CSE 503: Project Proposal: Semantical Merge

Ji Luo (Student No. 1940053; UW NetID `luoji`)

23 April 2019

Motivation

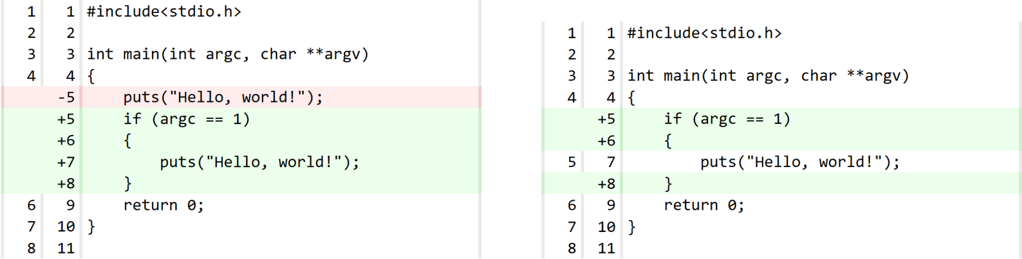
Version control systems (VCSes) are ubiquitously adopted in modern software development for codebase tracking. In distributed VCSes like Git, the history of a project is modeled as a directed acyclic graph, where each revision (commit) is a vertex pointing to its parents. Coding by a programmer corresponds to creating vertices with one parent — the parent represents the version on which the change is based, and the new vertex represents the updated version. When the software is developed by a team, team members contribute to the codebase concurrently, resulting in a graph where one vertex might have multiple children. Consolidation of team effort is achieved by merging multiple revisions. Automatic merging of revisions makes collaboration more effective, and finding a good merging algorithm is of great interest.

Computing the difference (`diff`) between revisions is closely related to merging. A widely used strategy (if not the only one) for merging is to compute the difference between the two vertices to be merged with their nearest common ancestor, then combine the differences to produce the result. Therefore, one can improve the merging algorithm by improving the `diff` algorithm, the way differences are combined, or both. In addition to being a stage of the merging algorithm, `diff` is often displayed so that a programmer can read out the changes between revisions, making it useful on its own.

Git computes `diff` of text files by line (and binary files as a whole), which naturally leads to line-based merging strategy. The method takes advantage of the fact that code files are often formatted so that each line roughly corresponds to a part in the semantical hierarchy (both for semantics of the programming language and semantics to human). However, line-based methodology has its problems in both computing `diff` and merging. The reason is that lines are a bad proxy of the semantical hierarchy.

Many programming languages ignore the difference between a whitespace character and consecutive whitespace characters, so line separators and indentation can be inserted or removed without changing the semantics of the program. Insertion and removal of line separators are often done for aesthetic reasons (e.g., to keep each line shorter than 80 characters). Insertion and removal of indentation take place with refactoring or change of logic external to the chunk being indented/unindented. When a line-based `diff` is produced, such less important changes are mixed with other semantically significant changes, making the `diff` less easy to understand for human (e.g., in Figure 1, the indentation is mixed with the real change — the introduction of a new condition). Another problem of line-based `diff` is that it sometimes matches irrelevant lines that are identical by accident (Figure 2).

Bad `diff`s make merging difficult. The previous example continued, if a block of code is indented in one revision and has an expression changed in the other, Git merging algorithm gives up and reports a merge conflict. For another example, even if there is no merge conflict, line-based (or more generally, text-based) merging might yield an undesirable result. Suppose a function is renamed in one revision, and in the other revision, another call to the function is added (using its old name). Merging the two revisions result in a state where the function is renamed, yet there is one call to that method using its old name. This subtle bug can be hard to notice if the old name happens to be reused in the first revision.



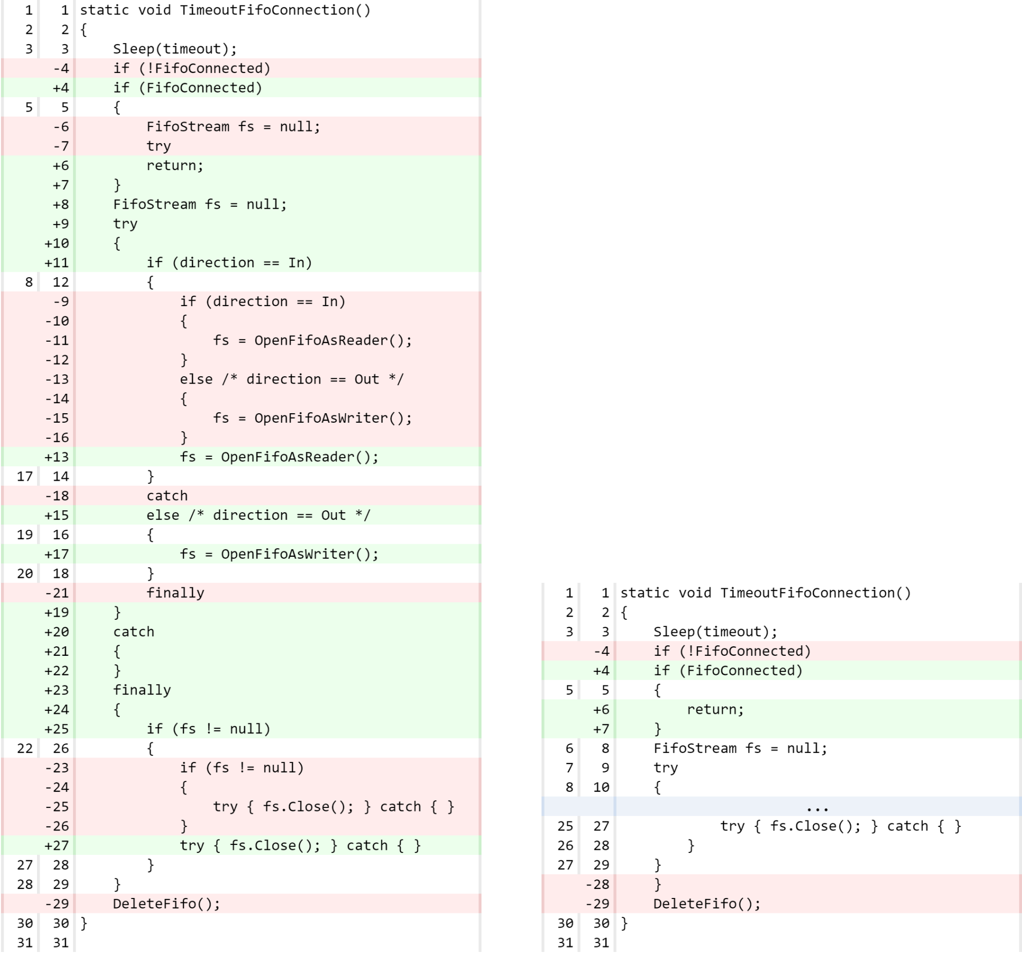
```
 1   1 #include<stdio.h>
 2   2
 3   3 int main(int argc, char **argv)
 4   4 {
    -5     puts("Hello, world!");
    +5     if (argc == 1)
    +6     {
    +7         puts("Hello, world!");
    +8     }
 6   9     return 0;
 7  10 }
 8  11
```

(a) A bad `diff`.

```
 1   1 #include<stdio.h>
 2   2
 3   3 int main(int argc, char **argv)
 4   4 {
    +5     if (argc == 1)
    +6     {
 5   7         puts("Hello, world!");
    +8     }
 6   9     return 0;
 7  10 }
 8  11
```

(b) A good `diff`.

Figure 1: Example exhibiting undesired result from line-based `diff`. Created with `https://strcmp.cc/`.



```
 1   1 static void TimeoutFifoConnection()
 2   2 {
 3   3     Sleep(timeout);
    -4     if (!FifoConnected)
    +4     if (FifoConnected)
 5   5     {
    -6         FifoStream fs = null;
    -7         try
    +6         return;
    +7     }
    +8     FifoStream fs = null;
    +9     try
   +10     {
   +11         if (direction == In)
 8  12         {
    -9             if (direction == In)
   -10             {
   -11                 fs = OpenFifoAsReader();
   -12             }
   -13             else /* direction == Out */
   -14             {
   -15                 fs = OpenFifoAsWriter();
   -16             }
   +13             fs = OpenFifoAsReader();
17  14         }
   -18         catch
   +15         else /* direction == Out */
19  16         {
   +17             fs = OpenFifoAsWriter();
20  18         }
   -21         finally
   +19     }
   +20     catch
   +21     {
   +22     }
   +23     finally
   +24     {
   +25         if (fs != null)
22  26         {
   -23             if (fs != null)
   -24             {
   -25                 try { fs.Close(); } catch { }
   -26             }
   +27             try { fs.Close(); } catch { }
27  28         }
28  29     }
   -29     DeleteFifo();
30  30 }
31  31
```

(a) A bad `diff`.

```
 1   1 static void TimeoutFifoConnection()
 2   2 {
 3   3     Sleep(timeout);
    -4     if (!FifoConnected)
    +4     if (FifoConnected)
 5   5     {
    +6         return;
    +7     }
 6   8     FifoStream fs = null;
 7   9     try
 8  10     {
                       ...
25  27                 try { fs.Close(); } catch { }
26  28             }
27  29         }
   -28     }
   -29     DeleteFifo();
30  30 }
31  31
```

(b) A better `diff`.

Figure 2: Example exhibiting result that forgets language structure from line-based `diff`.
The change is from deep nesting to early return.
Note how the block delimiters for `if-else` and `try-catch-finally` are mingled.

Project Plan

[AGMO17] uses tree-based `diff` and merging algorithms. We plan to pursue and push forward the methodology based on ASTs of the code. We will use static semantics, a step forward from ASTs, to do `diff` and merging.

The `diff` algorithm will have a subprocedure that is similar to the first few stages of a compiler. The subprocedure takes in source code, parses it into AST and perform syntax-directed translation on it to obtain static semantics of the code (which functions are defined, which the arguments are, which function this expression calls and what parameters are passed, etc.). The `diff` algorithm, given two sets of code files, compares the static semantics, and determines and outputs the difference. The difference is not a textual one, but a symbolic one. To merge two revisions, the merging algorithm compares both against their nearest common ancestor, combines the symbolic difference, and applies the combined difference to the ancestor.

Analyzing static semantics isn't an easy-to-program task. The C programming language has a good trade-off between language complexity and expresiveness, and it is also a popular programming language. So in this project, we plan to implement the algorithms that `diff`s and merges C source code.

We plan to evaluate the project by testing the algorithms on several publicly available repositories that mainly consist of C code files with the following metrics:

- The length of `diff`s produced for neighboring revisions (implicitly compared to the necessary length).
- The frequency and size of merge conflicts.
- The consistency of automatic merging results with manual merging results.
- The frequency of cases where the algorithm raises a merge conflict, yet for which Git silently produces a textually correct and semantically incorrect merge.

Lastly, we point out the novelties of our project:

- It explicitly decomposes merging into 2 stages: producing `diff`s and combining `diff`s, and proposes the methodology of improving `diff` (structurally, like from lines to trees; or within the same structure) to improve merging algorithm.
- It does not try to use a textual encoding for the internal structure to be `diff`ed (as is done for ASTs in [AGMO17]). Rather, it operates natively on the structured (in-memory) representation of the semantics.

References

[AGMO17] Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. Precise version control of trees with line-based version control systems. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, pages 152–169, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.