Bidirectional Proprocessor

Yiming Wu

October 16, 2014

1 Introduction

Program repair has long been an essential problem with heated discussion, since most software systems inevitably have bugs that need to be fixed. Developers for large software projects must confirm, triage, and localize defects before fixing these bugs and validating the fixes. To reduce human manual labor, numbers of automatic patch generation techniques or tools have been proposed.

Among these automatic tools, C oriented fixing tools count most of them because of the prevalence of C in open-scource or enterprise software projects. Weimer et al. proposed a population-based technique and developed **GenProg**, leveraging genetic programming[]. Nguyen et al. presented **SemFix** based on symbolic execution, constraint solving and program synthesis[]. Kaleeswaran et al. exploited statistical correlation analysis to identify perspective bugs[]. Qi et al. provided a prototype tool **RsRepair** using random search[].

However, all C oriented automatic repair tools face a common problem: preprocessing. Although preprocessing is not explicitly stated in their papers, automatic tools like **GenProg** abstract the C program into an AST which might include preprocessing. It is impossible to know the whole picture of a C program without preprocessing to get rid of predefined macro and conditional instructions. Developers for automatic repair tools have to write a qualified preprocessor before they can truly start core analysis of the program. This will cost meaningless labor and increase chances of bugs within bug-repair tools themselves.

Furthermore, patches generated by automatic repair tools may confuse their users because these patches are targeted for preprocessed code. (Need an exmaple here where patches loses macro name).

In this paper, we present a novel bidirectional preoprocessor **BXP** that can tackle preprocessing problem for automatic repair tools. Our bidirectional preprocessor performs like a standard preprocessor for automatic repair tools to get preprocessed code. It can also generate patches for *unpreprocessed* code according to the patches for *preprocessed* code from automatic program repair tools. With the help of **BXP**, we believe the develop of automatic C program repair tools can been greatly simplified. In addition, the patches generated by repair tools are now more agreeable and gracefull.

(This paragraph is left for evaluation)

(This paragraph is left for contribution list)

2 Bidirectional Preprocessor BXP

Many C oriented tools, for example, bug-fixing tools, operate on preprocessed code. However, programmers usually expect changes on original code, instead of preprocessed code. To make up the gap between changed preprocessed code and original code, we believe we need an inverse preprocessor to transform the new preprocessed code into a new *unpreprocessed* code.

To illustrate the use of inverse preprocessor more clearly, imagine a scenario where a programmer exploit a bug fixing tool to improve his code. A preprocessor `PP` : $OCode \mapsto PCode$ will take original code OC as input, discard all the macro, merge include files and give out preprocessed code PC. The bug fixing tool `BF` : $PCode \mapsto PCode$ will change the preprocessed code to PC', but not retriving those macros and include instructions back. The new preprocessed code PC is far different from original code OC. Programmer may get confused without those macro information he defined. Thus, we need a inverse preprocessor `IPP` : $PCode \mapsto OCode$ to make up the gap between PC and OC. The inverse preprocessor will produce new code OC' which looks as similar to OC as possible.

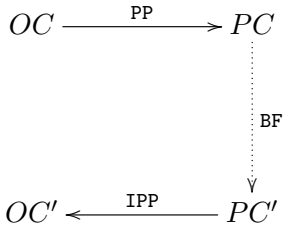

Figure 1: A bug fixing scenario.

With Inverse Preprocessor, we believe we can greatly expand usability of C oriented tools operating on preprocessed code. Besides that, Inverse Preprocessor can function as a platform for future C oriented work. It can spare the labor of code analysis programmer from concerning about macros and include path.

3 MEDG: Macro Expansion Dependency Graph

To retrive macro calls in original code, Inverse Preprocessor needs the information of all the macro expansions in the code. During preprocessing, out preprocessor `PP` record those information with the help of MEDG: Maco Expansion Dependency Graph.

To grab the main idea of MEDG, let us start with a quick example. In the example code, there are four macro definitions and one macro call. Preprocessor `PP` takes in original example code and produces preprocessed code as well as MEDG.

```
#define g(a,b) a+b
#define h(a) a*a
#define NUM 3
#define f(a,b) g(a,b)+h(a)+NUM

...
f(a,b)
...
```

Figure 2: A Quick Example Code

Like standard preprocessor, our Preprocessor `PP` iterates through the whole context over and over until there is no more macro expansion. Each time Preprocessor `PP` expands a macro call, it will record dependency connection edge between macro-call token and expanding tokens. Each connection edge is encoded with three domain: macro name, parameter list and expansion position in original code. These information help Inverse Preprocessor getting the original code back quickly and elagently.

The figure 3 shows the MEDG of the quick example, it goes in two iteration. In the first iteration,

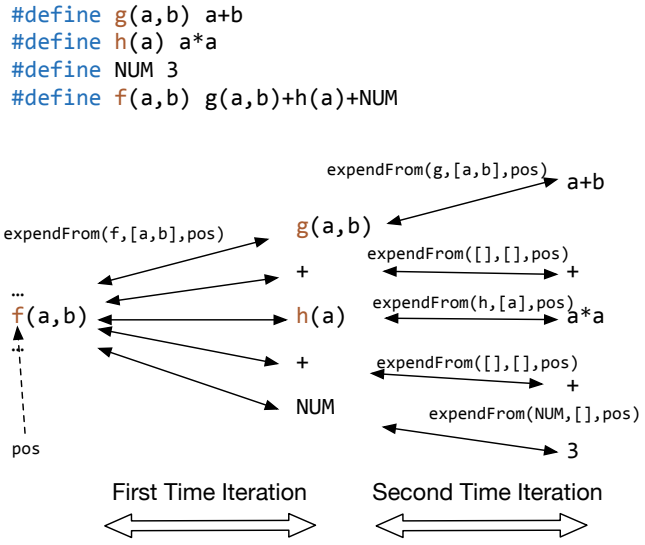

Figure 3: MEDG of Quick Example.

Here is our pseudo code of Preprocessor `PP`.

Algorithm 1 Preprocessor

Input: Original Code
Output: Preprocessed Code with MEDG
1: **function** IterateOnce(ctx)
2: **end function**

4 Deal with Changes on Code with MEDG

In this section, we will show how to deal with changes on preprocessded code and transform the changes back after we record macro call information with MEDG. We will first give out an example, then discuss the pesudo code for dealing with changes on code.

```
#define g(a,b) a+b
#define h(a) a*a
#define NUM 3
#define f(a,b) g(a,b)+h(a)+NUM
```

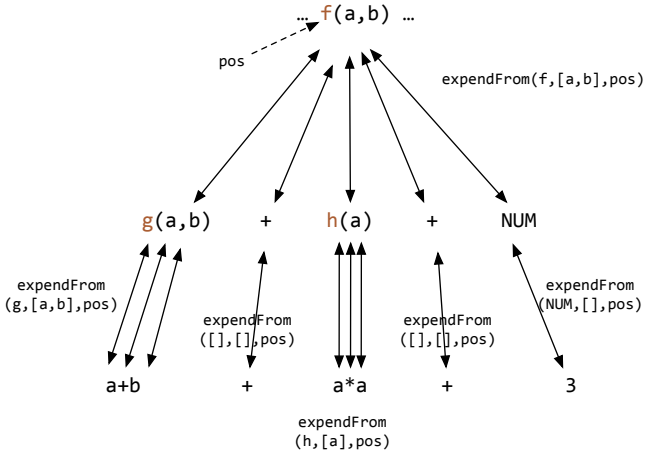

Figure 4: MEDG without Changes.

```
#define g(a,b) a+b
#define h(a) a*a
#define NUM 3
#define f(a,b) g(a,b)+h(a)+NUM
```

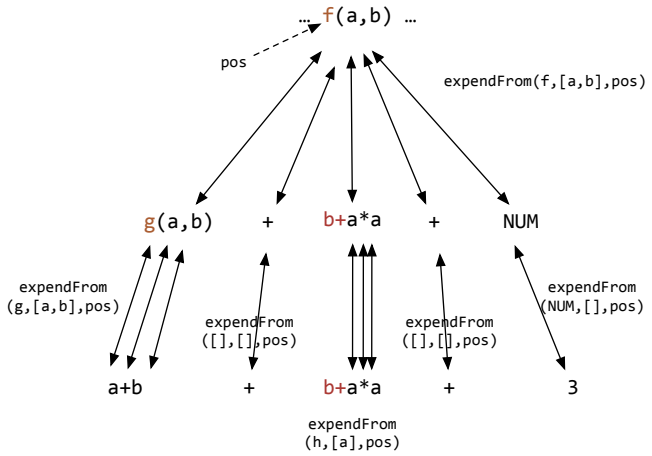

Figure 5: Adding Tokens to the Preprocessed Code.

Algorithm 2 a

Input: a
Output: a

```
1: function MergerSort(Array, left, right)
2:     result ← 0
3:     if left < right then
4:         middle ← (left + right)/2
5:         result ← result+ MergerSort(Array, left, middle)
6:         result ← result+ MergerSort(Array, middle, right)
7:         result ← result+ Merger(Array, left, middle, right)
8:     end if
9:     return result
10: end function
11:
12: function Merger(Array, left, middle, right)
13:     i ← left
14:     j ← middle
15:     k ← 0
16:     result ← 0
17:     while i < middle and j < right do
18:         if Array[i] < Array[j] then
19:             B[k++] ← Array[i++]
20:         else
21:             B[k++] ← Array[j++]
22:             result ← result + (middle − i)
23:         end if
24:     end while
25:     while i < middle do
26:         B[k++] ← Array[i++]
27:     end while
28:     while j < right do
29:         B[k++] ← Array[j++]
30:     end while
31:     for i = 0 → k − 1 do
32:         Array[left + i] ← B[i]
33:     end for
34:     return result
35: end function
```