# Chapter 12

# Clone Detection and Removal for Erlang/OTP within a Refactoring Environment

Huiqing Li[1], Simon Thompson[2]
*Category: Research*

***Abstract:*** A well-known bad code smell in refactoring and software maintenance is duplicated code, or code clones. A code clone is a code fragment that is identical or similar to another. Unjustified code clones increase code size, make maintenance and comprehension more difficult, and also indicate design problems such as lack of encapsulation or abstraction.
This paper proposes a token and AST based hybrid approach to automatically detecting code clones in Erlang/OTP programs, underlying a collection of refactorings to support user-controlled automatic clone removal, and examines their application in substantial case studies. Both the clone detector and the refactorings are integrated within *Wrangler*, the refactoring tool developed at Kent for Erlang/OTP.

### 12.1 INTRODUCTION

Duplicated code, or the existence of code clones, is one of the well-known bad code smells when refactoring and software maintenance is concerned. 'Duplicated code', in general, refers to a program fragment that is identical or similar to another, though the exact meaning of 'similar' might vary slightly between different application contexts.

While some code clones might have a sound reason for their existence [9], most clones are considered harmful to the quality of software, as code duplication

[1]Computing Laboratory, University of Kent, UK; `H.Li@kent.ac.uk`
[2]Computing Laboratory, University of Kent, UK; `S.J.Thompson@kent.ac.uk`

increases the probability of bug propagation, the size of both the source code and the executable, compile time, and more importantly the maintenance cost [18, 15].

Software clones appear for a variety of reasons, among which the most obvious is the reuse of existing code (by copy and paste for example), logic or design. Duplicated code introduced for this reason often indicates program design problems like the lack of encapsulation or abstraction. This kind of design problem can be corrected by refactoring out the existing clones in a later stage [4, 14, 6], it could also be avoided by first refactoring the existing code to make it more reusable, then reuse it without duplicating the code [14]. In the last decade, substantial research effort has been put into the detection and removal of clones from software systems; however, few such tools are available for functional programs, and there is a particular lack of tools that are integrated with existing programming environments.

Erlang/OTP [1] is an industrial strength functional programming environment with built-in support for concurrency, communication, distribution, and fault-tolerance. This paper investigates the application of clone detection and removal techniques to Erlang/OTP programs within the refactoring context, proposes a new hybrid approach to automatically detecting code clones across multiple modules, and describes three basic refactorings which together help to remove code clones under the user's control. Both the clone detector and the refactorings have been implemented within Wrangler [11], the refactoring tool developed at Kent for Erlang/OTP. Wrangler is integrated with both Emacs and Eclipse, and works with the full Erlang/OTP language.

The proposed clone detection approach is able to report code fragments in an Erlang program that are syntactically identical after semantic preserving renaming of variables, except for variations in literals, layout and comments. Syntactically, each of these code clones is a sequence of well-formed expressions or functions. This approach makes use of both token suffix tree and abstract syntax tree (AST) annotated with location and static semantic information. The use of token suffix tree allows us to detect clone candidates quickly, whereas the use of AST ensures that the tool only reports syntactically well-formed clone candidates. Furthermore, static semantic information annotated in the AST is used to check to whether two code fragments can be refactored to each other by consistent renaming of variables and literals, and thus to ensure that the clones detected are actually removable.

Three refactorings have been designed and implemented to support the clone removal process, and they are *generalise a function definition*, *function extraction*, and *fold expressions against a function definition*. Unlike fully automated removal of clones, which usually does not give perfect solutions all the times, these refactorings respect the importance of user intervention during the clone removal process, and allow clones to be removed step by step under the programmer's control. Apart from removing code clones from legacy programs, these refactorings are also for programmers to use as part of their daily programming activities to avoid the introduction of code clones from the very beginning.

The remaining of the paper is organised as follows. In section 2, we clarify

the definition of related terms, and give an overview of existing and related work. Section 3 presents the approach taken by the Wrangler clone detector; Section 4 discusses the three major refactorings developed for duplicated code removal purpose; The usefulness of the tool is demonstrated in section 5, and conclusions and future work are given in section 6.

## 12.2 RELATED WORK

The phrase 'code clones' in general refers to program fragments that are *identical* or *similar* to each other. Two code fragments can be similar if their program texts are similar or their functionalities are similar without being textually similar. Since semantic similarity is generally not decidable, in the research reported here we only consider textually identical or similar code fragments, which can be compared on the basis of their program text or internal representation, such as parse trees or ASTs. In this paper, we distinguish the following four types of clones. All these four types of clones ignore variations in literals, layout and comments.

- *Type 1*: Identical code fragments.
- *Type 2*: Code fragments that are identical after *consistent* (i.e. semantic-preserving) renaming of variable names.
- *Type 3*: Code fragments that are identical after renaming *all* variable names to the same name.
- *Type 4*: Code fragments that are identical only after renaming *all* function names and variable names to the same name respectively.

Obviously, these four types of clones satisfy a *subset* relation, i.e. clones of *Type* $i_{(i=1,2,3)}$ form a subset of clones of *Type (i+1)*. Among the four types of clones, *Type 1* and *Type 2* represent the clones that are most suitable for automatic clone removal because of the semantic equivalence between cloned code fragments, and they are also the kinds of clones that are reported by the Wrangler clone detector. *Type 3* and *Type 4* clones are not suitable for mechanical removal, but they some reveal structure-level duplication, and are obtainable from the intermediate results of the Wrangler clone detector.

A typical clone detection process first transforms source code into an internal representation which allows the use of a comparison algorithm, then carries out the comparison and finds out the matches. A recent survey on existing techniques is given by Roy and Cordy in [18], an overview of which is given now.

**Text-based approaches** consider the target program as sequence of lines/strings. Two code fragments, possibly after some pre-processing, are compared with each other to find sequences of same text/strings. The comparison techniques used may vary from each other. For example, suffix-tree based matching is used by Baker in [2], whereas fingerprint-based string comparison is used by Johnson in [18].

**Token-based approaches** first lex/transform the program to a sequence of tokens, then apply comparison techniques to find duplicated subsequences of tokens. Representative techniques include *CCFinder* [16], a language-independent

clone detector that reports clones of *Type 3*; *Dup* [3], which uses the notion of parameterised matching by a consistent renaming of identifiers; and *CP-Miner* [13], which uses a frequent subsequence mining technique to indentify a similar sequence of tokenized statements. Both *CCFinder* and *Dup* use suffix-tree based token matching techniques. Like text-based approaches, token-based approaches are in general efficient, but can report syntactically non well-formed clones. While *Dup* does consistent-renaming checking of variables, without knowing the scoping rules of the target language, false positives are not avoidable.

**AST-based approaches** search for similar subtrees in the AST with some tree matching techniques. Since naïve comparison of subtrees for equality does not scale, Baxter et al.'s *CloneDR* [5] partitions the sets of comparisons by categorizing sub-trees with hash values. The use of hash function enables one to do consistent renaming checking (therefore to detect clones of *Type 2*), to detect near-miss clones such as clones involving commutative operators with the operands swapped. In [5], Baxter et al. also suggest to remove code clones with the help of macros, but they did not carry out clone removal. *DECKARD* [7] is another AST-based language independent clone detection tool, whose main algorithm is to compute certain *characteristic vectors* to approximate structural information within ASTs and then cluster similar vectors, and thus code clones. In [17], Koschke et al. propose to use suffix tree representation of AST to detect clones, and point out that their tool could have a better precision if consistent renaming were checked.

There are also clone detection approaches based on the program dependency graph, as demonstrated in [16]. Most of the above mentioned clone detection tools target large legacy programs, and none of them is closely integrated with an existing programming environment. Without applying deeper knowledge of the scoping rules of the target programming language, language-independent clone detection tools tend to have a lower precision, and are not very suitable for mechanical clone refactoring.

Our aim is to develop a clone detection and removal tool for a specific language, Erlang/OTP in this case. The clone detection tool should be able to handle large Erlang programs, to report as many clones as possible but without giving false positives; instead of fully automating the clone refactoring process, which is not ideal in most cases, the clone removal tool should give the user more control on which clone to remove and how to remove it. Being part of a programming environment, the tool is more accessible to working programmers, and has a better chance to be used in practice.

## 12.3 THE WRANGLER CLONE DETECTOR

Common terminology for the clone relations between two or more code fragments are the phrases *clone pair* and *clone class* [8]. A clone pair is a pair of code fragments which are identical or similar to each other; a *clone class* is the maximal set of code fragments in which any two of the code fragments form a clone pair.

With Wrangler, clones are reported in the form of *clone classes* by giving the

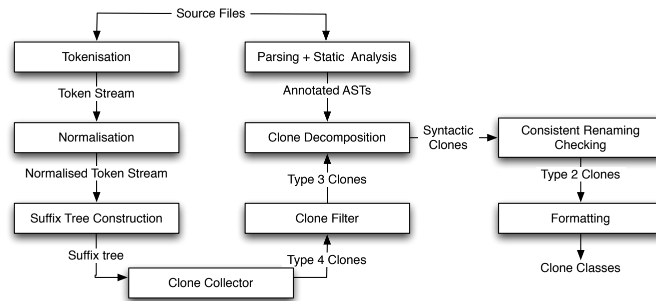


**FIGURE 12.1. An Overview of the Wrangler Clone Detection Process**

number of clones included in a clone class, and each member clone's start and end locations in the program source. Two threshold values can be given by the user to specify the granularity of the clone classes reported by the clone detector, and they are:

- the minimum number of tokens that a code clone should have, and
- the minimum number of members of a clone class.

Figure 12.1 gives an overview of the Wrangler clone detection process. First, the target program is tokenised into a token stream with location information attached to each token. The generated token stream is then normalised with all the atom identifiers, variables and literals being replaced by a special symbol respectively. After that a suffix tree is built on the transformed token stream, and the initial clone classes are collected from the suffix tree. At this stage, the collected clone classes are clones of *Type 4*. The collected *Type 4* clone classes are further processed to filter out those clone classes which are not of *Type 3* by token-level comparison of function name tokens. In order to decompose non-syntactic clones into syntactic units, and check for consistent renaming of variables names, annotated ASTs of the related Erlang modules are built, and the token representation of remaining *Type 3* clones are mapped to their AST representation. Final clone classes of *Type 2* are reported after the decomposition and consistent renaming checking processes.

### 12.3.1 Token-level Clone Detection

As the first step of the clone detection process, the target program is transformed into a token stream, in which each token is associated with its location information including the name of the source file, line and column numbers. White spaces between tokens and comments are not included in the token stream. In the case that the target program contains more than one Erlang file, tokens of all these files are concatenated into a single token stream. The generated token stream is further

```
-> case lists:subtract(SLocs1, ELocs2) of
     []-> R1 = lists:filter(fun({S, E}) ->
                              lists:member(E, SLocs1) end, Range),
          R2= lists:map(fun({S, E}) -> S end, R1),
          {lists:zip(R2, ELocs1), Len1+Len2, F1};
     _ ->
```

(a)

```
-> case lists:subtract(ELocs1, SLocs2) of
     [] -> R3= lists:filter(fun({S,E}) ->
                              lists:member(S, ELocs1) end, Range),
           R4 = lists:map(fun({S,E}) -> E end, R3),
           {lists:zip(SLocs1, R4), Len1+Len2, F1};
     _ ->
```

(b)

**FIGURE 12.2. A clone pair found in Wrangler**

processed by normalising all atom identifiers, variables, and literals to a special symbol respectively, but keywords and operators are left untouched. Before suffix tree construction, the whole token stream is mapped into a string over a fixed sized alphabet by mapping each token into a character from the alphabet. Tokens with the same value are mapped to the same character.

Suffix tree analysis [19] is the technique used by most token-based clone detection approaches because of its speed [3, 8]. A suffix tree is a representation of a string as a trie where every suffix is represented through a path from the root to a leaf. The edges are labelled with the substrings, and paths with common prefixes share an edge. A clone can be indentified in the suffix tree as an inner node. The length of the clone is the number of characters from the root to this inner node, and the number of occurrences of the clone is the number of leaves that can be reached from this inner node. To save space, instead of labelling edges in the suffix tree with actual substrings, we label each edge with the substring's location in the whole string. As there is a one-to-one mapping between the tokens in the token stream and the characters in the string, the location of a substring also indicates the location of the corresponding token sequence in the token stream. Once the suffix tree has been constructed, it is traversed and clone classes are collected. The clone classes generated at this stage are of *Type 4* because of the normalisation of function and variables names before the suffix tree construction. The clone classes generated are then processed to take function names into account, during which an original clone class could be decomposed into smaller clone classes (in

terms of the size of the code fragments or the number of clone members), or be discarded, because of the differences in function names. After this step, only clone classes that are of *Type 3* are kept.

While the code clones reported at this stage provide useful code duplication information about the target program, some of these clones might be spurious, or not very interesting from the code removal perspective. For example, Figure 12.2 shows a clone class of two code fragments, which are actually from the same function, found by the above process from the source code of Wrangler. A couple of problems immediately show from this example:

- The code fragments do not form syntactic units. While in general there could be one or more meaningful syntactic units embedded in a non-syntactic code fragments, in some extreme cases, it is possible that, after removing those incomplete parts, a code clone does not contain any interesting syntactic units at all, or the contained syntactic units are under the specified thresholds, therefore should not be reported to the user. Decomposing code fragments into meaningful syntactic units needs certain syntax information, which is naturally available from ASTs but not so obvious from the token stream.
- While the two code fragments do look similar, semantically they are different. Applying consistent variable renaming to the two code fragments does not produce the same code, therefore it is not an interesting candidate from the clone removal perspective. This problem is due to the normalisation process applied to the token stream, in which all variables names are treated as the same. Consistent renaming of variable names at token stream level is in general complex for programming languages that allow nested scopes.

We make use of ASTs annotated with location and static semantic information to decompose non-syntactic clones into syntactic ones, and filter out those clones that do not unify after consistent renaming of variables/literals.

### 12.3.2 Annotated Abstract Syntax Trees

Like most refactoring tools, Wrangler uses AST as the internal representation of the program under refactoring. Both program analysis and transformation manipulate the AST directly. To facilitate the refactoring process, we annotate the generated ASTs with further syntax, semantic, and location information, therefore comes the term *annotated abstract syntax tree* (AAST). Some of the annotation information was naturally used by the clone detector to post-process the clone classes reported from the previous step. This information includes:

- Location information. Each AST node is annotated with the start and end location of the program entity that it represents in the source code. Location information was originally added for interface purposes, but it also makes it straightforward to map the token sequence representation of a code fragment to its AST representation, or vice versa (recalling that each token in the token stream is also associated with its location in the program source).

- Binding structure information. The binding structure describes the association between the uses of an identifier and its definition. In Wrangler, this information is incorporated in the AST through the defining and occurrence locations of an identifier. For example, each occurrence of a variable node in the AST is annotated with the location of its occurrence in the source code as well as the location where it is defined. Two occurrences of the same identifier name refer to the same thing if and only if they have the same defining location. With this kind of binding information, we can easily check whether a code clone fragment can be transformed to another code fragment in the same clone class by applying consistent variable renaming.

### 12.3.3 Decomposing into Syntactic Clones

The previous token-based step produces a set of clone classes each of which contains a maximal set of code clones of *Type 3*. However these code fragments may not form complete syntactic units as shown in Figure 12.2. In this step, we decompose these code fragments into sub portions, each of which forms a syntactic unit. Within the context of Erlang programs, we say that a clone is a *syntactic clone*, or forms a syntactic unit, if it consists of an sequence of expressions separated by a comma, or a sequence of functions separated by a full stop.

To process a clone class, we first choose a code clone from the class, and construct the AAST of the module to which the code clone belongs, then traverse the generated AAST in a top-down left-to-right order collecting those nodes whose start and end locations in the program source fall into the range of the code clone, and whose syntax type is expression or function. Once a node has been collected, its arguments are not to be traversed. These collected nodes are then put into groups, each group containing a maximal consecutive sequence of expressions/functions. Because all the code fragments in a clone class have identical syntactical structure, only one fragment's AAST is needed for the decomposition purpose; once this fragment has been decomposed, the decomposition of the others can be done at token-level by projecting the new code portions to the token sequence, and removing those unwanted tokens.

Returning to the example in Figure 12.2, this clone class will be decomposed into two classes, one containing the guard expression of the case expression, and the other containing the sequence of three expressions of the first case clause. However because the code clones in the first class contains less than 30 tokens (the default threshold value), this clone class will be removed from the result. Therefore after this step, the original clone class become the one shown in Figure 12.3. Here we assume that the two remaining code fragments are not members of other existing clone classes.

### 12.3.4 Checking For Consistent Renaming

Checking for consistent renaming of identifiers is another important aspect of a clone detector. With the Wrangler clone detector, we mainly check the consis-

```
R1 = lists:filter(fun({S, E}) ->
       lists:member(E, SLocs1) end, Range),
R2= lists:map(fun({S, E}) -> S end, R1),
{lists:zip(R2, ELocs1), Len1+Len2, F1};
```

(a)

```
R3= lists:filter(fun({S,E}) ->
      lists:member(S, ELocs1) end, Range),
R4 = lists:map(fun({S,E}) -> E end, R3),
{lists:zip(SLocs1, R4), Len1+Len2, F1};
```

(b)

**FIGURE 12.3. The clone pair after decomposition**

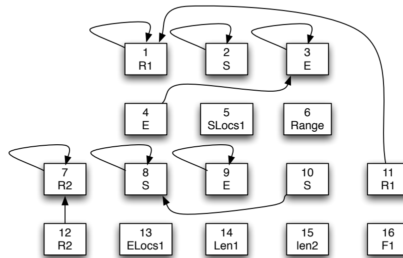


**FIGURE 12.4. The binding graph of clone (a)**

tent renaming of bound variables, i.e., those variables that are locally declared in the code fragment under consideration. The checking for consistent renaming is done by comparing the binding structure of the clone members of a clone class. This is feasible only because all the code fragments in a clone class are structurally/syntactically identical. Given a code fragment, we can treat each variable occurrence in it as a node in a graph; we explain the construction of this in more detail now.

Instead of using variable names as the node names, we replace each variable occurrence with a number according to their textual occurrence order in the code. For example, the first variable occurrence is numbered as 1, the second is numbered as 2, and so on. If a node represents a use occurrence of a bound variable, then there is an edge drawn from this node to the node representing the defining occurrence of this variable (recalling that, in the AAST, each variable occurrence

is associated with its defining location). In the case that the node itself represents a defining occurrence of a variable, an edge is drawn to itself. No edges are associated with nodes that representing occurrences of free variables, i.e., variables that are used, but not declared, in the code fragment. This means that we treat each free variable occurrence as a different entity, even though some occurrences might share the same variable name.

In this way, we are able to represent the binding structure of a code fragment as a graph, and two structurally/syntactically identical code fragments can be transformed to each other by consistent variable renaming only if their binding structure graphs are the same. As an example, Figure 12.4 shows the binding structure graph of the code fragment (a) shown in Figure 12.3, in which we annotated the variable name to each node for clarity.

Returning to the previous clone example shown in Figure 12.3, it is obvious that these two code fragments have different binding structure graphs, therefore this clone class will be removed from the finial result too.

It is possible, in this step, that a clone class is partitioned into two or more small clone classes according to their binding structure equivalence. Again, clone classes under the specified thresholds are discarded. At this point, all the reported clones are syntactic clones of *Type 2*.

## 12.4 REFACTORING SUPPORT FOR CLONE REMOVAL

Duplicated code often indicates lack of encapsulation and reuse; therefore a primary purpose of clone detection is to remove them from the system via refactoring to improve the system's quality. As code clones will generally be scattered throughout the program, removing clones manually can be tedious and error prone. To support the clone removal process, we have developed a set of refactorings which together can help to remove clones efficiently and reliably.

Another scenario for the use of these refactorings is to refactor the existing code and then to reuse it, thus avoiding the introducing of clones from the beginning. There are other refactorings in Wrangler, such as *renaming*, *moving a function definition between modules*, etc, which were not designed especially for duplicated code elimination purpose, but still can help in some cases.

Instead of fully automating the clone removal process, we give the user more control as to which clone instance to remove and how to remove. Furthermore, the *undo* feature of Wrangler can always be used to recover a removed clone if the user changes his/her mind.

Next, we introduce the three main refactorings we have developed for clone removal purpose, and some examples are given where it is necessary.

### 12.4.1 Function Extraction

*Function extraction* is the first step towards clone removal. This refactoring encapsulates a sequence of expressions into a new function. To perform this refactoring with Wrangler, the user highlights in the editor a sequence of expressions,

```
start_loc(Node, Toks)->
  case refac_syntax:type(Node) of
    if_expr -> Cs = if_expr_clauses(Node),
                {S, E} = get_range(hd(Cs)),
                extend_forwards(Toks,S, 'if');
    cond_exp -> Cs = cond_expr_clauses(Node),
               {S, E} =get_range(hd(Cs)),
               extend_forwards(Toks,S, 'cond');
   end.
```

```
start_loc(Node, Toks) ->
  case type(Node) of
    if_expr -> Cs = if_expr_clauses(Node),
               newfun(Cs, Toks);
    cond_exp ->Cs = cond_expr_clauses(Node),
               {S, E} = get_range(hd(Cs)),
               extend_forwards(Toks, S, 'cond');
  end.
newfun(Cs, Toks) ->
  {S, E} = get_range(hd(Cs)),
  extend_forwards(Toks, S, 'if').
```

**FIGURE 12.5. The 'function extraction' refactoring**

and inputs the new function name when prompted. Wrangler checks whether the selected expression sequence can be extracted, and whether the new function name causes conflicts within current module. If all the checking succeeds, a new function will be created automatically with the selected expression sequence as its function body, and free variables of the expression sequence as its formal parameters. The selected expression sequence is then replaced by a function call, or a match expression with the function call as its right-hand side if the expression sequence exports values. The newly created function is put right after the enclosing function of the selected expression sequence. In Figure 12.5, the boldfaced text in the first code fragment represents the code for extraction, and the result of this refactoring is shown in the second fragment.

### 12.4.2 Generalisation of Function Definition

*Generalisation of a function definition* makes the function more reusable. This is especially useful when the clone members of a clone class have variations in literals. With Wrangler, to generalise a function over an expression in its function body, the user only needs to highlight the expression from the source, and provide

```
start_loc(Node, Toks) ->
  case type(Node) of
    if_expr -> Cs = if_expr_clauses(Node),
               newfun('if', Cs, Toks);
    cond_exp ->Cs = cond_expr_clauses(Node),
               {S, E} = get_range(hd(Cs)),
               extend_forwards(Toks, S, 'cond');
  end.
newfun(Keyword, Cs, Toks) ->
  {S, E} = get_range(hd(Cs)),
  extend_forwards(Toks, S, Keyword).
```

**FIGURE 12.6. The program after generalisation**

a new parameter name when prompted. If the pre-condition checking succeeds, Wrangler will generalise the function by adding the new parameter to the function's definition, replacing the selected expression with the new parameter, and making the selected expression the actual value of the new parameter at the call sites of this function. In the case that the selected expression has side-effects or free variables, it would be wrapped in a function expression before being supplied to the call sites of the function. Figure 12.6 shows the program after generalising function `newfun` on the literal expression `'if'`.

### 12.4.3 Folding against a Function Definition

*Folding against a function definition* is the refactoring which actually removes code clones from the program. This refactoring searches the program for instances of the right-hand side of a function clause, and replaces them with applications of the function to actual parameters under the user's control. This refactoring can detect not only instances where parameters are replaced by variables or literals, but also instances where parameters are replaced by arbitrary expressions. Therefore the instances detected by the refactoring could be a superset of the clone instances reported by the clone detector.

To apply this refactoring to a program, the user only needs to select the function clause against which to fold by pointing the cursor to it (or input the function clause information if it is not defined in the current module), and the refactoring command from the menu. Wrangler automatically searches for code fragments that are clones of the selected function clause's body expression. Once clone instances have been found, the user can decide whether to fold all the clone instance without any further interaction with Wrangler, or to go through the instances one by one, and instruct Wrangler whether a particular instance should be replaced or not. Note that the folding is not performed within the selected function clause

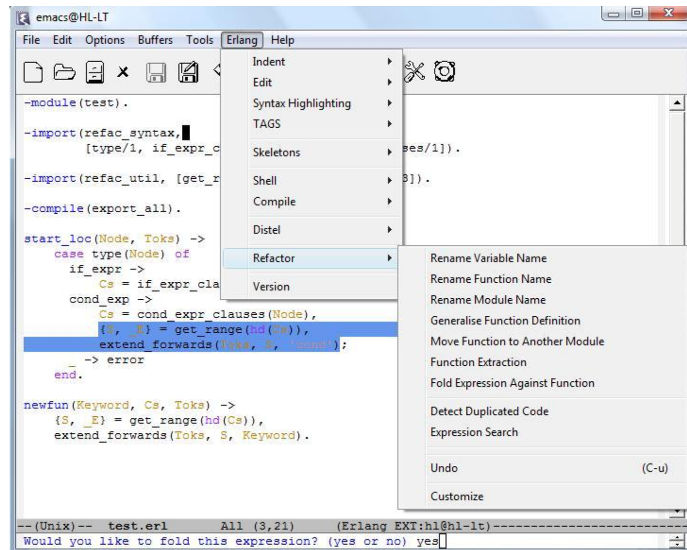


**FIGURE 12.7. A snapshot of Wrangler showing folding**

itself, since doing this will change the program's semantics.

The snapshot in Figure 12.7 shows the scenario of folding against the function `newfun`: the user has selected the function `newfun`, selected the *Folding Expression against Function* command from the *refactor* menu, and have decided to go through the candidates one by one. The snapshot shows that the Wrangler highlights the first candidate instance, and asks whether the user wants to replace this expression sequence with the application of `newfun`. If the users types *yes* within the minibuffer, the highlighted expression will be replaced by `newfun('cond', Cs, Toks)`, otherwise the highlighted expression will remain unchanged. In either case, Wrangler will move to the next candidate instance if there is any, or finish the process if no more candidate instances left.

## 12.5 A CASE STUDY

As a case study, we have applied the tool to the source code of Wrangler itself. Wrangler is implemented in Erlang; its current version contains 24 Erlang modules, 20K lines of code in total. Due to the compactness of programs written in functional programming languages, this is by no means a small program.

In this case study, we customised the clone detector to report both the final result and the intermediate results after each step, and we used the default threshold setting, which is 30 for the minimum number of tokens in a clone, and 2 for the minimum number of members in a clone class. Within a few minutes, the result shows that 1210 clone classes of *Type 4* were detected based on the information

in the suffix tree. After putting the original function names back to the tokens, this number dramatically reduced to 191. These are clones of *Type 3*, but some of them are not syntactically complete. After the *decomposition* step, 86 clone classes remained, all of which are syntactic clones of *Type 3*; and the final *consistent renaming checking* step further reduced the previous result to 36 clone classes of *Type 2*.

Further examination shows that 12 out of the final 36 clone classes are across multiple modules, and among which 3 clone classes are function duplications resulting from a 'copy and paste' operation. The copy and paste happened when a module, A say, wants to use a function defined in another module, B say, but for some reason the author did not want module B to export this function. These clones have been removed from Wrangler by relocating the related function definitions to a library module.

In theory, all the detected expression clones, except for one which is an expression list within a list comprehension expression, are removable from the code. But in practice, we have refactored out some of them, and left the others unchanged. The main reason that we left them untouched was because the code fragments were so simple that we did not feel that it is necessary to encapsulate them into functions. This kind of clones normally involve applications of functions with a large number of parameters, and would be filtered out from the final result if we increase the threshold value of the minimum number of tokens in a clone.

With the same threshold setting, we have also applied the tool a third party codebase which contains 89 Erlang files, 32K lines of code, and 109 clones of *Type 2* were reported.

## 12.6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a hybrid clone detection technique which makes use of both the token stream and the AST to improve performance and efficiency, and a collection of 3 refactorings which together help to remove clones from code under the user's control.

The Wrangler clone detector benefits from both the speed of token-based clone detection approaches and the accuracy achieved by AST-based approaches. The usefulness and ease of use of the 3 refactorings were also demonstrated via examples. Both the clone detector and the refactorings are part of the Erlang refactoring tool Wrangler, which is embedded in Emacs and Eclipse. Integrating Wrangler within the program development environment allows it to be used in the normal development process, and in the spirit of this paper to remove any clone as soon as it appears.

In the future, we would like to improve the tool in two directions. First we would like to make use of visualisation techniques to improve the presentation of the clone results; second, we would like to develop more refactorings to better support the elimination of function clones.

While the presented tool is especially for Erlang/OTP programs, the idea is not limited to this single language. In fact, we would like to apply the technique

to Haskell programs, to add duplicated code detection and elimination support to HaRe [12, 10], the tool we have developed for refactoring Haskell programs. Since Haskell is a statically typed language, we can foresee that type information needs to be taken into account when clone removal is concerned.

## REFERENCES


[1] J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

[2] B. S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[3] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, Los Alamitos, California, 1995.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial Redesign of Java Software Systems Based on Clone Analysis. In *Working Conference on Reverse Engineering*, pages 326–336, 1999.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *ICSM '98*, Washington, DC, USA, 1998.

[6] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring Support Environment Based on Code Clone Analysis. In *IASTED Conf. on Software Engineering and Applications*, pages 222–229, 2004.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Computer Society Trans. Software Engineering*, 28(7):654–670, 2002.

[9] C. Kapser and M. W. Godfrey. "Clones Considered Harmful" Considered Harmful. In *Proc. Working Conf. Reverse Engineering (WCRE)*, 2006.

[10] H. Li and S. Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation*, San Francisco, California, USA, 2008.

[11] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy. Refactoring Erlang/OTP Programs. In *EUC'06*, Stockholm, Sweden, November 2006.

[12] H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer, HaRe, and its API. *Electr. Notes Theor. Comput. Sci.*, 141(4):29–34, 2005.

[13] Z. Li, S. Lu, and S. Myagmar. CP-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *METRICS '02*, Washington, DC, USA, 2002.

[16] R. Komondoor and S. Horwitz. Tool Demonstration: Finding Duplicated Code Using Program Dependences. *Lecture Notes in Computer Science*, 2028:383–386, 2001.

[17] R. Koschke and R. Falke and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *WCRE '06*, pages 253–262, Washington, DC, USA, 2006.

[18] C. H. Roy and R. Cordy. A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University at Kingston, Candada, 2007.

[19] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.