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

Huiqing Li
Computing Laboratory, University of Kent, UK
H.Li@kent.ac.uk

Simon Thompson
Computing Laboratory, University of Kent, UK
S.J.Thompson@kent.ac.uk

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.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.6 []: Programming Environments; D.2.7 []: Distribution, Maintenance, and Enhancement; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 []: Processors

General Terms Languages, Design

Keywords Erlang, refactoring, Wrangler, duplicated code, program analysis, program transformation.

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 (Kapser and Godfrey 2006), most clones are considered harmful to the quality of software, as code duplication increases the probability of bug propagation, the size of both the source code and the executable, compile time, and more importantly the maintenance cost (Roy and Cordy 2007; Monden et al. 2002).

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

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 such as the lack of encapsulation or abstraction. This kind of design problem can be corrected by refactoring out the existing clones in a later stage (Balazinska et al. 1999; M. Fowler 1999; Higo et al. 2004), it could also be avoided by first refactoring the existing code to make it more reusable, then reuse it without duplicating the code (M. Fowler 1999). 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 (Armstrong 2007) 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 (Li et al. 2006a, 2008), the refactoring tool developed at Kent for Erlang/OTP.

Wrangler is a tool that supports interactive refactoring of Erlang/OTP programs. It is integrated with Emacs and now also with Eclipse. Wrangler itself is implemented in Erlang, apart from the implementation of a suffix tree construction algorithm, which is written in C. The current version of Wrangler supports more than a dozen refactorings, as well as functionalities for code inspection to find 'bad smells' of various kinds

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, also allowing 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 a token suffix tree and abstract syntax tree (AST) annotated with location and static semantic information. The use of the token suffix tree allows us to detect clone candidates quickly, whereas use of the AST ensures that the tool only reports syntactically well-formed clone candidates. Furthermore, static semantic information annotated in the AST is used to check 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 from Wrangler 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, 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 remainder of the paper is organised as follows. An overview of the Erlang/OTP language is given in section 2. Section 3 presents the approach taken by the Wrangler clone detector, as well as introducing terminology used in the remainder of the paper. Section 4 discusses the three major refactorings developed for use in duplicated code removal scenarios. The usefulness of the tool is demonstrated in section 5; section 6 gives an overview of related work; finally, conclusions and future work are given in section 7.

2. The Erlang/OTP Language

Erlang is a strict, dynamically typed, functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance, and dynamic code reloading (Armstrong 2007).

One of the fundamental properties of the Erlang language is built-in support for light-weight processes, each with its own memory, and the use of explicit message passing for inter-process communication. With the support for SMP (Symmetric MultiProcessing) being added to the recent releases of the Erlang Virtual Machine (VM), applications implemented in Erlang as a number of cooperating Erlang processes can take advantage of multi-core technology without being modified.

Erlang's elementary data types are atoms, numbers (integers and floats), process identifiers, references, binaries, and ports; compound data types are tuples, records and lists. Types in Erlang are checked at run time, and compound data structures will be heterogeneous; some compile-time type checking is provided by tools such as Dialyzer (Sagonas 2005).

An Erlang program typically consists of a number of modules, each of which defines a collection of functions. Only functions exported explicitly may be called from other modules, and a module can only export functions which are defined in the module itself.

Figure 1 shows an Erlang module containing the definition of the factorial function. In this example, `fac/1` denotes the function `fac` with arity of 1. In Erlang, a function name can be defined with different arities, and the same function name at different arities will generally represent entirely different computations.

```
-module (fac).
-export ([fac/1]).
fac(0) -> 1;
fac(N) when N > 0 -> N * fac(N-1).
```

Figure 1. Factorial in Erlang

Erlang allows static scoping of variables, in other words, matching a variable to its binding only requires analysis of the program text. However, some variable scoping rules in Erlang are rather different from other functional programming languages, such as Haskell (S. Peyton Jones 2003). For instance,

- In Erlang, the binding occurrence of a variable always occurs in a pattern, but a pattern may also contain applied (i.e. non-binding) occurrences of variables.
- Non-linear patterns with multiple occurrences of the same variable are allowed.
- A variable may have more than one binding occurrence, due to the fact that branches in a `case` or `receive` expression in Erlang can export variables.

Knowledge of these scoping rules is essential to the correct resolution of the binding structure of variables in an Erlang program. The binding structure of variables is used by Wrangler's clone detection process, as will be discussed in section 3.6, as well as in a number of refactorings implemented in Wrangler; see section 4.

The Erlang language itself is small, but it comes with libraries containing a large set of built-in functions. Erlang has also been extended by the Open Telecom Platform (OTP) middleware platform, which provides a number of ready-to-use components and design patterns, such as finite state machines, generic servers, etc, embodying a set of design principles for fault-tolerant robust Erlang systems

3. The Wrangler Clone Detector

The phrase 'code clone' in general refers to a set of 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.

Our aim is to develop a clone detection and removal tool for a specific language, Erlang/OTP in this case. Instead of starting from scratch, we make use of the infrastructure established for Wrangler; this also allows a natural integration of the tool into Wrangler. The clone detection tool should be able to handle large Erlang programs, to report as many clones as possible but without giving false positives.

Being part of the programming environments of choice for Erlang programmers, the tool is more accessible to working programmers, and so it has a better chance to be used in practice.

It has been our experience in building refactoring tools that what might initially be seen as a single refactoring operation in fact has a number of variants. For instance, in generalising a function over a sub-expression, should the generalisation be over a single occurrence of the sub-expression, all such occurrences or a user-defined selection of them: different answers apply in different situations. A similar phenomenon occurs in clone detection, and so instead of fully automating the clone refactoring process, the clone removal tool gives the user more control on which clone to remove and how to remove. This is done by providing a number of basic refactorings which can together be used to accomplish clone removal. These are discussed in more detail in section 4.

3.1 Terminology

Common terminology for the clone relations between two or more code fragments are the phrases *clone pair* and *clone class* (Kamiya et al. 2002). 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.

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 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 somehow reveal structure-level duplication, and are obtainable from the intermediate results of the Wrangler clone detector.

3.2 Clone detection architecture

With Wrangler, clones are reported in the form of *clone classes* by giving the 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 the reported clones should contain, and
- the minimum number of members of a clone classes reported.

Figure 2 gives an overview of the Wrangler clone detection process. First, the target program is tokenised into a token stream. The generated token stream is then normalised with all the atom identifiers, variables and literals being replaced by a special symbol representing each kind of token. After that a suffix tree is built on the transformed token stream, and the initial clone classes, which are of *Type 4*, are collected from the suffix tree. 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 tokens representing a function name. In order to decompose non-syntactic clones into syntactic units, and check for consistent renaming of variables, annotated ASTs of the related Erlang modules are built, and the token representation of remaining *Type 3* clones are mapped to their AST representation. The final clone classes of *Type 2*, ordered by size, are reported after the decomposition and consistent renaming checking processes.

3.3 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 and the line and column numbers of each occurrence. 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 processed by normalising all atom identifiers, variables, and literals as described earlier, but keywords and operators are left untouched. Before suffix tree construction, the whole token stream is mapped into a string over a fix-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 (Ukkonen 1995) is the technique used by most token-based clone detection approaches because of its speed (Baker 1995; Kamiya et al. 2002). A suffix tree is a representation of a string as a tree 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.

Figure 3 shows the well known suffix tree example – the suffix tree representation of *MISSISSIPPI* padded with a special terminating character '$'. A clone can be identified in the suffix tree as an inner node. An inner node is a node in the tree that has more than one child node, but is not the root 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 leaf nodes that can be reached from this inner node. Instead of labelling edges in the suffix tree with actual substrings, we label each edge with the list of locations, in terms of start and end index, where the substring appears 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.

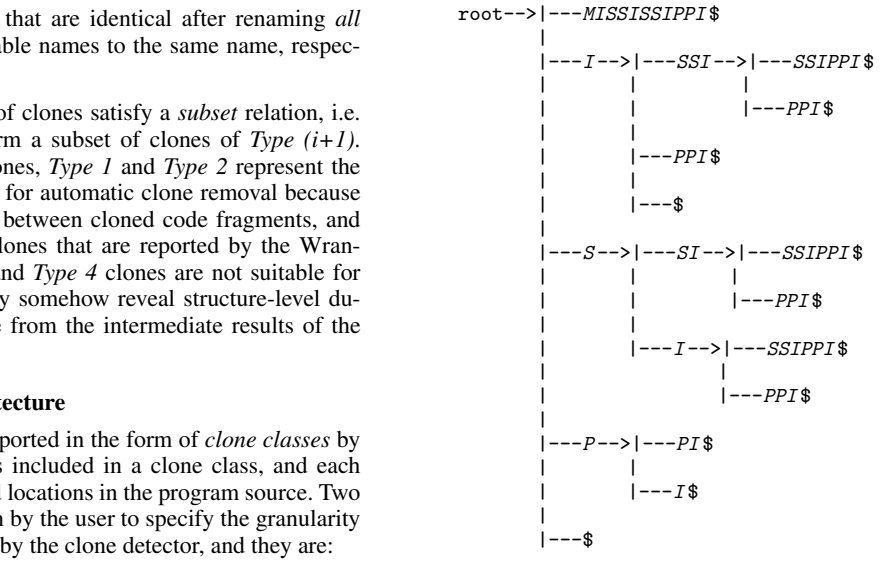

Figure 3. Suffix Tree for *MISSISSIPPI*

Once the suffix tree has been constructed, it is traversed and clone classes are collected. While each inner node represents a set of cloned code fragments, only those nodes representing a maximal set of cloned code fragments, i.e. a clone class, are included in the final result. Suppose we represent a set of cloned code fragments as a tuple $\{S, N\}$, where S is the cloned code fragment, and N is the number of times S is cloned, then $\{S, N\}$ is maximal if and only if there is not another set of cloned code fragments, $\{S1, N1\}$ say, such that S is a substring of $S1$ and N is less than $N1$. Back to the *MISSISSIPPI* example, given a threshold value of 1 for the minimum number of characters that a clone should have, and 2 for the minimum number of members of a clone class, the suffix-tree clone collector will report the following cloned substrings: *I*, *ISSI*, *S*, and *P*.

For efficiency reason, we did not implement the construction of suffix tree in Erlang, instead we made use of an open source ANSI C implementation (Tsadok 2002) of E.Ukkonen's suffix tree construction algorithm (Ukkonen 1995), which has $O(nlog|\sum|)$ time and $O(n)$ space complexity (where n is the length of the source string, $\sum$ is the alphabet and $|\sum|$ is the size of the alphabet.

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 process 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 difference in function names. After this step, only clone classes that are of *Type 3* are kept.

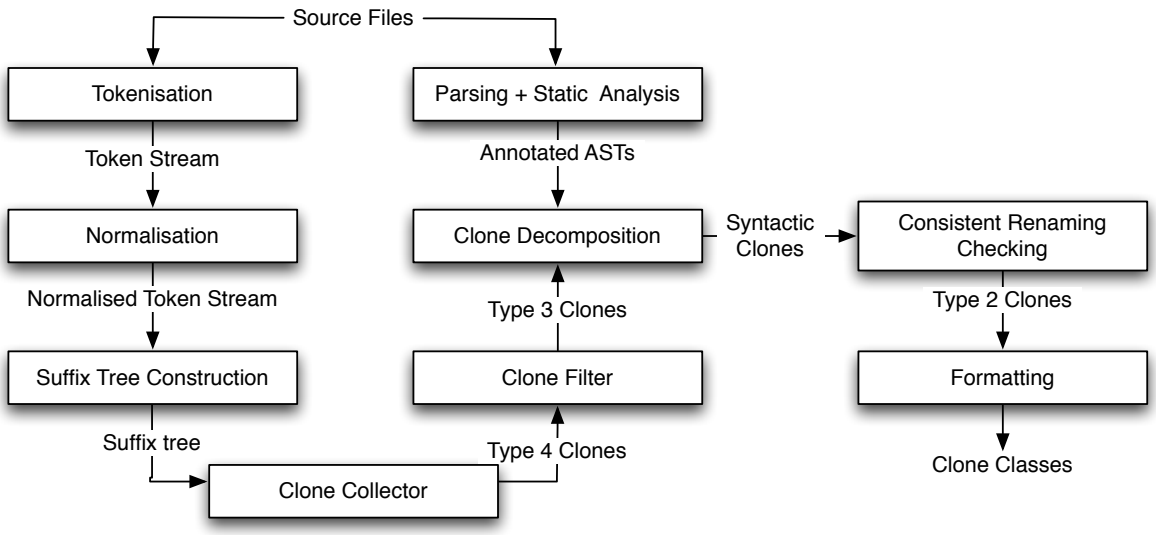

Figure 2. An Overview of the Wrangler Clone Detection Process

```
-> 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 4. A clone pair found in Wrangler

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 4 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. A syntactically not well-formed code fragment cannot be removed as a whole by means of function extraction. While in general there could be one or more meaningful syntactic units embedded in a non-syntactic code fragment, 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, and 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 is 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 the same name to be declared across 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.

3.4 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. These 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. 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 program 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.

3.5 Decomposing into Syntactic Clones

The previous token-based step produces a set of clone classes of *Type 3*. However code fragments reported by these clone classes may not form complete syntactic units as shown in Figure 4. 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 a 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 4, 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, since the guard expression only has 8 tokens, it is very likely that the first clone class is below the threshold value specified, and discarded during the process. Therefore after this step the original clone class could become the one shown in Figure 5. Here we assume that the two remaining code fragments are not members of other existing clone classes.

3.6 Checking For Consistent Renaming

Checking for consistent renaming of identifiers is another important aspect of a clone detector. A related study (R. Koschke and R. Falke and P. Frenzel 2006) has shown that clone detectors that check for consistent renaming have better precision than those that do not. From the clone removal perspective, clone classes whose clone members can be transformed to each other by consistent renaming of variables and literals are good candidates for automatic clone removal.

With the Wrangler clone detector, we check the consistent 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.

```
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 5. The clone pair after decomposition

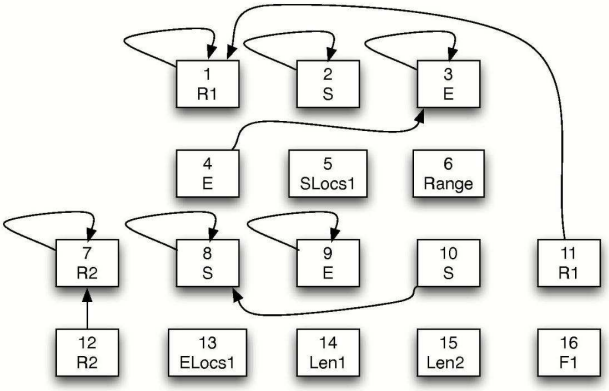

Figure 6. The binding graph of clone (a)

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 6 shows the binding structure graph of the code fragment (a) shown in Figure 5, in which we annotated the variable name to each node for clarity.

Returning to the previous clone example shown in Figure 5, 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.

```
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 7. The 'function extraction' refactoring

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*.

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.

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, 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 7, the italised text in the first code fragment represents the code for extraction, and the result of this refactoring is shown in the second fragment.

```
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 8. The program after generalisation

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 a new parameter name when prompted. If the side-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 8 shows the program after generalising function `newfun` on the literal expression `'if'`.

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 found 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

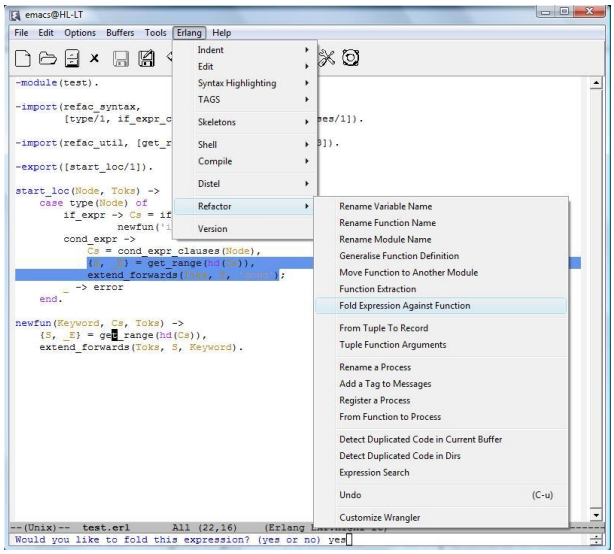

Figure 9. A snapshot of Wrangler showing folding

	Wrangler	Mensia	Yaws
No. of files	44	38	68
Program size(K LOC)	30.9	28.2	26.9
Time(Min)	< 6	< 3	< 3
Type 4 clones	311	133	181
Type 3 clones before syntactic decomposition	56	123	178
Type 3 clones after syntactic decomposition	72	53	92
Type 2 clones	52	43	66
Inter-module *Type 2* clones	35	5	18

Table 1. Clone Detection Results

No. of class members	30-40 (tokens)	40-50 (tokens)	50-70 (tokens)	70-100 (tokens)	>100 (tokens)
2	17	1	7	3	18
3	1				
4	1				
5			1		
>5	2	1			

Table 2. Clone Distribution of Wrangler

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

The snapshot in Figure 9 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 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.

5. Experiments

The section presents a detailed evaluation of Wrangler's clone detection and elimination support by applying it to real Erlang applications.

5.1 Experiment Setup

To evaluate the tool, we have applied it to several applications written in Erlang including Wrangler itself (30,872 LOC), Mnesia (28,152 LOC) (Mattsson et al. 1998) which is a distributed database management system from the Erlang/OTP release, and Yaws (29,603 LOC) (Yaw) which is a web server. Due to the compactness of program written in functional programming languages, these applications are by no means small Erlang programs. All experiments were run on a laptop with Intel(R) 2.00 GHz processor, 2015MB RAM, and Windows Vista installed.

With these experiments, we aimed to evaluate the tool in terms of the following: running time, the number of clones reported, the percentage of clones that are inter-module and number of clones refactored out using Wrangler's clone elimination support. In order to illustrate the number of false positives eliminated during the clone detection process, we customised the clone detector to report both the final result and the intermediate results after each step. As to the threshold settings, we used the default values, i.e., 30 for the minimum number of tokens in a clone, and 2 for the minimum number of members in a clone class.

5.2 Experiment Results

Table 1 shows the result of applying Wrangler's clone detector to the three Erlang applications mentioned above. For all three applications, Wranlger were able to finish the clone detection process in reasonable time. The running time is not only affected by the size of the program, but also by the number of initial candidates collected from the suffix tree. In this table, we distinguish the number of *Type 3* clones before and after the syntactic decomposition step. More often, the number of clones classes will be reduced after syntactic decomposition, but it is also possible for the number to increase because a clone class with large code fragments could be decomposed into several clone classes with smaller code fragments. These experiments also demonstrated the benefit of *consistent renaming checking*, hence the knowledge of the scoping rules of the target language; without consistent renaming checking, the accuracy of the tool would be deteriorated significantly.

Table 1 also shows that Wrangler has a very high percentage of inter-modules clones. This is due to the fact that Wrangler keeps two versions of token scanner, one in the module `refac_scan.erl` and the other in `refac_scan_with_layout.erl`. The latter keeps white spaces and comments in the token stream, and is a modified version of `refac_scan.erl`. Initially just for experimental purpose, we copied the `refac_scan` module, and made some necessary modifications. Therefore these two modules share a lot common functions, which should be refactored out into a separate module. Unsurprisingly, 25 clone classes of the inter-module clones reported are related to these two modules.

Table 2-4 illustrates the distribution of clones from each application in terms of the size of cloned code fragments, i.e., the number of tokens, and the number of times the cloned code fragment appear in the code. Menesia apparaently has the least amount of duplicated code.

5.3 Clone Elimination

To evaluate Wrangler's support for clone elimination by means of refactoring, we underwent the process of removing the clones found in Wrangler using Wrangler itself.

No. of class members	30-40 (tokens)	40-50 (tokens)	50-70 (tokens)	70-100 (tokens)	>100 (tokens)
2	20	8	7	4	1
3	1	2			

Table 3. Clone Distribution of Mnesia

No. of class members	30-40 (tokens)	40-50 (tokens)	50-70 (tokens)	70-100 (tokens)	>100 (tokens)
2	22	12	5	7	7
3	1	2	1		1
4	3	3	1		
>5		1			

Table 4. Clone Distribution of Yaws

The first step is to remove those inter-module function clones related to `refac_scan` and `refac_scan_with_layout`. This is achieved by moving these duplicated functions to a newly created module, `refac_scan_lib`, by applying the refactoring *move function to another module* from Wrangler. With this step 19 duplicated function definitions were removed.

The refactoring *move function to another module* moves a function definition from its current module to a module specified by the user, and changes all the references to this function across the program accordingly. As a side-condition, the function to be moved should not cause any conflicts in the target module. Originally, this refactoring would fail if the same function name is already defined in the target module, regardless whether the two function definitions are syntactically, therefore semantically, the same or not; during this clone removal process, we modified the implementation of this refactoring so that the refactoring process will continue if syntactically the same function is already defined in the target module, that is, Wrangler will remove the function definition from its original module without adding it to the target module, and will change all references to this function consistently.

Using the combination of *function extraction*, *generalisation* and *folding*, we managed to remove another 20 clone classes reported. From our experience, these three refactorings are all very convenient to use, except that the user needs to figure out which parts (i.e., literals) of the new function introduced by *function extraction* need to be generalised before applying the *folding* refactoring to this function.

The remaining clones were left unchanged because the duplicated code fragments only contain a single function application with a large number of parameters. For example, the following expression:

```
scan(Cs, Stack, [{'->', {Line, Col}} | Toks],
      {Line, Col + 2}, State, Errors);
```

was reported with 17 duplications, but it is not necessary to encapsulate it again.

6. Related Work

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 of existing techniques is given by Roy and Cordy in (Roy and Cordy 2007), an overview of which is given now.

6.1 Text-based approaches

Text-based approaches consider the target program as sequence of lines/strings. Two code fragments, possibly after some preprocessing, 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 (Baker 1992); fingerprint-based string comparison is used by Johnson in (Roy and Cordy 2007); whereas Ducasse et al. (Ducasse et al. 1999) use string-based Dynamic Pattern Matching (DPM) to textually compare whole lines that have been normalised to ignore whitespace and comments.

Text-based approaches can be sensitive to minor changes made in the copy-pasted code, and for such approaches it is hard to guarantee that the reported clones form well-formed syntactic units. Checking of consistent renaming of variables at a textual level is also a challenge.

6.2 Token-based approaches

Token-based approaches first perform lexical analysis on the program to produce a sequence of tokens, then apply comparison techniques to find duplicated subsequences of tokens. Representative techniques include *CCFinder* (R. Komondoor and S. Horwitz 2001), a language-independent clone detector that reports clones of *Type 3*; *Dup* (Baker 1995), which uses the notion of parameterised matching by a consistent renaming of identifiers; and *CP-Miner* (Li et al. 2006b), 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 impossible to avoid.

6.3 AST-based approaches

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* (Baxter et al. 1998) partitions the sets of comparisons by categorizing sub-trees with hash values. The use of hashing enables consistent renaming checking to be performed, and therefore for clones of *Type 2*) to be detected. It also supports the detection of near-miss clones such as clones involving commutative operators with the operands swapped.

In (Baxter et al. 1998), Baxter et al. also suggest a mechanism for the removal of code clones with the help of macros, but they did not carry out clone removal. *DECKARD* (Jiang et al. 2007) 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 (R. Koschke and R. Falke and P. Frenzel 2006), 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.

6.4 Using the program dependency graph

There are also clone detection approaches based on the program dependency graph, as demonstrated in (R. Komondoor and S. Horwitz 2001). Using data mining techniques, Basit et at. (Basit and Jarzabek 2005) have gone one step further to infer design-level similarities based on patterns of clones. 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.

The hybrid approach that we have described earlier in the paper can be seen to combine the speed of a token-based approach with the accuracy of the AST-based approach, through using the former to identify candidate clones, whose more detailed analysis can be performed with reference to the annotated AST.

7. 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 three directions. First we would like to make use of visualisation techniques to improve the presentation of the clone results; Second, we would also like to provide functionalities for scripting refactoring commands, basing on clone results, to support users who prefer fully automated clone removal; Third, we would like to develop functionalities for detecting code fragments that are similar but not clones in the spirit of this paper. A particularly fruitful direction here is the refactoring possibilities for test code, to be investigated in the ProTest project (ProTest).

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 (Li et al. 2005; Li and Thompson 2008), 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.

The idea of hybrid clone detection can be applied to programs from any paradigm; it would be a very useful project to assess the effectiveness of hybrid techniques in other programming domains.

Acknowledgement

The authors would like to thank the UK Engineering and Physical Sciences Research Council for its support for Wrangler (project EP/C524969/1).

References

J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

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

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.

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.

H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes*, 30(5):156–165, 2005. ISSN 0163-5948.

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.

S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings ICSM99 (International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.

M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

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.

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. IEEE Computer Society.

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.

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

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

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.

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

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

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

H. Li, S. Thompson, G. Orosz, and M. T"oth. Refactoring with Wrangler, updated. In *ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada*, September 2008.

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, 2006b. ISSN 0098-5589. Member-Yuanyuan Zhou.

H. Mattsson, H. Nilsson, and C. Wikstrom. Mnesia - a distributed robust dbms for telecommunications applications. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 152–163, London, UK, 1998. Springer-Verlag.

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.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. ISBN 0-521-82614-4.

ProTest: property-based testing. `http://www.protest-project.eu`.

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

K. Sagonas. Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects In Erlang Applications. Presented at the *ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

D. Tsadok. ANSI C implementation of a Suffix Tree. Technical report, Computer-Science Department, Haifa University, Israel, August 2002.

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

Yaws – An Open Source Web Server Written in Erlang. `http://yaws.hyber.org/`.