# Similar Code Detection and Elimination for Erlang Programs

Huiqing Li and Simon Thompson

School of Computing, University of Kent, UK
`{H.Li, S.J.Thompson}@kent.ac.uk`

**Abstract.** A well-known bad code smell in refactoring and software maintenance is duplicated code, that is the existence of *code clones*, which are code fragments that are identical or similar to one another. Unjustified code clones increase code size, make maintenance and comprehension more difficult, and also indicate design problems such as a lack of encapsulation or abstraction.
This paper describes an approach to detecting 'similar' code based on the notion of *anti-unification*, or least-general common abstraction. This mechanism is used for detecting code clones in Erlang programs, and is supplemented by a collection of refactorings to support user-controlled automatic clone removal. The similar code detection algorithm and refactorings are integrated within Wrangler, a tool developed at the University of Kent for interactive refactoring of Erlang programs. We conclude with a report on case studies and comparisons with other tools.

**Key words:** Anti-unification, Code clone detection, Erlang, Program analysis, Program transformation, Refactoring, Similar code, Wrangler.

## 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. The term 'duplicated code', in general, refers to program fragments that are identical or similar to one another; the exact meaning of 'similar code' might be substantially different between different application contexts.

While some code clones might have a sound reason for their existence [1], most clones are considered harmful to the quality of software, since code duplication increases the probability of bug propagation, the size of the source and executable, and most importantly the cost of maintenance [2,3].

The most obvious reason for code duplication is the reuse of existing code, typically by a sequence of *copy*, *paste* and *modify* actions. Duplicated code introduced in this way often indicates program design problems such as a lack of encapsulation or abstraction. This kind of design problem can be corrected by refactoring out the existing clones at a later stage [4,5,6], but it could also be avoided by first refactoring the existing code to make it more reusable, and then reusing it without duplicating the code [5].

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 programming languages, and there is a particular lack of tools that are integrated with existing programming environments, thus supporting clone removal as a part of the programmer's normal work pattern.

This paper describes an approach to detecting 'similar code' in Erlang programs based on the notion of *anti-unification* [7,8], as well as a mechanism for automatic clone elimination under the user's control. The anti-unifier of two terms denotes their *least-general common abstraction*, therefore captures the common syntactic structure of the two terms.

In general, we say two expressions or expression sequences, `A` and `B`, are *similar* if there exists a non-trivial least-general common abstraction, `C`, and two substitutions $\sigma_A$ and $\sigma_B$ which take `C` to `A` and `B` respectively. By 'non-trivial' we mean that the size of the least-general common abstraction should be above some threshold, but certain other conditions can be specified, and this is under active investigation.

The approach presented in this paper is able, for example, to spot that the two expressions `((X+3)+4)` and `(4+(5-(3*X)))` are similar as they are both instances of the expression `(Y+Z)`, and so both instances of the function

```
add(Y,Z) -> Y+Z.
```

Our approach uses as the representation of an Erlang program the Abstract Syntax Tree (AST) for the parsed program annotated with static semantic information. Scalability, one of the major challenges faced by AST-based clone detection approaches, is achieved by a two-phase clone detection technique. The first phase uses a more efficient syntactic technique to identify candidates which might be clones, which are then assessed by means of an AST-based analysis to give only genuine clones. While the paper shows this approach being implemented for Erlang in particular, we see no reason why it should not be applicable to similar code detection in any other programming language.

The application of the approach of this paper to a substantial case study is discussed in [9]; the account here concentrates on the underling theory and implementation of the technology.

The remainder of the paper is organised as follows. Section 2 gives an overview of Erlang and Wrangler, and in particular our earlier mechanism for clone detection and elimination, while clarifying the motivation and goal of this paper. Section 3 introduces some terminology to be used; Section 4 describes the similar code detection algorithm. The elimination of code clones is discussed in Section 5, and initial experimental results are reported in Section 6. Section 7 gives an overview of related work, and finally, Section 8 concludes the paper and briefly discusses future work.

## 2 Erlang and Wrangler

**Erlang** [10,11] is a strict, impure, dynamically typed functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance, and dynamic code loading. Unlike other functional programming languages such as Haskell [12], Erlang does not have built-in support for type classes, inheritance or polymorphism. 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.

The Erlang language 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.

**Wrangler** [13,14] is a tool that supports interactive refactoring of Erlang programs. It is integrated with Emacs as well as with Eclipse, through the ErlIDE plugin. Wrangler itself is implemented in Erlang. Wrangler supports a variety of refactorings, as well as a set of 'code smell' inspection functionalities, and facilities to detect and eliminate code clones. Wrangler supports a number of basic structural refactorings such as *renaming*, *function generalisation*, *function extraction*, *folding*, *move a function definition to another module*, *tuple function arguments*, etc, as well as a sets of macro- and process-related refactorings. Significant effort has been put to improve usability of the tool, and Wrangler is aimed to be used by real-world Erlang programmers from beginners to experts.

A clone detection and elimination framework was first added to Wrangler in 2007 [15]. In contrast to the approach proposed here, Wrangler's original clone detector reports syntactically well-formed code fragments that are *identical* up to consistent renaming of variables and substitution of literals. A hybrid clone detection technique which makes use of both the token stream and the AST was used to achieve performance and efficiency. Three refactorings, *function extraction*, *function generalisation* and *folding*, can together be used to remove clones from the program. More about this approach can be found in [15].

Wrangler's original clone detection mechanism is rather limited:

– The clone detector cannot detect code fragments that are similar but not identical, such as `X+Y` and `X+(Y+1)`.
– The user needs to figure out which of the literals contained in a cloned code fragment need to be generalised in order to capture the commonality of all duplications.
– Moreover, the user needs to identify which of variables locally declared in the cloned code fragment are used by the code following it, so that their values can be returned by the generalised function.
– To get these two sorts of information identified above, a manual inspection and comparison of *every* clone occurrence is needed, an impractical proposition in a system of any size.

To overcome these limitations, we have designed a new approach which can detect not only identical code but also code fragments that are similar through anti-unification. The clone elimination process has been greatly simplified so that the user no longer needs to work out the common abstraction and the set of variables to be returned, as these are identified automatically by the tool. With the new approach, we aim to spot more code clones, and make the clone removal process practically applicable.

## 3 Terminology

### 3.1 Anti-unfication

The idea of anti-unification was first proposed by Plotkin [7] and Reynolds [8] in 1970. Anti-unification applies the process of *generalisation* on pairs, or sets, of terms. The resulting term captures all the commonalities of the input terms.

A *substitution* is a mapping from variables to terms, and is in general represented as a set of bindings $\{x_1 \mapsto e_1, ..., x_n \mapsto E_n\}$. Applying a substitution $\sigma$ to a term $E = E(x_1, ..., x_n)$ gives the term $E\sigma = E(e_1, ..., e_n)$ in which each variable $x_i$ is replaced by the corresponding term $e_i$.

Given terms $E_1 ... E_n$, we say that $E$ is a *generalisation* of $E_1, ..., E_n$ if there exist substitutions $\sigma_i$ for each $E_i, 1 \leq i \leq n$, such that $E_i = E\sigma_i$. $E$ is the *least-general* common generalisation of $E_1 ... E_n$ if for each $E'$ which is also a common generalisation of $E_1, ..., E_n$, there exists a substitution $\theta$ such that $E = E'\theta$. The least-general common generalisation of $E_1, ..., E_n$ is called the *anti-unifier* of $E_1, ..., E_n$, and the process of finding the anti-unifier is called *anti-unification*.

To apply anti-unification techniques to ASTs of Erlang programs, restrictions as to which kinds of subtrees can be replaced by a variable, and which cannot, need to be taken into account. For instance, objects of certain syntactic categories, such as operators, guard expressions, record names, cannot be abstracted and passed in as the values of function parameters, and therefore should not be replaced by a variable during anti-unification. Furthermore, an AST subtree which exports some of its locally declared variables should not be replaced by a variable either. On the other hand, it is perfectly fine to substitute the function name in a function application with a variable because higher order functions are supported by Erlang.

### 3.2 Similarity Score

Anti-unification provides a concrete way of measuring the structural similarity between terms by showing how both terms can be made equal. In order to measure the similarity between terms in a quantitative way, we defined the *similarity score* between terms.

Let $E$ be the anti-unifier of sub-trees $E_1$, ..., $E_n$, the similarity score of $E_1$, ..., $E_n$ is computed by the following formula:

$$\textbf{Similarity Score} = \min\{S_E/S_{E_1}, ..., S_E/S_{E_n}\}$$

where $S_E$, $S_{E_1}$ ... $S_{E_n}$ represent the number of nodes in $E$, $E_1$... $E_n$ respectively. The similarity score allows the user to specify how similar two sub-trees should be to be considered as clones. Given a similarity score as the threshold, we say that a set of sub-trees are *similar* if their similarity score is above the threshold.

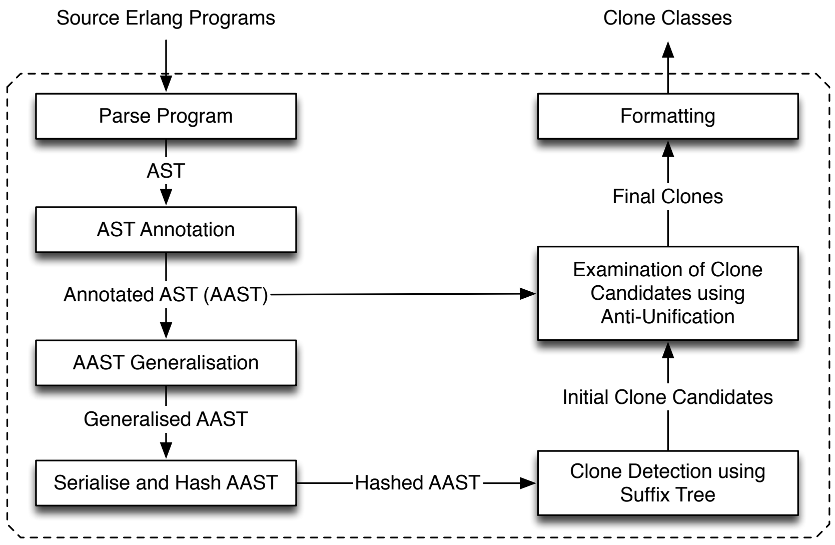


Fig. 1: An Overview of the Clone Detection Process

### 3.3 Definition of Clones

Common terminology for clone relations between two or more code fragments are the phrases *clone pair* and *clone class* [16]. A *clone pair* is a pair of code fragments which are identical or similar to each other. A *clone class* is a set of code fragments in which any two of the code fragments form a clone pair.

In the context of this paper, each member of a clone pair/class is a sequence of Erlang expressions. Note that sub-sequences of expression sequences in each clone pair/class could also make clone pairs/classes. Suppose we have a clone class with 3 class members: $\{[a_1, a_2, ..., a_n], [b_1, b_2, ..., b_n], [c_1, c_2, ..., c_n]\}$, then

$$\{[a_i, ...a_j], [b_i, , ...b_j], [c_i, ..., c_j]\}_{(1=<i=<j=<n))}$$

could also be clone classes. For ease of description, we use $C_{i,j}$ to represent the clone class whose class member are formed by the sub-sequence, starting from index $i$ and ending at index $j$, of each class member of clone class $C$.

While only those maximal clone classes whose similarity score is above the threshold specified are reported to the user, sub-sequence clone classes are used by the clone detection process; further details of this are given in Section 4.

## 4 The Similar Code Detection Algorithm

The similar code detector takes a project (or just a set of Erlang modules) as input, performs clone detection, and reports clone classes in the project. Each clone class is reported by giving the number of instances of the cloned code, each instance's start and end locations in the program source, as well as the least-general common generalisation represented as an Erlang function definition. The entire clone detection process is shown in Fig. 1. The process consists of seven steps as described in the rest of this section.

Three parameters can be used to specify the granularity of clone classes reported, and they are:

– the minimum number of expressions included in a cloned code fragment, which is a sequence of expressions;
– the minimum number of class members of a clone class, and
– the similarity score threshold.

**Parse Program and Generate AST** Erlang files are first lexed and parsed into ASTs. The lexer and parser used are modified versions of the standard Erlang lexer and parser, so that both line and column numbers of identifiers are kept in the AST. Location information makes it possible to map between different representations of the same piece of code. In order to reflect the original program text, the Erlang pre-processor is bypassed to avoid macro expansion, file inclusion, conditional compilation, etc.

**Annotate AST with Static Semantic Information** Binding information of variables and function names is annotated to the AST in terms of defining and use locations. Unlike some other AST representation approaches which use a single leaf node to represent all the occurrences of the same variable, the AST representation used by Wrangler does not allow node-sharing between different occurrences of the same variable. In this case, we use location information to express the binding structure of identifiers. For instance, each occurrence of a variable or function name in the AST is annotated with its occurrence location in the source and the location(s) where it is defined. Binding information allows us to check whether two variable or function names refer to the same object by looking at their defining locations; this is required during the anti-unification process.

Being static-semantics-aware, our clone detection tool is able to achieve the degree of accuracy that cannot be achieved by language-independent clone detection tools, or indeed tools that rely on the lexical structure alone.

**Generalise and Hash the AST** A major challenge faced by AST-based clone detection approaches is scalability. Naïve anti-unification of every subtree with every other subtree involves a prohibitively large amount of computation and memory usage, and is not feasible in practice. Scalability is achieved by our approach using a two-phase clone detection. The first phase carries out a quick, semantics-unaware clone detection over a generalised version of the program, and reports initial clone candidates to be further examined by the second phase. This second phase examines the initial clone candidates in the context of the original program by means of anti-unification, getting rid of false positives, and reports the final clone classes.

The first phase makes use of suffix tree techniques to collect initial candidates. Suffix tree analysis [17] is the technique used by most text or token-based clone detection approaches because of its speed [18,16]. A suffix tree is a representation of a string as a tree where every suffix is represented by a path from the root to a leaf. The edges are labelled with the substrings, and paths with common prefixes share an edge. The suffix tree analysis itself is only able to report duplications of strings that are identical. To make use of the suffix tree techniques, while being able to report similar code fragments, the AST needs to be pre-processed before being passed on for suffix tree construction. The pre-processing is carried out in two steps. Firstly, the AST is generalised so that only a structural skeleton of each expression statement is kept; secondly, a hash function is applied to each expression statement to map it to a number. This is covered next.

The aim of structural generalisation is to capture as much structural similarity between expressions as possible while keeping each expression's original structural skeleton. This process traverses each expression statement subtree in a top-down order, and replace certain kinds of subtrees with a single node representing a placeholder. A subtree is replaced by a placeholder only if syntactically it is legal to replace that subtree with a node representing a variable, and the subtree does not represent a pattern, a match expression or a compound expression such as a conditional expression, a `receive` expression, a `try...catch` expression, etc.

Taking the following code as an example, the generalisation process will turn the function definition on the left-hand side into the pseudo function definition on the right-hand side. As a design decision, our clone detector does not attempt to detect similar patterns simply because generalisation of a function over patterns could make the function much harder to understand in practice. Therefore in this example, the literal pattern `one` is not changed.

```
   foo(X) ->                         foo(X) ->
    Y = case X of                     ? = case ? of
          one    -> 12;                     one    -> ?;
          Others -> 196                     ?      -> ?
        end,                                end,
    X + Y.                            ?.
(a) original code                 (b) generalised code
```

Expression sequences in the AST are then pretty-printed and serialised into a single sequence of expressions with a delimiter to separate each sub expression sequence. After that, a hash function is applied to each expression statement in the sequence returning a hash value. Expression statements that are textually the same get the same hash value. All hash values are stored in an indexed table without duplication. This way, we are able to map a sequence of expressions into a sequence of numbers. To save space and make the algorithm more efficient, the actual implementation represents an expression using its start and end locations in the program source, and a hash value using its index in the table as an integer is much short than the hash value itself. The mapping is represented as a list of two-element tuples, whose first elements are locations and second elements are index values.

**Initial Clone Detection using a Suffix Tree** This step fetches the index values from each tuple in the list returned from the previous step, and concentrates them into a single string; a delimiter character is inserted after every index value during the concatenation. A suffix tree is then built on the string generated, and clone classes of index sequences are collected from the suffix tree. Location information is used to map clone classes in terms of indexes back to clone classes in terms of expression sequences. The suffix tree algorithm used is part of Wrangler's original clone detection algorithm, the implementation of which is reported in [15].

**Examine Clone Candidates using Anti-unification** The previous step returns a collection of clone classes whose class members are structurally similar, but which do not necessary share a non-trivial anti-unifier; even so it helps to reduce the amount of comparisons needed significantly. This step examines the initial clone class candidates one by one using anti-unification and removes those false positives. It takes one clone class as input each time, and returns none, one or more clone classes that satisfy the thresholds. Together with each final clone class, the anti-unifier of the class members is returned. Due to space restrictions, the anti-unification algorithm is not discussed in this paper.

For each clone class candidate, $C$ say, the clone detector takes a class member, $A$ say, as the first member of a new clone class, $C_1$ say, and try pairwise anti-unification with each of the other class members. A class member from $C$ is added to $C_1$ only if doing so does not make the similarity score of $C_1$ go under the threshold specified. When no more new members can be added to $C_1$, the clone detector checks whether the number of clone members in $C_1$ is above the parameter specified by the user, and discards it if the answer is 'no'. After this, another class member is selected from the remaining members of $C$, and the process is repeated until no more new clone classes can be found.

In the case that none or more than one maximal clone class is returned from the candidate clone class, i.e. the candidate clone class is not anti-unifiable as a whole, its sub-portion clone classes are examined too. As an example, the class candidate shown in Fig. 2 has four class member $E_1$, $E_2$ $E_3$ and $E_4$. By anti-unification this class is divided into two new clone classes $C_1 = \{E_1, E_3\}$ and $C_2 = \{E_2, E_4\}$. Clone members of $C_1$ are not anti-unifiable with class members of $C_2$ because of their different binding structure of variables. Suppose the minimum length of a cloned expression sequence to be reported is `3`, then the clone detector will continue to examine the two sub-portion clone classes $C_{1,3}$ and $C_{2,4}$. Examination of $C_{1,3}$ will return the whole clone class, while examination of $C_{2,4}$ returns two new classes, but because the two new clone classes are subclones of $C_1$ and $C_2$, they are discarded. Therefore the examination of clone class $C$ results in three new clone classes: $C_1$, $C_2$ and $C_{1,3}$.

```
S1 = "This",         S1 ="This"             D1= [1],          D1=[X+1],
S2 = " is a ",       S2 ="is another",      D2= [2],          D2=[5],
S3 = "string",       S3 ="String",          D3 =[3],          D3=[6],
[S1,S2,S3]           [S3,S2,S1]             [D1,D2,D3]        [D3,D2,D1]
   (E1)                  (E2)                  (E3)              (E4)
```

Fig. 2: An initial clone class candidate with four class members

This step dominates the overall cost of the clone detection algorithm. Examination of a candidate clone class of $n$ members has a worst case of $O(n^2)$ complexity.

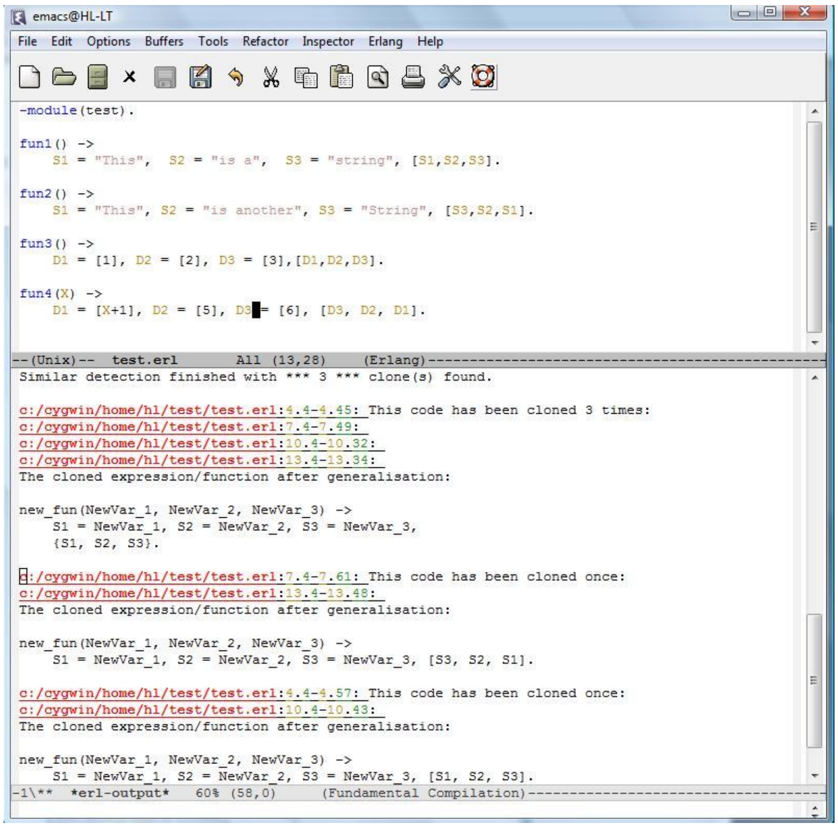
*Discussion.* More constraints can be applied during the anti-unification process so that certain kinds of node are not replaced by variables even if doing so is theoretically correct. For example, generalisation over expressions that contain locally declared free variables is possible, but doing so makes the program harder to understand, and may well not be of interest to the user. Another constraint would be the maximal number of new variables introduced during the anti-unification process, so as to avoid the generation of functions with too many variables, which represents another kind of bad code smell. We are currently working towards making the clone detector a customizable tool so that the user could specify which kinds of generalisation are preferred or not preferred.

**Formatting** Final clone classes are sorted and displayed in two different orders, first by the number of duplications, then by the length of expression sequences. The location of each clone member, identified by the combination of source file name, line and column numbers, is mouse clickable. Associated with each clone class is the least-general common abstraction of the clone class in form of a function definition. The function name and variable names of the form `NewVar_i` are generated by the clone detector. Variables that are declared locally but used elsewhere are included in the tuple returned by the function.

Fig. 3 shows the clone detection in action. The buffer above is an Erlang module consisting of four functions whose bodies correspond to the class members in Fig. 2, and the buffer below shows the result of running the clone detector on this buffer, illustrating the clones $C_{1,3}$, $C_2$, and $C_1$, as well as their anti-unifiers.

## 5 Refactoring Support for Similar Code Elimination

The primary purpose of clone detection is to identify them so that they can be eliminated. A number of Wrangler refactorings, together with the least-general common abstractions suggested by the clone detector, make clone elimination straightforward. With the current framework, the clone removal process involves the following steps:


emacs@HL-LT
File Edit Options Buffers Tools Refactor Inspector Erlang Help

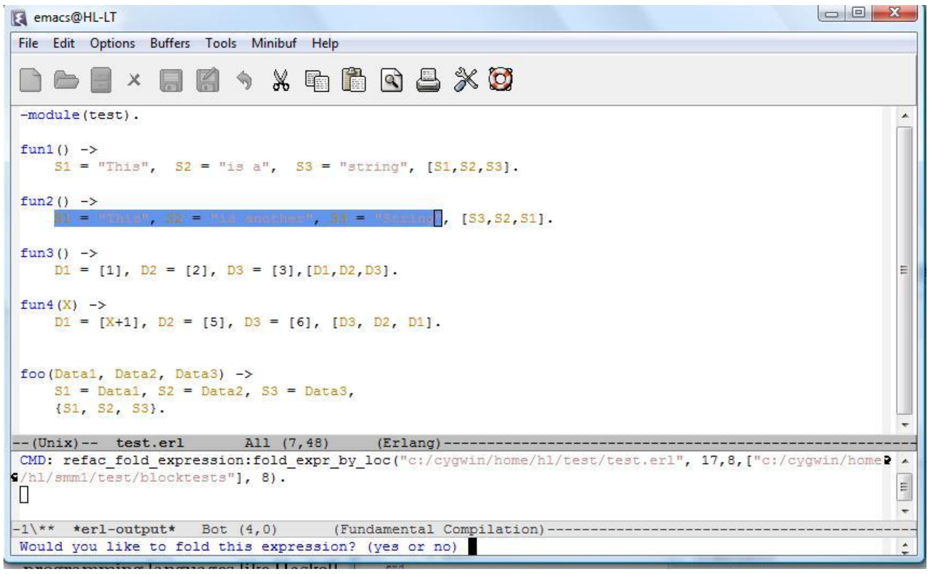
```
-module(test).

fun1() ->
    S1 = "This",  S2 = "is a",  S3 = "string", [S1,S2,S3].

fun2() ->
    S1 = "This", S2 = "is another", S3 = "String", [S3,S2,S1].

fun3() ->
    D1 = [1], D2 = [2], D3 = [3],[D1,D2,D3].

fun4(X) ->
    D1 = [X+1], D2 = [5], D3 = [6], [D3, D2, D1].

--(Unix)--  test.erl       All (13,28)     (Erlang)----------------------------------------------
Similar detection finished with *** 3 *** clone(s) found.

c:/cygwin/home/hl/test/test.erl:4.4-4.45: This code has been cloned 3 times:
c:/cygwin/home/hl/test/test.erl:7.4-7.49:
c:/cygwin/home/hl/test/test.erl:10.4-10.32:
c:/cygwin/home/hl/test/test.erl:13.4-13.34:
The cloned expression/function after generalisation:

new_fun(NewVar_1, NewVar_2, NewVar_3) ->
    S1 = NewVar_1, S2 = NewVar_2, S3 = NewVar_3,
    {S1, S2, S3}.

c:/cygwin/home/hl/test/test.erl:7.4-7.61: This code has been cloned once:
c:/cygwin/home/hl/test/test.erl:13.4-13.48:
The cloned expression/function after generalisation:

new_fun(NewVar_1, NewVar_2, NewVar_3) ->
    S1 = NewVar_1, S2 = NewVar_2, S3 = NewVar_3, [S3, S2, S1].

c:/cygwin/home/hl/test/test.erl:4.4-4.57: This code has been cloned once:
c:/cygwin/home/hl/test/test.erl:10.4-10.43:
The cloned expression/function after generalisation:

new_fun(NewVar_1, NewVar_2, NewVar_3) ->
    S1 = NewVar_1, S2 = NewVar_2, S3 = NewVar_3, [S1, S2, S3].
-1\**  *erl-output*   60% (58,0)     (Fundamental Compilation)-----------------------------------
```

Fig. 3: A snapshot showing similar code detection

1. select a clone class, copy and paste the least-general common abstraction into the proper Erlang module;
2. rename variable names if necessary;
3. re-order the function parameters if necessary;
4. rename the function to some suitable name;
5. apply the refactoring 'fold expressions against a function definition' to the new function.

Both *renaming* and *folding* are refactorings supported by Wrangler. *Reordering* of function parameters is not supported by Wrangler yet, but this does not add any overhead to the clone removal process as long as the reordering of parameters is done before 'folding' is applied, i.e. before the function is actually used.

*Folding expressions 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 the function clause selected, and replaces them with applications of the function to actual parameters under the



```
-module(test).

fun1() ->
    S1 = "This",  S2 = "is a",  S3 = "string", [S1,S2,S3].

fun2() ->
    S1 = "This", S2 = "is another", S3 = "String", [S3,S2,S1].

fun3() ->
    D1 = [1], D2 = [2], D3 = [3],[D1,D2,D3].

fun4(X) ->
    D1 = [X+1], D2 = [5], D3 = [6], [D3, D2, D1].


foo(Data1, Data2, Data3) ->
    S1 = Data1, S2 = Data2, S3 = Data3,
    {S1, S2, S3}.
--(Unix)--  test.erl       All (7,48)      (Erlang)------------------------------
CMD: refac_fold_expression:fold_expr_by_loc("c:/cygwin/home/hl/test/test.erl", 17,8,["c:/cygwin/home
/hl/smm1/test/blocktests"], 8).

-1\**  *erl-output*   Bot (4,0)      (Fundamental Compilation)-------------------
Would you like to fold this expression? (yes or no)
```

Fig. 4: A snapshot of Wrangler showing folding

user's control. This refactoring can not only detect instances where parameters are replaced by variables or literals, but also instances where parameters are replaced by arbitrary expressions. Expressions with side effects or locally declared variables are wrapped in a `fun` expression (or closure) to preserve the semantics. When this refactoring is initiated to a function clause selected, Wrangler automatically searches for code fragments that are clones of this function clause. Once clone instances have been found, the user can indicate whether to fold a particular clone instance or not. Folding is not performed within the selected function clause itself, since doing this will change the program's semantics.

Fig. 4 shows a snapshot of this refactoring in action. The user has chosen to apply 'folding' to the function `foo`. The expression sequence highlighted is one of the clone instances found by this refactoring, and the text shown in the minibuffer asks the user whether this clone instance should be removed. We should point out that the fact that Erlang is a weakly typed language and does not support polymorphism has made the clone detection and elimination process easier. For example, with Erlang programs we can be sure that `X+Y` and `A+B` are clones without carrying out complex type analysis, whereas this is not in general possible in strongly-typed programming languages like Haskell.

## 6 Clone Detection Applied

The clone detector has been applied to various Erlang applications and test code. Our case studies show that test code written under the Erlang/OTP Test Server framework has a much higher percentage of duplicated code than normal Erlang applications or test code written under other testing frameworks. This was not very surprising given the fact that all Erlang/OTP Test Server test functions follow a predefined coding pattern, and the *copy*, *paste*, then *modify* style of editing can be very tempting to testers.

One of test suites we have examined contains 4 Erlang modules, 9189 lines of code. This test suite is actually used by industry, and at the time we examined this test code more testing functions were still being added. With the default parameter settings, i.e. 5 for the minimum number of expressions, 2 for the minimum number of repeats, and 0.8 for the similarity score, it takes the clone detector less than 2 minutes to report 354 initial clone class candidates and 150 final clone classes. This was run on a laptop with Intel(R) 2.00 GHz processor, 2015MB RAM, and running Windows Vista. Of the 150 clone classes, the largest clone class, whose least-general common generalisation is shown below, contains a sequence of five match expressions with 75 instances across 3 modules.

```
new_fun(NewVar_1, NewVar_2, NewVar_3) ->
   FilterName_1 = "F_1",
   Pos = 1,
   FilterRuleSetList = [{FilterName_1, Pos, NewVar_1}],
   NetSide = NewVar_2,
   Dir = NewVar_3,
   NetDirFilterList = [{NetSide, Dir, FilterName_1}],
   {FilterRuleSetList, NetDirFilterList}.
```

The clone class with the longest expression sequence reports an expression sequence of 89 lines occurring twice in the same module with only two literal strings being different.

Working together with programmers familiar with the test suite and the application being tested, we looked to eliminate clones from the code. We took one of the test modules, containing 2600 lines of code, as an example: the clone detector reports 31 clone classes for this module. We started by removing clones with the largest number of repeats, thus working *bottom up.* Instead of devoting time to the details of the removal process, we were able to concentrate on its higher-level aspects, such as choosing how to name the functions representing the cloned code.

This experiment also showed the importance of user inspection during the clone elimination process. We have the Wrangler support for identifying candidates for clones but they may well need further analysis and insight from users to identify what should be done. For example, a clone might contain some expressions whose functionality belongs to the next part of the code, and should be removed from the least-general common generation before clone removal is applied, if the extracted function is to represent a meaningful operation.

## 7 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 by Roy and Cordy can be found in [2]. Overall there are

- text-based approaches [19,2,20], which consider the target program as sequence of lines/strings;
- token-based approaches [21,18,22], which apply comparison techniques to the token representation of programs.
- AST-based approaches [23,24,25,26,27], which search for similar subtrees in the AST with some tree matching techniques; and
- program dependency graph based approaches [21], which look for isomorphic subgraphs to find clones.

Our approach presented in this paper uses the AST-based approach. AST-based approaches in general could report more clones than text-based and/or token-based approaches, but since naïve comparison of subtrees for equality does not scale, various techniques are needed to make them scalable.

The most closely related work to ours is by Bulychev et al. [26] who also use the notion of anti-unification to perform clone detection in ASTs. Their approach consists of three steps, first identify similar statements using anti-unification and classify them into clusters, this is done by attempting anti-unification of each statement with each potentially matching cluster; then find identical sequences of cluster IDs, corresponding to statement sequences within a compound statement; after that anti-unification is used again to refine the candidate sequences identified previously for overall similarity. Anti-unification distance, which can be seen as the total size of subtrees to be replaced, is used to check the similarity of clone pairs.

Our approach is different from Bulychev et al.'s in several aspects. First, we use a different approach, which is faster but reports more false positives, to get the initial clone candidates; second, their approach reports only clone pairs, while our approach reports clone classes as well as their anti-unifiers; third, Bulychev et al.'s approach is programming language independent, and the quality of the algorithm depends on whether the occurrence of the same variable (in the same scope) refers to one leaf in the AST; whereas our tool is for Erlang programs, though the idea also applies to other languages, and static semantics information is taken into account to disallow inconsistent substitutions.

Another related work is by Evans et. al [24] who search for large common patterns in ASTs. It is based on heuristics and works in a bottom-up manner, specifying and increasing the patterns step-by-step. The disadvantage of this is that it can only find duplicated statements, not sequences of statements.

In [23], Baxter et al. use a hash function to place each full subtree of the AST into a bucket, then every two full subtrees with a bucket are compared. The hash function is chosen to be insensitive to identifier names so that these can be parameters in a procedural abstraction. In [23], Baxter et al. also suggest a mechanism for the removal of code clone with the help of macros. *DECKARD* [25] 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.

Most of the above mentioned clone detection tools target large legacy programs, and none of them is closely integrated with an existing programming environment, not to mention support for interactive automatic clone elimination. 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.

## 8 Conclusions and Future Work

In this paper, we have presented a similar code detection and elimination technique based on the notion of anti-unification, or least-general common abstraction, as well as techniques taken to improve performance and efficiency. The tool is able to detect more clones than Wrangler's original code detection tool, which only reports code fragments that are identical after consistent variable renaming and substitution of literals. The tool reports not only clones, but also the least-general common abstraction of each clone class in form of an Erlang function definition. The least-general common abstraction helps the user decide whether the clone is worth elimination or not, and also makes the clone removal process much easier. The clone detector tool is built on top of the infrastructure of Wrangler, the Erlang refactorer, and also integrated within the Wrangler environment. User-controlled automatic elimination of clones was made possible with Wrangler's refactoring support. Case studies carried out with real-world industrial code demonstrated the usefulness of the tool.

Our future work goes in a number of directions. While this paper lays out the infrastructure of the tool, in the future we are going to do an empirical study of clones detected from different Erlang systems with different parameter settings. Our current similar code detection tool cannot detect expression sequences which are similar up to a single insertion or deletion of an expression, or similar up to a number of expression-level edits, and we are trying to extend the tool to detect this kind of more general similarity. We would also like to explore the application of the approach to other functional programming languages like Haskell, in which case a type-aware anti-unification is needed.

This research is supported by EU FP7 collaborative project ProTest (`http://www.protest-project.eu/`), grant number 215868; we thank our funders and colleagues for their support and collaboration.

## References

1. Kapser, C., Godfrey, M.W.: "Clones Considered Harmful" Considered Harmful. In: Proc. Working Conf. Reverse Engineering (WCRE). (2006)
2. Roy, C.H., Cordy, R.: A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University at Kingston, Ontario, Candada
3. Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K.: Software Quality Analysis by Code Clones in Industrial Legacy Software. In: METRICS '02, Washington, DC, USA (2002)

4. Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K.: Partial Redesign of Java Software Systems Based on Clone Analysis. In: Working Conference on Reverse Engineering. (1999) 326–336
5. M. Fowler: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
6. Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: ARIES: Refactoring Support Environment Based on Code Clone Analysis. In: IASTED Conf. on Software Engineering and Applications. (2004) 222–229
7. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence **5** (1970) 153–163
8. Reynolds, J.C.: Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence **5** (1970) 135–151
9. Li, H., Lindberg, A., Schumacher, A., Thompson, S.: Improving your test code with Wrangler. Technical Report 4-09, School of Computing, Univ. of Kent, UK
10. Armstrong, J.: Programming Erlang. Pragmatic Bookshelf (2007)
11. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media, Inc. (2009)
12. S. Peyton Jones, ed.: Haskell 98 Language and Libraries: the Revised Report. Cambridge University Press (2003)
13. Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang Programs. In: EUC'06, Stockholm, Sweden (November 2006)
14. Li, H., Thompson, S., Orosz, G., Töth, M.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada
15. Li, H., Thompson, S.: Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In: PEPM'09, Savannah, Georgia, USA (January 2009)
16. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Computer Society Trans. Software Engineering **28**(7) (2002) 654–670
17. Ukkonen, E.: On-Line Construction of Suffix Trees. Algorithmica **14**(3) (1995) 249–260
18. Baker, B.S.: On Finding Duplication and Near-Duplication in Large Software Systems. In Wills, L., Newcomb, P., Chikofsky, E., eds.: Second Working Conference on Reverse Engineering, Los Alamitos, California (1995)
19. Baker, B.S.: A Program for Identifying Duplicated Code. Computing Science and Statistics **24** (1992) 49–57
20. S. Ducasse, M.R., Demeyer, S.: A language independent approach for detecting duplicated code. In: Proceedings ICSM99, IEEE (1999) 109–118
21. R. Komondoor and S. Horwitz: Tool Demonstration: Finding Duplicated Code Using Program Dependences. Lecture Notes in Computer Science **2028** (2001)
22. Li, Z., Lu, S., Myagmar, S.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. **32**(3) (2006) 176–192
23. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: ICSM '98, Washington, DC, USA (1998)
24. W. Evans, C.F., Ma, F.: Clone Detection via Structural Abastraction. In: the 14th Working Conference on Reserse Engineering. (2008) 150–159
25. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 96–105
26. Bulychev, P., Minea, M.: Duplicate code detection using anti-unification. In: Spring Young Researchers Colloquium on Software Engineering. (2008) 51–54
27. R. Koschke and R. Falke and P. Frenzel: Clone Detection Using Abstract Syntax Suffix Trees. In: WCRE '06, Washington, DC, USA (2006) 253–262