Recurme

Authors: Carl Ross – cross95, Grant Hughes - gyhughes

Motivation

Recursion is an incredibly powerful tool that allows us to rapidly and elegantly develop solutions to problems that would otherwise require complex and convoluted iterative blocks. However, recursion holds fundamental disadvantages to such iterative blocks. Most notably that recursion tends towards higher runtimes. This is due to the overhead of maintaining the stack which also contributes to software which bears a heavier memory. Furthermore, recursive functions, in particular in the scope of the Java language, are typically bound by stricter inputs constraints. For example, there exists a large set of problems that provided with very large data sets, recursive solutions will result in stack overflow exceptions. Conversely, imperative loop solutions to those same problems will not result in any sort of overflow exception. Instead, they will simply perform the computation over a longer time period. That is, there is no inherit limit to the input size to these problems for an iterative solution.

We take a moment to acknowledge that a handful of languages implement compile time or even runtime Tail Recursion Optimization. However, Java, for a multitude of reasons both based in particular legacy design decisions and perpetuated by a bias towards imperative loops does not implement such compiler optimizations.

Therefore, there exists a substantial collection of situations in which Java developers elect to utilize recursion in lieu of imperative loops purely for ease of development. While in deployment it would be preferable that such recursive solutions were converted to their imperative counterparts.

Approach

In order to develop this recursive tool, we propose an Eclipse plugin. We elect a developer utilized plugin over strategies further down the deployment chain (i.e. as components of the compiler or JVM interpreter) as this allows greater control. Users of our plugin will be able to selectively convert recursive functions that they believe to be good candidates for refactoring. In this manner we keep control in the hands of the developer instead of forcing our own doctrine. This is particularly crucial for the set of problems to which a recursive solution in Java is truly superior to that of an iterative. For which it would be improper to force a refactoring. Likewise, by changing the actual source code we maximize portability. For example, if a large team is collaborating on a single project individual members may elect to utilize our tool without requiring universal adoption.

```
1  public static int collectiveSum(int i) {
2      if(i == 0) {
3          return i;
4      } else {
5          return i + collectiveSum(i - 1);
6      }
7  }
```

Figure 1 Pre-Refactor

An example usage case of our plugin can be seen in figures 1 and 2.

Edit->Refactor->To Iterative

```
1   public static int collectiveSum(int i) {
2       /* Recurme Autoconversion from recursive:
3           public static int collectiveSum(int i) {
4               if(i == 0) {
5                   retur i;
6               } else {
7                   return i + collectiveSum(i - 1);
8               }
9           }
10      */
11      int j = 0;
12      for(int k = i; k >= 0; k--) {
13          j += k;
14      }
15      return k;
16  }
```

Figure 2 Post-Refactor

Behind the scenes our plugin will work to interpret the selected function between figures 1 and 2. The specifications of this interpretation are yet to be determined. However, we anticipate a conversion of recursive logic to a decision tree before reverting said decision tree into iterative code. Utilizing primarily reflective logic (interpretation of the literal text).

Challenges and Risks

One of the biggest challenges to overcome is properly re-implement the recursive functions into iterative functions. As we stated earlier, our plugin is supposed to modify the function to lower the runtime and memory usage. If we do not re-implement the function correctly, then the code will not work. Even if the code does work, there may be hidden side-effects that weren't there before. If we want our plugin to gain any attention, then it has to be safe, secure, and effective. Users do not want to use software that may or may not work. Users want to use software that always works and improves their coding experience.

We have to find a way to produce an improved version of the function that improves the performance of the code. Our plugin has to properly isolate the recursive task and the base of the recursive function. Once our plugin refactors the function, it will silently run a quick sanity check to make sure that the new iterative function is in fact a working, improved version of the recursive function. If the new function does work, then it replaces the recursive function. If the new function does not work, it will not replace the function, but will prompt the user to take the look at the refactored suggestion. This way, the user can approve or modify the iterative function before replacing the older function. We believe that this feature will limit the inherent risks of using automated refactoring tools.