Partially Automated Refactoring

Daniel Speyer

December 18, 2004

Abstract

Modern application software can easily reach hundreds of thousands if not millions of lines of code. Keeping track of all of this, and ensuring that all of it is properly secured, debugged and otherwise of adequate quality can be extremely difficult. Generally, it is desirable to limit the length of a program as much as possible. One way to do this is to eliminate duplicative sections, but this is very difficult for a large code-base. Therefore, I propose a way to automate the task, in which every piece of a program is entered into a hash tree, and those which match are unified into new functions.

1 Introduction

Modern software is extremely large, so much so that its size causes difficulties in maintaining it. It is well established that bugs grow with physical lines on code (sources differ over whether the growth is linear or quadratic). Larger codebases are also more difficult to maintain, because it is more difficult to keep track of all that happens within them.

One thing that can be done to bring down code size is the elimination of duplicate code. It is estimated that as much as 10% of in-house code is duplicative. Software written for public consumption may be somewhat less so. Along with the ordinary problems of code size, duplicate code has a special tendency towards bugs: a programmer may make a change in an underlying data structure, adjust one copy, and think he had adjusted the other as well.

It is difficult to detect duplicate code by hand, unless one remembers creating it, because it is necessary to compare every part of the codebase with every other part. Even if a programmer has the time to search through it all by hand, by the time she reaches the second appearance of a snippet, will she remember the first?

Recognizing exact duplicate code is simple. What is more difficult is recognizing *near misses*: code which is similar enough that it aught to be made into a function but still not quite identical. Several systems exist for detecting these, but all of them are based on specific types of near misses. Here I propose a fully general algorithm.

2 The Algorithm

Sections of code which can and should be unified into functions are not likely to be byte-for-byte identical. Several steps are needed to make unifiable code segments fall into the same hash bucket.

Firstly, we will not work with the code directly, but with its abstract syntax tree. Next, we will rearrange sequential code. Then we will rename variables, fields and constants. Finally, we will convert all extraneous structures to simple ones.

Once these simplifications are done, we will add wildcards so as to catch near misses. We will then place the resulting values in a hash table and take note of conflicts. Those nodes whose hashes match will be fed into a unifier.

2.1 Abstract Syntax Trees

Converting code into abstract syntax trees is a task best left to existing compilers, and that is what we do here. All that is worth mentioning is that this takes place *after* the code is preprocessed. This simplifies matters for us later, as macros do not fit neatly into syntax trees, but it does lose a small amount of information, particularly by replacing symbolic constants with their values.

2.2 Sequential Code

In most abstract syntax trees, including those output by gcc, sequential code (code in which one statement is executed after other and the return values discarded) is represented in nested pairs. This does not work well for our purposes, as the subsections which can be unified may not be ones which are single units in the tree. Therefore our first task is to flatten the sequences, creating parent nodes for each one, and giving the parent nodes an arbitrary number of statements as children.

Once this is done, it is necessary to reorder the statements. Each statement must be examined in terms of side effects and data-flow, and those which can be safely reordered should be sorted. The sorting should be done in terms of the set of node type, ignoring all else, because it is node types which cannot be unified. In analyzing dataflow, it is probably necessary to regard function calls as potentially accessing and overwriting anything, which does limit the re-ordering's usefulness. It may be possible to do better.

Once the sequence is prepared, it must be divided up. Every subsequence must be made into a temporary sequence node and fed into the next step. This increases the effort quadratically, but it ensures that sequences will match even if surrounded by unmatching code.

2.3 Renaming

When we receive a subtree for hashing, we must eliminate variable names. However, it is significant whether or not two variables are the same. The snippets "`x=x+1`" and "`y=y+1`" are the same but "`x=y+1`" is not. To address this, variables are numbered by their order of appearance in an in-order traversal.

Similarly, all constants are recorded simply as types. Constants can easily be passed as arguments, so differences between them may be safely ignored.

Treating field names like variables is more questionable, but examination of existing code shows it to be useful. C does not support fields as native first class objects, and there is no way to iterate over them. It is perhaps because of this that programmers often iterate over them manually. This creates large amounts of duplicative code, and a significant burden in terms of keeping it all up-to-date. I address this by creating the macros:

```
#define ATTROF(field,type)        ((int)(&(((type *)0)->field)))
#define GETATTR(var,offset,type) (*(type*)(((void*)(&var))+offset))
```

which make fields passable as arguments. How much duplicate code must be present to compensate for the added conceptual burden of these macros is an open question.

2.4 Expression Conversion

Expression conversion is entirely straightforward. C provides several equivalent operations, such as `for` and `while`. We convert them to the simplest ones (`for` to `while` in this case. The benefits of this are small, but it requires little effort.

2.5 Wildcarding

Once all the transformations on the abstract syntax subtree are performed, we are left with trees that are in canonical form. It is now necessary to deal with *near misses*. These are blocks of code that are very similar but have one difference that is not of any special form. We still want to match these, so we use wildcards.

Once given a subtree, we iterate through all the nodes with low enough degree (ie are *far* enough from the root) and hash the tree with the current node replaced by a wildcard symbol. Code segments which are near misses will match when the differing code is wildcarded out in both cases.

2.6 Unification

The hash table simply gives us lists of nodes which are suspected of being equivalent. It is then the job of the unifier to create a function that can replace all of them. Most of this is straightforward: generate code for what matches; generate arguments for non-local variables and non-matching constants. The only complexity is near misses, in which case the unifier creates an argument called `mode` which determines which will be run. Each call to the new function will have a different value of mode.

The generic mode variable, the other variables named `v1`, `v2`, etc. and the numeric values of preprocessor constants all combine to make the generated code somewhat unreadable. At present, this is left to the user to deal with. The code to be replaced is referenced by file and line number, and each block of code is marked regarding which argument to the created function it corresponds to. Hopefully, this makes the code fairly easy for the user to clean up. In the test cases I've run so far, it has done so.

3 Implementation Notes

The existing implementation works in two stages (or three, depending on environment). The first stage is preprocessing, done with the standard `gcc -E` command. This stage may be omitted for simple code, but it usually needed for code with complex header dependencies. Then the abstract syntax tree is built, using a version of gcc specially modified to produce machine-readable abstract syntax tree dumps. Finally, the AST dump is fed into the analyze program, which is written in python, using standard python hashing functions.

The system works solely on C code, though it could probably be extended to C++, Fortran, and maybe Java fairly easily. I chose to target C because it is simpler than C++ or Java and a lot of very large programs are written in it.

4 Performance

For practical purposes, the performance of the system is poor. It takes on the order of ten hours to process 20 thousand lines of code on a 3.2GHz CPU. A significant part of the reason for this is python, which is known for poor performance. Another significant factor is that all data structures are built out of strings, to enhance transparency. Changing these factors could probably create a large speedup.

For theoretical purposes, the performance is excellent. The entire system is strictly linear in the number of functions. It is roughly quadratic in the size of the functions, as each subtree must be processed, and the size and number of the subtrees are both linear in the complexity of the function. Things are slightly worse for long sequences without control-flow. These should not be particularly important, because functions should not get extremely large – rarely over 100 lines. Extremely large programs tend to have large numbers of reasonably sized functions, so the system should scale well.

4.1 Parallelism

The system is also highly parallelizable. While the final comparison in the hash table must be applied to the entire program at once, most of the work is in generating the keys to accompany each node. This can be parallelized limited only by available resources and the size of the program with very little overhead.

5 Results

Performance limitations have prevented large scale testing of the system. When fed the platform-independent components of the Freeciv client (which was chosen purely for convenience), the system located only one large block of duplicate code: separate functions to calculate the best attacking units and best defending units within a city, and they differ in only the field accessed. This was clearly a case of copy-and-paste programming.

The system found smaller duplicates as well, usually also varying by a field. It was because of this that I made field unification a special primitive in the system.

In general, however, the system found very little duplicative code in Freeciv. Most likely this is because Freeciv is a fairly clean codebase, frequently re-examined and cleaned up.

6 Future Work

To test the system properly, its performance will need to be improved. This may be a matter of routine optimization, or (more likely) require re-writing the system in a more efficient language.

The readability of the output code is poor. In the cases I have examined, it is reasonably straightforward to correct the code by hand, but this may not always be the case. The ideal system would be interactive and graphical, presenting the potential code along with the code it was taken from in an organized manner.

Finally, a better heuristic for calculating the value of unifying sections of code is needed. At present, we use the size of the tree and the number of times it appears. At the very least, we should consider the number of arguments and the amount of modal code. It would be better to attempt an understanding of the meaning or at least the cognitive load of the code segment.