A Model-Based Approach to Type-3 Clone Elimination

Torsten Görg

University of Stuttgart
Universitaetsstr. 38, 70569 Stuttgart, Germany

torsten.goerg@informatik.uni-stuttgart.de

Abstract

This paper presents an approach to elimination of Type-3 code clones based on generative techniques. At a first step it is shown how to replace Type-3 clones by higher-order functions. Then this approach is further generalized by utilizing generative designs.

1 Introduction

Code clones are a well known phenomenon that occurs in program source code. As defined in [3] usually four types of code clones are differentiated. This article focuses on Type-3 clones that do not crosscut syntactical boundaries. In contrast to Type-2 clones, Type-3 clones are characterized as copied code fragments with any modifications. Many clone detection mechanisms have been developed. The question that comes up after clones in a system are detected is what to do with them. Many authors believe that code clones impact maintainability negatively and it is a good idea to get rid of that redundancy. This can be achieved by code transformations. The following sections shows several techniques for clone elimination. It is presumed that one set of code fragments that are all similar to each other is provided as input.

2 Elimination of Type-2 Clones

Generally code clones can be eliminated by introducing a new abstraction or extending an existing abstraction. Several publications show how to implemented that for Type-2 clones, e.g., [2].

A further classification dimension for clones is their granularity related to the syntactical level they appear on. Syntactical levels can also be used as a criteron to classify clone elimination techniques. For clones at different snytactical levels different elimination techniques are applied. These are some syntactical levels: expression level, statement level/statement sequence level, subprogram (procedure/function) level, and type level/class level.

Many elimination techniques have been developed for the statement sequence level and the subprogram level. Common statement sequences can be extracted to a new procedure. Cloned subprograms can be merged. As clones live in different contexts, access to variables and subprograms that are referenced by the clone code has to be established through parameters that are added to the signature of the extracted or merged subprogram.

3 Elimination of Type-3 Clones

To eliminate Type-3 clones the techniques mentioned in the previous section can be extended. The subprogram extraction for the statement or subprogram level has to consider that the abstracted statement sequences are different. The extracted subprogram must behave differently depending on the calling context. This can be handled with additional subprogram parameters. The easiest way is to introduce an ID to discriminate different contexts. The execution of context-specific code is controlled by case switches. This approach has the disadvantage that context specifics are introduced into the extracted subprogram. It violates the separation of concerns principle.

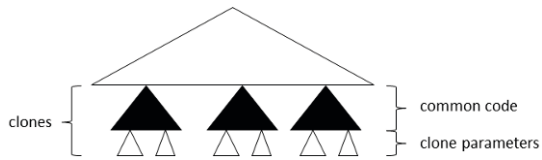

Figure 1: AST of code containing Type-3 clones

This problem can be solved with higher-order functions. The idea is to pass the code parts that are different in the clones as parameters. This can be viewed as a horizontal cut in the AST that represents the clone code (see Figure 1). The upper region above the cut is common for all clones. Code differences are expressed by subtrees that are below the cut. These subtrees can be written as lambda expressions or as separate functions and passed to the extracted subprogram via a function parameter. Of cource functional programming languages are predestinated for that technique, but in many imperative languages this is possible as well. Here is an C# example:

```
void extracted( Action<int, int> specific ) {
   int a = 5; int b = 3;
   Console.Write( "Result: " );
   specific( a, b );
}
...
extracted( (x, y) => { Console.Write(x); } );
extracted( (x, y) => { Console.Write(y); } );
```

In the example the parameter function `specific` is called with `a` and `b` as parameter values. This provides context information but, in contrast to the extracted subprogram that needs the outside context, here the inside context is required for the parametrization that is done outside.

4 Model-Based Techniques

Even more powerful are generative techniques. A generator produces some output based on templates. Obviously all instantialitions of the same template are clones. The generation is parameterized with model data. The resulting clones are Type-3 clones. E.g., UML tools generate source code skeletons from class models, the code for all classes looks similar as it is based on the same template. Here is another example with GUI code that creates some buttons:

```
w.add( new Button( "Yes", handlerYes ) );
w.add( new Button( "No", handlerNo ) );
w.add( new Button( "Cancel", handlerCancel ) );
```

If we reverse the generation process, it can be used for clone elimination. Given cloned code fragments are reduced to a domain-specific model. The model is free of redundancy and simplifies modifications during maintenance. But a model is not directly executable. For compilation, source code is required. In the build process source code corresponding to the model data is generated as described above. An appropriate generator has to be available. In fact, reversing the generation process does not really eliminate clones. The clones are just hidden. They are implicitly created by the generator. But the effect is similar as with clone elimination, as no manual modifications have to be done on the generated code. Just the model and the generator are modified manually.

To eliminate Type-3 clones this way, three artifacts have to be derived from the clones: a domain-specific model, a metamodel, and a corresponding generator.

(1) The model contains all differences between the clones. Similar to the parameters of an extracted subprogram, the model provides the contexts. For the GUI example above these are the extracted differences:

```
"Yes" handlerYes
"No" handlerNo
"Cancel" handlerCancel
```

(2) The data contained in the model has a structure that is specified by a metamodel. In Figure 2 it is shown for the example. The metamodel also has to be extracted from the clone code. The metamodel is important for the generator to know which template to apply to the model data.

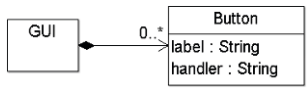

Figure 2: Metamodel for the GUI example

(3) The generator mainly consists of template definitions with placeholders for referenced parameters. Here is the Button template for the GUI example:

```
w.add( new Button( <<label>>, <<handler>> ) );
```

A template is created based on the clone code by replacing varying parts by placeholders for clone parameters.

So far the example does not go far beyond handling Type-2 clones. For a full support for Type-3 clones, the derived metamodel is combined with the metamodel of the underlying programming language. This is similar to the extension of extracted subprograms to higher-order functions, as shown in the previous section. Where delegates are used to express complex variations in higher-order functions here arbitraty ASTs can be embedded in the domain-specific model. Figure 3 provides a combined metamodel for

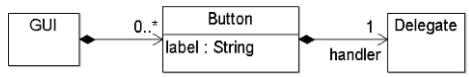

Figure 3: Combined metamodel for the GUI example

the GUI example. Instead of referencing handler functions by name, the extended metamodel enables to model handlers as arbitrary delegates.

5 Related Work

A realization of the higher-order function approach for Haskell code is described in [1].

6 Conclusion

Higher-order functions can be used to eliminate Type-3 clones at the statement sequence level. Generative techniques provide full flexibility to eliminate Type-3 clones that are arbitrarily parameterized at any syntactical level.

References

[1] C. Brown and S. Thompson. Clone Detection and Elimination for Haskell. In *PEPM '10 Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM, 2010.

[2] R. Fanta and V. Rajlich. Removing clones from the code. *Journal of Software Maintenance*, pages 223–243, 1999.

[3] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009.