Aspects of Code Pattern Removal

Mikhail Prokharau

University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany
mikhail.prokharau@informatik.uni-stuttgart.de

Abstract

While many approaches for detecting undesirable or recurring code patterns have been presented over the years, there often remains the question how to properly interpret their results. This article presents a number of aspects of semi-automatic pattern elimination and discusses consequences of their application for safety and efficiency.

1 Introduction

The idea behind undesirable pattern detection is obviously to remove the detected patterns and, thus, to improve the overall quality of the analysed software. Often the removal itself has to be performed manually, in which case the user has to decide whether it is at all reasonable to eliminate a particular instance, how to do it best and, last but not least, how to perform this operation safely guaranteeing that the semantics of the original code remains intact. Another important criterion which is often overlooked, however, is how the change is going to affect code generation and program analyses performed on the software later on.

Many detection approaches divide the recurring pattern space into four types [2] of patterns named clones. The types are classified according to code similarity. The higher the pattern type, the higher the degree of modification that is required to perform the removal. Yet, as more changes are introduced, the higher the likelihood of incorrect or in some way detrimental modification becomes. To alleviate the problem, parts of the elimination process can be automated. To do so, the results of program analyses can be used that provide a number of important details fundamental to the decision whether the actual elimination should be performed and how to do it best. An advanced program analysis tool featuring suitable intermediate representation that preserves close links to the original code is most suitable for this purpose. The Bauhaus project [1] is a good example.

2 Semi-automatic Pattern Removal

While pattern detection is effectively and efficiently possible using higher-level code abstractions such as abstract syntax trees, the removal can profit significantly from the results of program analysis techniques applied to lower-level code abstractions. Specifically, a number of candidates definitely known to be false positive can be automatically excluded from having to be considered by the user. The exclusion is often performed by pattern detection tools, yet the results might be rather limited depending on the availability and precision of control and data-flow analysis data.

Some of the common code removal techniques include identifying subprograms which perform the same action and unifying them into a single subprogram. In some other cases it might be reasonable to unify certain cloned regions of code into a single subprogram and call it separately from within each context, possibly with varying parameters and case-statements to differentiate among the contexts. A number of non-local variables used in the fragment might become parameters of the new subprogram. Depending on their visibility some of the non-local variables might be accessed directly. Program analysis techniques can provide the user with detailed information regarding the specific choices.

If more than two instances of the same code passage have been detected and some yet not all of those have additional common properties as inferred by comprehensive program analyses, further subgrouping is a reasonable way of their presentation to the user. This is the case, for example, if subprogram pointers were identified to point to one specific subprogram in a subset of detected recurring patterns as a result of the must-point-to analysis. In fact, if this is the only difference, automatic substitution of the subprogram pointer with a direct call of the identified subprogram results in promoting the relevant code patterns from Type-2 to Type-1 clones.

3 To Remove or not to Remove?

Advanced programming constructs such as references, function pointers or dispatching calls are found in many contexts. These contexts probably present significant difficulty for the user who has to decide whether patterns using such constructs lead to the same execution in all contexts. In those situations the results of the previously conducted in-depth static program analyses are of particular value. For instance, if the program analyses had difficulty reducing the points-to-sets to a specific object, providing the

detailed analysis information to the user may facilitate her or his decision-making.

Unification of certain contexts where previously mentioned advanced programming constructs are present may also significantly affect subsequent code generation and analyses. For instance, in cases where the reduction of the number of objects being possibly pointed to by a pointer was previously attainable, it might no longer be so. This may have a number of potentially very negative consequences, such as lower efficiency of the resulting code due to the limited number of possible optimisations and the larger number of false positives delivered by the analyses depending on the pointer target sets. Where such effects are present, the user should be informed of the consequences of code pattern removal. The feedback may range from a generic warning to specific details regarding the expected complications.

4 Patterns in Specialised Contexts

Particular care should be taken when removing code patterns in specialised contexts.

Where parallel execution might be present, certain variables can be used for special tasks such as locks or synchronisation variables guaranteeing separate execution. An improper modification can result in serious and difficult-to-detect errors such as race conditions.

Introducing additional subprogram calls instead of cloned code may negatively affect the code efficiency. Unless inlining is used, a subprogram call normally requires a context switch taking longer execution time. While not likely to cause problems in linear parts of the code where only a small number of calls occur, this is of particular importance if loops, especially with a high degree of nesting, are involved.

In the context of real-time systems, where deadlines must be adhered to, pattern removal techniques that affect the resulting code efficiency quickly become unusable unless the exact quantifiable timing effect can be provided.

5 Practical Consequences

In the example shown in Figure 1 a code fragment is given where functions `f1` and `f2` might be recognised to be Type-3 clones by a clone detection analysis. In Figure 2 a possible result of pattern removal is shown. As can be observed, a function pointer was introduced to represent the code parts which are different. A similar strategy can be used in object-oriented languages where dispatching calls would substitute function pointers. In all of those cases the resulting context fusion poses a significant problem for points-to program analyses. Since pattern removal is performed in the synchronised context, the parallelisation analyses are also affected. The result might be a significant increase in the number of detected points-to targets and, consequently, false positive race condition notifications.

```
1  void f1() {
2      pthread_mutex_lock (&mutex1);
3      global++;
4      global *= 2;
5      pthread_mutex_unlock (&mutex1);
6  }
7
8  void f2() {
9      pthread_mutex_lock (&mutex2);
10     global++;
11     global *= 3;
12     pthread_mutex_unlock (&mutex2);
13 }
14
15 void c() {
16     pthread_mutex_lock (&mutexc);
17     f1();
18     f2();
19     pthread_mutex_unlock (&mutexc);
20 }
```

Figure 1: The original code fragment

```
1  void work1() {
2      global *= 2;
3  }
4
5  void work2() {
6      global *= 3;
7  }
8
9  void f(pthread_mutex_t* lock, void (*work)(void)) {
10     pthread_mutex_lock (lock);
11     global++;
12     work();
13     pthread_mutex_unlock (lock);
14 }
15
16 void c() {
17     pthread_mutex_lock (&mutexc);
18     p = work1;
19     f(&mutex1, p);
20     p = work2;
21     f(&mutex2, p);
22     pthread_mutex_unlock (&mutexc);
23 }
```

Figure 2: The code fragment after pattern removal

6 Conclusion

While removal of undesirable or recurring code patterns may prove very useful in improving a number of important software metrics, care should be taken not to introduce significant performance overheads while maintaining a high degree of the software analysibility. Semi-automation based on in-depth static program analyses can be quite valuable for reducing the otherwise unforseen negative effects.

References

[1] A. Raza, G. Vogel, and E. Plödereder. Bauhaus - a tool suite for program analysis and reverse engineering. In *Ada-Europe 2006*, volume 4006 of *LNCS*, pages 71–82. Springer, 2006.

[2] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.