first evaluates the condition based on the current context, and then replaces the whole conditional with either the true branch or the false branch. The whole replacing sequence is returned as `changed` and `finalized` is empty.
- R2: This rule processes other preprocessor directives. Function `guard` determines whether the token sequence starts with a preprocess directive. Function `action` parses the directives, makes necessary changes to the context, and returns the index

of the first token after the directive as `restIndex`, with `finalized` and `changed` empty.

- R3: This rule expands a macro invocation. Function `guard` determines whether the first token in the sequence is the name of an object-like macro or the first two tokens are the name of a function-like macro and an open parenthesis. Function `action` consists of two steps.
  - First, the arguments are preprocessed by recursively applying R3 and R4[1]. When expanding an argument, the current context is used, and the token sequence includes only the argument.
  - Second, the occurrences of the parameters in the macro body are substituted by the preprocessed arguments. The new body are returned as `changed`, with `finalized` empty, and `restIndex` is the index of the next token after the macro invocation.
- R4: This rule processes normal tokens not captured by the other rules. Function `guard` always returns true. Function `action` returns the first token as `finalized`, an empty `changed`, and the index of the next tokens as `restIndex`.

As an example, let us consider the following program.

```
#define x 100
hello x
```
(1.3)

The first applicable rule is R2, which parse the macro definition, store it in the context and moves to the next line. The remaining sequence is now `hello x`. Then R4 is applied to move `hello` into the finalized sequence. Next, R3 is applied to expand `x` into `100`. Finally, `100` is scanned again and R4 moves it into the finalized sequence. Now the remaining sequence is empty and the preprocess stops.

[1] Including preprocessor directives in the argument list is an undefined behavior in CPP 20. Here we ignore directives in arguments

1.2.2 Modeling the changes

Program-editing tools can change the program with a variety of operations. In this paper we consider three basic kinds: replacement, insertion, and copy. These operations are summarized from popular bug-fixing tools such as GenProg 1 and SemFix 4, which typically fix a bug by copying or creating an expression to replace another one or inserting at some location.

All the three operations directly manipulate the token sequences. A replacement is a pair `(l, s)`, where `l` is the index of the token to be replaced, and `s` is a token sequence that will replace the token at `l`. An insertion is a pair `(l, s)`, where token sequence `s` will be inserted after the token at index `l`. A copy is a triple (l, l_b, l_e), showing the token sequence between index l_b (inclusive) and index l_e (exclusive) is copied after the token at `l`.

Note that the replacement naturally subsumes deletion. A change `(l, [])` deletes a token, where `[]` denotes an empty sequence.

In the rest of the section we shall first show how to perform a backward transformation with replacements and then map the other two types of operations to replacements.

1.2.3 Rewriting Steps

As mentioned in the introduction, our backward transformation reverses each application of rewriting rules. To perform a backward transformation, we use a data structure to record what rules have been applied to what parts of the source. We call such a record *rewriting steps*. Each rewriting step is a quadruple, `(src, i, ctx, r)`, specifying that rule `r` has been applied to a sub sequence of `src` starting at `i` with context `ctx`. Token sequence `src` is always the current token sequence, containing all modifications made from previous rule applications. Accordingly, a forward transformation is recorded as a sequence of rewriting steps. For example, preprocessing code piece 1.3 produces the following sequence of rewriting steps: `(P, 0, {}, R2)`, `(hello x, 0, {x}, R4)`, `(hello x, 1, {x}, R3)`, and `(hello 100, 1, {x}, R4)`, where `P` is the original program.

1.2.4 Backward Transformation with Replacement

The basic idea of our backward transformation is to propagate the changes backward along the sequence of rewriting steps, such that if we perform a forward preprocessing again, either exactly the same rewriting step is performed, or this rewriting step is replaced by a new sequence of equivalent rewriting steps (whose length can be zero). The algorithm for backward transformation can be found in Algorithm 2. The function `backward` propagates changes along one rewriting step, or reports a failure if no proper propagation can be found.

Input: a sequence of rewriting step `rs`
Input: a set of replacements `c`
Reverse sequence `r`;
for `r` ∈ `rs` **do**
    `c` ← `backward(rs, c)`;
    if `backward` *failed* **then**
        print "Changes cannot be applied.";
        return;
    end
end
return `c`;

Algorithm 2: Algorithm for backward transformation

The key for implementing the backward transformation is to implement the function `backward`. The behavior of `backward` differs for each rewriting rule, and we discuss it from the simplest to the most complex rule.

First, if the rewriting step is performed by R2, the forward preprocessing consumes the preprocessor directive. The backward transformation shifts the replacements according to the length of the consumed macro. For example, consider the following CPP program.

```
#undef hello
hello
```

Now if we have a replacement `(0, x)`, i.e, `hello` is changed into `x`, the backward transformation shifts the replacement into `(4, x)` to ensure that it still changes the macro invocation `hello`.

Second, if the rewriting step is performed by R1, the forward transformation replaces

the whole conditional directive with one of its branches. In the backward transformation, we map the changes on the replaced branch back to its original location, and shift other changes if necessary. Since we never change macro definitions, the conditional is guaranteed to evaluate to the same branch.

Third, if the rewriting step is performed by R4, the forward transformation moves one token to `finalized`. The backward transformation needs to ensure that R4 is still performed on the changed token sequence. For example, if the user replaces `hello` in `hello c` with `a b`, we need to check, starting from `a`, whether the only viable path is still to apply R4 twice to preprocess the replacing sequence, i.e., only `guard` of R4 evaluates to true on either `a b c` or `b c`. Here failures may occur. For example, if the user changes `hello` into `x`, where `hello` is not a macro but `x` is an object-like macro. Another example is that the user changes `GARRAY hello` into `GARRAY (hello)`, where `GARRAY` is a function-like macro. In this case, `GARRAY` was previously preprocessed by R4 because it is not followed by an open parenthesis, whereas in the changed sequence `GARRAY` is, and therefore the backward transformation fails at the rewriting step on `GARRAY`. Such failures are necessary as in both the above cases, there is no way to propagate back the changes.

Finally, the case of R3, which expands a macro invocation, is the most complex. R3 consists of two sub steps. We first try to reverse the two sub steps in an reversed order. If any of the sub step fails, we try to expand the macro invocation.

In the second sub step, the occurrences of parameters in a macro body are replaced by expanded arguments. The backward transformation maps the changes back to the arguments. There are two cases where an expansion may be triggered: (1) tokens that are not from the arguments are changed, and (2) multiple occurrences of the same parameter are changed to different values. For example, giving the following piece of code,

```
#define x 1
#define plus(a) a+a
plus(x);
```
(1.4)

if we changes `1 + 1` into `1 - 1` or `1 + 2`, an expansion will be triggered.

In the first sub step, the arguments are preprocessed. The backward transformation

recursively call `backward` to propagate the changes along the rewriting steps of each argument. Finally, a safety check is performed on the propagated changes: if the changed argument contains a comma at the top level (and the comma is not enclosed by a pair of parentheses) or any unmatched parenthesis, we trigger an expansion. This is because a comma or an unmatched parenthesis can break the original structure of arguments.

If the backward transformation on any of the sub step fails, we expand the macro invocation. The expansion will generate changes that replace the macro invocation with its expanded body. As an example, let us assume the second `1` in `1 + 1` is changed into `2` in code piece 1.4. After backward transformation, the propagated change will replace `plus` in `plus(x)` with `x + 2` and delete `(x)`.

The key to the expansion is how to construct the replacing sequence, `x + 2`. From the forward transformation we know that the unchanged preprocessed token sequence is `1 + 1`, and both `1`s are from the argument `x`. So we copy the rewriting steps of `x` to the locations of the two `1`s, and use the copied rewriting steps to propagate back the changes on the `1`s. The first `1` is not changed, so the original `x` is kept. The second `1` is changed into `2`, so the macro invocation `x` is also expanded, and we get the final text `x + 2`.

However, we cannot always copy the rewriting steps of an argument to the occurrences of its corresponding parameter. Basically, when we copy the rewriting step of an argument, we are assuming that the occurrence of the parameter can be replaced by its unpreprocessed argument, and all rewriting steps behave exactly the same as before. For example, when expanding `plus(x)`, we can replace the two occurrences of `a` by `x`, forming `x + x` where the two `x` expands exactly the same way as the argument `x`, forming `1 + 1`. However, this is not always the case. Let us consider the following code.

```
#define p (x)
#define pplus(x) plus x hello
pplus(p)
```
(1.5)

With the macros defined in code piece 1.4, the above code is preprocessed into to `1 + 1 hello`. However, if we need to expand `pplus(p)`, we cannot expand it into `plus p hello`, because it will only expand to `plus (1) hello`. This is because the use

of `p` instead `(x)` breaks the original macro invocation.

As a result, we have to add a safety check when we copy rewriting steps. We copy the rewriting steps of an argument only if none of the following conditions is satisfied.

- The preprocessed argument starts with a left parenthesis.
- The preprocessed argument contains a top-level comma.
- The preprocessed argument contains unclosed parentheses.
- The preprocessed argument becomes different if we preprocess it again.

The first three conditions correspond to situations similar to the above example: the preprocessed argument is used as part of another macro invocation in the expanded form, and replacing it with the unpreprocessed one will break the macro invocation. The last condition corresponds to the following case.

```
#define id(x) x
id(plus p)
```

This piece of code expands to `1 + 1`, but if we expands `id(plus p)` into `plus p`, the other macro expansions will be blocked. In other words, with macro invocation, an argument will be scanned twice, but when the macro invocation is expanded, an argument will only be scanned once. We need to make sure this does not affect correctness.

Correctness

We can directly reason that the backward transformation satisfies Requirements 1, 3, 4 and 5. Requirement 2 will later be evaluated by our experiments. Requirement 1 holds because we never propagate changes to macro definitions. Requirement 3 holds because if we happen to introduce a new macro invocation, we must have changed a normal token that should be preprocessed by R4 into part of a macro invocation, and the check in R4 will prevent such a change. Requirement 4 holds because we introduce new changes only when we expand a macro invocation, and we would not expand a macro invocation if nothing is changed. Requirement 5 has been reasoned throughout this subsection. To prove it formally, we need to inductively prove that each rewriting step either still behaves the same on the changed steps, or is replaced by a sequence of equivalent rewriting steps (the size of the sequence may be zero). A tricky point is when we copy rewriting steps of

arguments to the expanded method body, the copied rewriting steps may be mixed with later rewriting steps. We can show that, still by induction, the order of applying these rewriting steps can be changed without affecting the final result. We omit the details due to space limitations.

Optimization

In the current algorithm, we need to record the whole program after each rewriting step, even if only a small portion has changed. In the backward direction, we also need to shift all changes at each step. This contributes to a major performance penalty of the algorithm.

To optimize the algorithm, we first divide the original token sequence into a set of subsequences, where each subsequence is independently preprocessed by a set of rewriting steps. In this way, we can treat each subsequence as an independent program to perform the backward transformation, and then merge the changes. Giving a rewriting step `(src, i, ctx, r)`, if the token at `i` is not generated by a preceding rewriting rule, i.e., is from the original program, then we say the location right before `i` is a *splitting point*. With this definition, we guarantee that no rule application crosses a splitting point. We divide the original program into subsequences along the splitting points, and perform the backward transformation independently for each subsequence before merging the changes. When merging changes from two subsequences, we need to check that the preprocessed version of left sequence and the unpreprocessed version of the right sequence would not form a new macro invocation. This check was performed by R4 in the unoptimized algorithm.

For example, code piece 1.3 can be divided into three subsequences: the macro definition, `hello`, and `x`. Suppose `plus` is a macro defined earlier. If the user changes `hello` into `plus` and changes `100` into `(x)`, a new macro invocation is formed across the boundary and we should report a failure.

1.2.5 Extending to full CPP

In this sub section we discuss how we can extend the above algorithm to support full CPP. Due to space limitations, we only discuss the main ideas without the full details.

To support # and ## operators in forward preprocessing, we need to add two additional types of sub steps in R3, one for stringifying tokens and one for concatenating tokens. Furthermore, parameters used with these operators are replaced by their unpreprocessed forms but not their preprocessed forms, so in the first sub step we need to keep both the unpreprocessed and the preprocessed forms. We need to add three extensions to the backward transformation. First, we need to design the backward transformation for the two new sub steps. Second, we need to add a safety check to determine whether the unpreprocessed form and the preprocessed form are changed consistently. Finally, when expanding a macro, we should not try to recover # and ## operators as the two operators cannot exist on the top level.

To support #include in the forward preprocessing, we need to add another rewriting rule to support #include. In the backward transformation, we need to trace how changes are propagated to each file, and check changes propagated back from different #include directives of the same file are consistent.

Finally, CPP does not allow expanding a macro within its own expansion to prevent infinite loops. To support this, we need to add finer control of the context in our forward rewriting rules. When we are dealing with the tokens expanded from an invocation to macro `m`, the definition of `m` should be removed from the current context.

1.2.6 Extending to other types of changes

We have discussed how to deal with replacements. Now let us proceed to insertions and copying. Note that an insertion can be directly converted into a replacement. If we insert a token `y` before a token `x`, we can convert it as replacing `x` with `y x`. However, this direct conversion may cause unnecessary macro expansions. For example, if we insert `y` before `100` in the preprocessed code piece 1.3, `hello 100`, we should not expand `x` in the backward transformation, but if we model the insertion as replacing `100` with `y 100`, our

backward transformation will expand `x` because its body has been changed.

The above example exhibit an insertion at a splitting point, and such an insertion is guaranteed not to expand macros. To reduce the number of unnecessary expansions, We treat the inserted token sequence at a splitting point as an independent sequence and check whether only R4 is applicable to it, since there is no way to put it back if any other rewriting rules can be applied on the inserted sequence. Then we use the same method used in the optimized algorithm to merge the subsequences.

The copy operation is similar to insertion. The only difference is that copied segments may contain macro invocations and we shall try to recover these macro invocations. For this, we perform a special backward transformation. First, we generate changes on the preprocessed code that deletes all tokens except the segments being copied. Then we perform a backward transformation by ignoring rewriting steps with R1, R2, and R3 whose associated macro is not defined or defined differently at the target position. In this way we ensure that only macros that is defined at the target positions are recovered. Then, we insert the token sequence returned by the backward transformation to the target position.

1.3 Evaluation

1.3.1 Research Questions

In this section we focus on the following research questions.

- **RQ1: Macro Preservation.** According to requirement 2, our approach aims to preserve existing macro invocations. How does the strategy perform on actual programs? How does it compare to other techniques?
- **RQ2: Correctness.** Our approach is guaranteed to be correct according to requirements 4 and 5. How important is this correctness? How does our approach compare to other techniques that do not ensure correctness?
- **RQ3: Failures.** Our approach may report a failure when it cannot find a proper way to propagate the change. How often does this happen? Are the failures false alarms (there exists a suitable change but our approach cannot find it)?

To answer these questions, we conducted a controlled experiment to compare our approach with the two naive approaches described in Section 1.1.3 on a set of generated changes on Linux kernel source code. In the rest of the section we describe the details of the experiment.

1.3.2 Setup

Implementation

We have implemented our approach in Java by modifying JCPP[2], an open source C Preprocessor. We also implemented the two naive approaches in Section 1.1.3 for comparison. Our implementation and experimental data can be found on our web site[3].

Benchmark

Our experiment was conducted on the Linux kernel version 3.19. We chose Linux source code because Linux kernel is one of the most widely used software projects

[2] http://www.anarres.org/projects/jcpp/
[3] https://github.com/harouwu/BXCPP

implemented in C. It contains contributions from many developers, and has a lot of preprocessor directives and macro invocations.

To conduct our experiment, we need a set of changes on the Linux kernel code. Since we concern about how different backward transformations affect preprocessing, we generated changes only in functions that contain macro invocations. To do this, we first randomly selected 180 macros definitions from the kernel code. Since there are far more object-like macros than function-like macros, we would select very few function-like macros if we use pure random selection. So we controlled the ratio between object-like and function-like macros to be 1.5 : 1. Based on the selected macros, we randomly selected a set of functions which contain invocations to the macros. Finally, we randomly selected 8000 lines from the functions. There are in total 133 macro invocations in the selected lines.

Next we generated a set of changes on the selected lines. To simulate real world changes, we randomly generated two types of changes. The first type is token-level change, in which we randomly replace/delete/insert a token. The second type is statement-level change, in which we delete a statement or copy another statement to the current location. These two types of changes are summarized from popular bug-fixing approaches 1, 3, 21. The statement-level changes are directly used by GenProg 1 and RSRepair3. The token-level changes simulate small changes such as replacing the argument of a method or change an operators used in approaches such as PAR 21.

More concretely, we had a probability p to perform an operation on each token, where the operation is one of insertion, replacement and deletion, which had equal probability. The replacement was performed by randomly mutating some characters in the token. The insertion was performed by randomly copying a token from somewhere else. Similarly, we had a probability q to perform an operation on each statement, where the operation is copy or deletion. The copied statement was directly obtained from the previous statement. We recognized a statement by semicolon.

Different tools may have different editing patterns: a migration tool typically changes many places in a program, whereas a bug-fixing tool may change a few places to fix a bug. To simulate these two different densities of changes, we used two different set of

probabilities. For the high-density changes, we set `p = 0.33` and `q = 0.1`. For the low-density changes, we set `p = 0.1` and `q = 0.05`.

We generated ten sets of changes, five with high-density and five with low-density. The number of the changes generated for each set is shown in Table 1.2.

表 *1.2: Changes generated for the experiment*

Low Density	Set	1	2	3	4	5
	Changes	952	885	956	967	884
High Density	Set	6	7	8	9	10
	Changes	3133	3136	3088	3123	3048

Independent variables

We considered the following independent variables. (1) *Techniques*, we compared our approach with the two naive solutions, per-file and per-line. (2) *Density of changes*, we evaluated both on the five high-density change sets and the five low-density change sets.

Dependent variables

We considered two dependent variables. (1) *Number of remaining macro invocations*. We re-ran the preprocessor after the backward transformation, and counted how macro invocations are expanded during preprocessing. Since none of the techniques will actively introduce new macro invocations, the number of expanded invocations is the number of remaining invocations. To avoid noise from included files, we count only the macro invocations in the current file. (2) *Number of errors*. We re-ran the preprocessor, and compared the new preprocessed program with the previously changed program by Unix file-comparing tool `fc`. Every time `fc` reported a difference, we counted it as an error. (3) *Failures*. Our approach may fail to propagate the changes, and we record whether a failure is reported for each change set.

1.3.3 Threats to Validity

A threat to external validity is whether the results on generated changes can be generalized to real world changes. To alleviate this threat, we used different types of changes and different density of changes, in the hope of covering a good variety of real-world changes.

A threat to internal validity is that our implementation of the three approaches may be wrong. To alleviate this threat, we investigated all errors we found in the experiments, to make sure it is a true defect of the respective approach but not a defect in our implementation.

1.3.4 Results

表 1.3: Experimental Results

Low Density	Set	1	2	3	4	5
Our Approach	Macros	73	75	72	80	81
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	23	25	23	20	26
	Errors	6	7	6	7	7
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0
High Density	Set	6	7	8	9	10
Our Approach	Macros	47	51	53	48	44
	Errors	0	0	0	0	0
	Failures	n	n	n	n	n
Per-Line	Macros	9	7	7	8	10
	Errors	6	6	7	6	6
Per-File	Macros	0	0	0	0	0
	Errors	0	0	0	0	0

Row “Macros” shows the number of remaining macros. Row “Errors” shows the number of errors caused. Row “Failures” indicates whether a failure is reported in the backward transformation.

The result of our evaluation is shown in Table 1.3. We discuss the results with respect to the research questions below.

RQ1

As we can see, our approach preserves macro invocations. Per-line preserves very few macro invocations, while per-file, as we expected, preserves no macro invocations. We further investigated why per-line preserves so few macro invocations. One main reason we found is that there are usually multiple macro invocations per line, and per-line will expand all of them if any tokens in this line is changed.

RQ2

Our approach and per-file lead to no errors while several errors are caused by per-line. This is because there are quite a few macro invocations that cross multiple lines. These macros take expressions or statements as argument, which are usually too long to be included in one line.

RQ3

As discussed before, our approach may report a failure during the backward transformation. This is usually because the changes accidentally introduce a new macro invocation in the preprocessed code, where there is no way to satisfy PUTGET. However, we do not observe any such cases in our experiment. The reason is that macros usually have special names and it is not easy to collide with a macro name by copying or mutation. Note the other two approaches never report a failure, so the corresponding fields in Table 1.3 are left blank.

Although probably being rare in practice, theoretically our approach may report false alarms: our approach reports a failure but a correct change on the source program exists. For example, let consider the following code piece,

```
#define p (x)
plus p
```

where `plus` is the macro defined in code piece (1.4). After preprocessing, this code piece becomes `plus (x)`. If we change the last parenthesis into `) hello`, our approach reports a failure because first `p` will be expanded and then the expanded content forms a new macro invocation with `plus`. However, there exists a feasible change: replacing `p` with `hello p`.

1.4 Related Work

1.4.1 Bidirectional Transformation

Our work is inspired by research on bidirectional transformation. A classical scenario is the *view-update problem* 22–26 from database design: a view represents a database computed for a source by a query, and the problem comes when translating an update of the view back to a corresponding update on the source.

Languages have been designed to streamline the development of such applications involving transformations running bidirectionally. Notably the *lenses* framework 19, 27–35, covering a number of languages that provide bidirectional combinators as language constructs. A different approach is to mechanically transform existing unidirectional programs to obtain a backward counterpart, a technique known as *bidirectionalization* 15–17, 36–42. In the software model transformation literature, the underlying data to be transformed are usually in the form of graphs (instead of trees), and a relational (as oppose to functional) approach that specifies the bidirectional mappings between different model formats is more common 43–48. However, the requirement of our work goes beyond what these languages offer: in our framework, not only data, but also transformations (macros) are subject to bidirectional updates.

1.4.2 Analyzing and editing unpreprocessed C code

The C preprocessor poses a great challenge for static program analyses. The ability of producing a number of possible preprocessed variants causes a combinatorial explosion, rendering it infeasible to employee traditional tools that are designed to analyze a single variant at a time. Only until very recently, sound parsing and analyzing unpreprocessed C code is made possible through *family-based analyses* 49–51. Earlier tools have to resort to unsound heuristics or restrict to specific usage patterns 52–54.

Similarly, a lot of efforts in refactoring C code are devoted into dealing with multiple variants. Most approaches 13, 55–57 try to find a suitable model that represent both the C program and the preprocessor directives. A recent approach 58 suggests an alternative: perform refactoring on one variant and prevent the refactoring if problems may be caused

in other variants. This is based on the observation that changes on one variant seldom causes problems in other variant.

Unlike these approaches, our approach currently considers only one variant. In the future we may combine our approach with these approaches to deal with multiple variants. However, handling only one variant is already useful in many cases: (1) many programs, though with conditional compilation, do not have many variants; (2) as revealed by Overbey et al. 58, changes in one variant often do not cause problems in other variants.

1.4.3 Empirical studies on the C preprocessors

Over the years, there has been no shortage of academic empirical studies that are critical towards the C preprocessor 8, 59, 60, and replacements of CPP are proposed such as syntactical preprocessors 12, 61 and aspect-oriented programming 62–64 are plenty. However until present, there is no sign of any adoption of these alternatives in industry, with the C preprocessor is still being seen as the tool of the trade **Medeiros2015** .

1.5 Conclusion

Handling the C preprocessor in program-editing tools is difficult, as a result many tools either produce unsound results or give up on handling CPP entirely. In this paper we show that we can separate the concerns by using bidirectional transformations to deal with the preprocessor, so that program-editing tools may focus only on the preprocessed code, achieving a more modular design.

CPP also represents a family of transformation systems where the transformation program and the source data are bound together, for example bidirectional PHP 65. Existing approaches 65 often resort to ad-hoc treatment of the bidirectionalizing algorithm and correctness reasoning. The algorithm in this paper indicates a plausible systematic way of bidirectionalizing such systems: treating programs as data and bidirectionalize on the high-level operational semantics. We would like to see a general theory to be developed and applied to many different systems.

参考文献

[63] B. Adams *et al.* "*Can We Refactor Conditional Compilation into Aspects?*" In: *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*. Charlottesville, Virginia, USA: ACM, **2009**: 243–254. http://doi.acm.org/10.1145/1509239.1509274.

[22] F. Bancilhon and N. Spyratos. "*Update Semantics of Relational Views*". *ACM Trans. Database Syst.* **1981**, *6*(4): 557–575.

[52] I. D. Baxter and M. Mehlich. "*Preprocessor Conditional Removal by Simple Partial Evaluation*". In: *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. Washington, DC, USA: IEEE Computer Society, **2001**: 281–. http://dl.acm.org/citation.cfm?id=832308.837146.

[29] Ed. by G. C. Necula and P. Wadler. "*Boomerang: resourceful lenses for string data*". In: *POPL*. ACM, **2008**: 407–419.

[64] Q. Boucher *et al.* "*Tag and Prune: A Pragmatic Approach to Software Product Line Implementation*". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Antwerp, Belgium: ACM, **2010**: 333–336. http://doi.acm.org/10.1145/1858996.1859064.

[25] Y. Cui, J. Widom and J. L. Wiener. "*Tracing the Lineage of View Data in a Warehousing Environment*". *ACM Trans. Database Syst.* 2000-06: 179–227. http://doi.acm.org/10.1145/357775.357777.

[23] U. Dayal and P. A. Bernstein. "*On the Correct Translation of Update Operations on Relational Views*". *ACM Trans. Database Syst.* **1982**, *7*(3): 381–416.

[32] Ed. by J. Whittle, T. Clark and T. Kühne. "*From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case*". In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, **2011**: 304–318. http://dx.doi.org/10.1007/978-3-642-24485-8_22.

[8] M. D. Ernst, G. J. Badros and D. Notkin. "*An empirical analysis of C preprocessor use*". *Software Engineering, IEEE Transactions on*, **2002**, *28*(12): 1146–1170.

[26] L. Fegaras. "*Propagating updates through XML views using lineage tracing*". In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, 2010-03: 309–320.

[14] B. Fluri *et al.* "*Change distilling: Tree differencing for fine-grained source code change extraction*". *Software Engineering, IEEE Transactions on*, **2007**, *33*(11): 725–743.

[30] Ed. by J. Hook and P. Thiemann. "*Quotient lenses*". In: *ICFP*. ACM, **2008**: 383–396.

[19] J. N. Foster *et al.* "*Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem*". *ACM Trans. Program. Lang. Syst.* 2007-05.

[34] Ed. by J. Gibbons. "*Three Complementary Approaches to Bidirectional Programming*". In: *SSGIP*. Springer, **2010**: 1–46.

[53] A. Garrido and R. Johnson. "*Analyzing Multiple Configurations of a C Program*". In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, **2005**: 379–388. http://dx.doi.org/10.1109/ICSM.2005.23.

[55] A. Garrido and R. Johnson. "*Challenges of Refactoring C Programs*". In: *Proceedings of the International Workshop on Principles of Software Evolution*. Orlando, Florida: ACM, **2002**: 6–14. http://doi.acm.org/10.1145/512035.512039.

[13] A. Garrido and R. Johnson. "*Embracing the C preprocessor during refactoring*". *Journal of Software: Evolution and Process*, **2013**, *25*(12): 1285–1304. http://dx.doi.org/10.1002/smr.1603.

[50] P. Gazzillo and R. Grimm. "*SuperC: Parsing All of C by Taming the Preprocessor*". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing, China: ACM, **2012**: 323–334. http://doi.acm.org/10.1145/2254064.2254103.

[24] Ed. by S. Abiteboul and P. C. Kanellakis. "*Foundations of Canonical Update Support for Closed Database Views*". In: *ICDT*. Springer, **1990**: 422–436.

[47] Ed. by P. Hudak and S. Weirich. "*Bidirectionalizing graph transformations*". In: *ICFP*. ACM, **2010**: 205–216.

[48] S. Hidaka *et al.* "*GRoundTram: An Integrated Framework for Developing Well-behaved Bidirectional Model Transformations*". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, **2011**: 480–483. http://dx.doi.org/10.1109/ASE.2011.6100104.

[33] M. Hofmann, B. Pierce and D. Wagner. "*Edit Lenses*". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Philadelphia, PA, USA: ACM, **2012**: 495–508. http://doi.acm.org/10.1145/2103656.2103715.

[27] Ed. by N. Heintze and P. Sestoft. "*A programmable editor for developing structured documents based on bidirectional transformations*". In: *PEPM*. ACM, **2004**: 178–189.

[20] ISO/IEC JTC 1, SC 22, WG 14. *Programming languages – C*, 2011-04.

[49] C. Kästner *et al.* "*Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation*". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. Portland, Oregon, USA: ACM, **2011**: 805–824. http://doi.acm.org/10.1145/2048066.2048128.

[21] D. Kim *et al.* "*Automatic patch generation learned from human-written patches*". In: *ICSE*, **2013**: 802–811.

[9] E. Kohlbecker *et al.* "*Hygienic macro expansion*". In: *LISP and functional programming*, **1986**: 151–161.

[11] J. Korpela. *Using a C preprocessor as an HTML authoring tool*, **2000**.

[2] C. Le Goues *et al.* "*A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each*". In: *ICSE*, **2012**: 3–13.

[1] C. Le Goues *et al.* "*GenProg: A generic method for automatic software repair*". *Software Engineering, IEEE Transactions on*, **2012**, *38*(1): 54–72.

[10] B. Lee *et al.* "*Marco: Safe, expressive macros for any language*". In: *ECOOP*. Springer, **2012**: 589–613.

[5] J. Li *et al.* "*SWIN: Towards Type-Safe Java Program Adaptation between APIs*". In: *PEPM*, **2015**: 91–102.

[60] J. Liebig, C. Kästner and S. Apel. "*Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code*". In: *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*. Porto de Galinhas, Brazil: ACM, **2011**: 191–202. http://doi.acm.org/10.1145/1960275.1960299.

[51] J. Liebig *et al.* "*Scalable Analysis of Variable Software*". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg, Russia: ACM, **2013**: 81–91. http://doi.acm.org/10.1145/2491411.2491437.

[62] D. Lohmann *et al.* "*A Quantitative Analysis of Aspects in the eCos Kernel*". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. Leuven, Belgium: ACM, **2006**: 191–204. http://doi.acm.org/10.1145/1217935.1217954.

[42] K. Matsuda and M. Wang. "*"Bidirectionalization for Free" for Monomorphic Transformations*". *Science of Computer Programming*, **2014**. http://www.sciencedirect.com/science/article/pii/S016764

[40] Ed. by R. Peña and T. Schrijvers. "*Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem*". In: *PPDP*. ACM, **2013**: 297–308.

[39] Ed. by M. Felleisen and P. Gardner. "*FliPpr: A Prettier Invertible Printing System*". In: *Programming Languages and Systems*. Springer Berlin Heidelberg, **2013**: 101–120. http://dx.doi.org/10.1007/978-3-642-37036-6_6.

[18] Ed. by A. D. Gordon. "*A Grammar-Based Approach to Invertible Programs*". In: *ESOP*. Springer, **2010**: 448–467.

[15] Ed. by R. Hinze and N. Ramsey. "*Bidirectionalization transformation based on automatic derivation of view complement functions*". In: *ICFP*. ACM, **2007**: 47–58.

[12] B. McCloskey and E. Brewer. "*ASTEC: A New Approach to Refactoring C*". In: *ESEC/FSE-13*, **2005**: 21–30.

[7] N. Meng, M. Kim and K. S. McKinley. "*Systematic Editing: Generating Program Transformations from an Example*". In: *PLDI*, **2011**: 329–342.

[28] Ed. by W.-N. Chin. "*An Algebraic Approach to Bi-directional Updating*". In: *APLAS*. Springer, **2004**: 2–20.

[4] H. D. T. Nguyen *et al.* "*SemFix: Program repair via semantic analysis*". In: *ICSE*, **2013**: 772–781.

[43] I. (Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*.

[58] J. L. Overbey, F. Behrang and M. Hafiz. "*A Foundation for Refactoring C with Macros*". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, **2014**: 75–85. http://doi.acm.org/10.1145/2635868.2635908.

[54] Y. Padioleau. "*Parsing C/C++ Code Without Pre-processing*". In: *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. York, UK: Springer-Verlag, **2009**: 109–125. http://dx.doi.org/10.1007/978-3-642-00722-4_9.

[6] Y. Padioleau *et al.* "*Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers*". In: *PLOS*, **2006**.

[3] Y. Qi *et al.* "*The strength of random search on automated program repair*". In: *ICSE*, **2014**: 254–265.

[35] R. Rajkumar *et al. Lenses for Web Data*, **2013**.

[45] Ed. by E. Mayr, G. Schmidt and G. Tinhofer. "*Specification of graph translators with triple graph grammars*". In: *Graph-Theoretic Concepts in Computer Science*. Springer Berlin Heidelberg, **1995**: 151–163. http://dx.doi.org/10.1007/3-540-59071-4_45.

[46] Ed. by H. Ehrig *et al.* "*15 Years of Triple Graph Grammars*". In: *Graph Transformations*. Springer Berlin Heidelberg, **2008**: 411–425. http://dx.doi.org/10.1007/978-3-540-87405-8_28.

[59] H. Spencer and G. Collyer. *#ifdef Considered Harmful, or Portability Experience With C News*, **1992**.

[57] D. Spinellis. "*Global Analysis and Transformations in Preprocessed Languages*". *IEEE Trans. Softw. Eng.* 2003-11: 1019–1030. http://dx.doi.org/10.1109/TSE.2003.1245303.

[44] P. Stevens. "*Bidirectional model transformations in QVT: semantic issues and open questions*". *Software & Systems Modeling*, **2010**, *9*(1): 7–20. http://dx.doi.org/10.1007/s10270-008-0109-9.

[56] M. Vittek. "*Refactoring browser with preprocessor*". In: *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, 2003-03: 101–110.

[16] Ed. by Z. Shao and B. C. Pierce. "*Bidirectionalization for free! (Pearl)*". In: *POPL*. ACM, **2009**: 165–176.

[17] J. Voigtländer *et al.* "*Combining syntactic and semantic bidirectionalization*". In: *ICFP*, **2010**: 181–192.

[37] J. Voigtländer *et al.* "*Enhancing semantic bidirectionalization via shape bidirectionalizer plugins*". *J. Funct. Program.* **2013**, *23*(5): 515–551.

[36] Ed. by M. M. T. Chakravarty, Z. Hu and O. Danvy. "*Incremental updates for efficient bidirectional transformations*". In: *ICFP*. ACM, **2011**: 392–403.

[41] Ed. by W.-N. Chin and J. Hage. "*Semantic bidirectionalization revisited*". In: *PEPM*. ACM, **2014**: 51–62.

[31] Ed. by C. Bolduc, J. Desharnais and B. Ktari. "*Gradual Refinement: Blending Pattern Matching with Data Abstraction*". In: *MPC*. Springer, **2010**: 397–425.

[38] M. Wang *et al.* "*Refactoring pattern matching*". *Sci. Comput. Program.* **2013**, *78*(11): 2216–2242.

[65] X. Wang *et al.* "*Automating presentation changes in dynamic web applications via collaborative hybrid analysis*". In: *FSE*, **2012**: 16.

[61] D. Weise and R. Crew. "*Programmable Syntax Macros*". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico, USA: ACM, **1993**: 156–165. http://doi.acm.org/10.1145/155090.155105.

附录 A　附件

pkuthss 文档模版最常见问题:

在最终打印和提交论文之前，请将 pkuthss 文档类选项中的 `colorlinks` 改为 `nocolorlinks`，因为图书馆要求电子版论文的目录必须为黑色，且某些教务要求打印版论文的文字部分为纯黑色而非灰度打印。

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 `openany` 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

致谢

pkuthss 文档模版最常见问题:

在最终打印和提交论文之前，请将 pkuthss 文档类选项中的 `colorlinks` 改为 `nocolorlinks`，因为图书馆要求电子版论文的目录必须为黑色，且某些教务要求打印版论文的文字部分为纯黑色而非灰度打印。

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 `openany` 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在 □ 一年 / □ 两年 / □ 三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日