Incremental Clone 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 the existence of *code clones*, which are code fragments that are identical or similar to one another. This paper describes an approach to *incrementally* detecting 'similar' code based on the notion of least-general common abstraction, or *anti-unification*, as well as a framework for user-controlled incremental elimination of code clones within the context of Erlang programs. The clone detection algorithm proposed in this paper achieves 100% precision, high recall rate, and is user-customisable regarding the granularity of the clone classes reported. By detecting and eliminating clones in an incremental way, we make it possible for the tool to be used in an interactive way even with large codebases. Both the clone detection and elimination functionalities are integrated with Wrangler, a tool for interactive refactoring of Erlang programs. We evaluate the approach with various case studies.

Key words: Software maintenance, Refactoring, Code clone detection, Erlang, Program analysis, Program transformation, Erlang, 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.

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, but could also be avoided by first refactoring then reuse the existing code. In the last decade, substantial research effort has been put into the detection and removal of clones from software systems; however, few such practical tools are available for functional programming languages. The work reported here is of particular value both to the working programmer and the project manager of a larger programming project, in that it allows clone detection to contribute to the 'dashboard' reports from incremental nightly builds, for instance.

This paper describes an approach to incrementally detecting 'similar code' in Erlang programs based on the notion of *least-general common abstraction*, or *anti-unification* [1,2], as well as a mechanism for incremental automatic clone elimination under the user's control. We take Erlang as the target language due to our research context. While the implementation discussed in this paper is specific to Erlang's syntax and static semantics rules; the methodology used by the approach is applicable to other functional programming languages as well.

In general, we say two expressions or expression sequences, `A` and `B`, are *similar* if there exists a non-trivial least-general common abstraction, or anti-unifier, `C`, and two substitutions σ_A and σ_B which take `C` to `A` and `B` respectively. By 'non-trivial' we mainly mean that the size of the least-general common abstraction should be above some threshold, but other conditions, such as the complexity of the substitution, can also be specified.

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

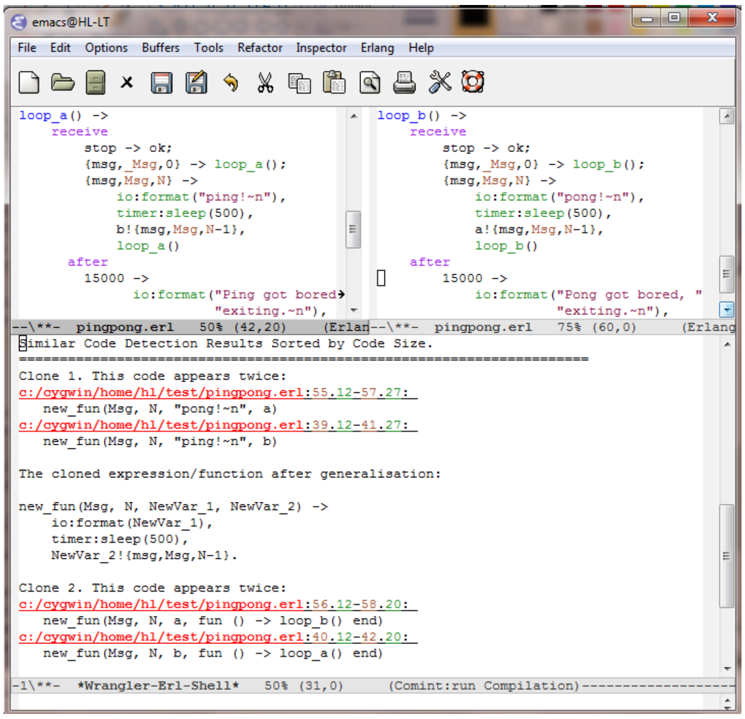
```
add(Y,Z) -> Y+Z.
```

When support for clone elimination is one of the major purposes served by a clone detection tool, accuracy and efficiency of the tool are essential for it to be usable in practice.

To achieve a 100% precision rate, i.e., only genuine clones are reported, our approach uses as the representation of an Erlang program the Abstract Syntax Tree (AST) for the parsed program annotated with static semantic information. Syntactic and static semantic information together make it possible that only those genuine, and syntactically well-formed, clones are reported to the user.

Scalability and efficiency, the major challenges faced by AST and/or semantics based clone detection approaches, are achieved by an *incremental* two-phase clone detection technique. The first phase uses a more efficient, but less accurate, syntactic technique to identify candidates which might be clones; the initial result is then assessed by means of an AST and static semantics based analysis during the second phase to give only genuine clones. When part of the codebase has been changed, the original clone detection result is no longer up-to-date, due to either the change of locations or the textual change of code. To keep the clone report up-to-date, instead of re-running the clone detection from scratch, the *incremental* clone detection algorithm reuses and updates the data collected from the previous run of the clone detection, and only processes clones related to the code that has been modified, added or deleted.

Our clone detection tool reports clone classes. As shown in the lower window of Fig. 1, each clone class is a set of code fragments in which any two of the code fragments are identical or similar to each other. Each clone class is associated with the least-general abstraction in the format of a function abstraction named `new_fun`. Variable names of the form `NewVar_i` are generated by the clone detector. For each member of a clone class, the clone detector also generates an


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

```
loop_a() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_a();
        {msg,Msg,N} ->
            io:format("ping!~n"),
            timer:sleep(500),
            b!{msg,Msg,N-1},
            loop_a()
      after
        15000 ->
            io:format("Ping got bored
                      "exiting.~n"),
--\**-  pingpong.erl   50% (42,20)     (Erlan
```

```
loop_b() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_b();
        {msg,Msg,N} ->
            io:format("pong!~n"),
            timer:sleep(500),
            a!{msg,Msg,N-1},
            loop_b()
    after
        15000 ->
            io:format("Pong got bored, "
                      "exiting.~n"),
--\**-  pingpong.erl   75% (60,0)      (Erlang
```

```
Similar Code Detection Results Sorted by Code Size.
======================================================================
Clone 1. This code appears twice:
c:/cygwin/home/hl/test/pingpong.erl:55.12-57.27:
   new_fun(Msg, N, "pong!~n", a)
c:/cygwin/home/hl/test/pingpong.erl:39.12-41.27:
   new_fun(Msg, N, "ping!~n", b)

The cloned expression/function after generalisation:

new_fun(Msg, N, NewVar_1, NewVar_2) ->
    io:format(NewVar_1),
    timer:sleep(500),
    NewVar_2!{msg,Msg,N-1}.

Clone 2. This code appears twice:
c:/cygwin/home/hl/test/pingpong.erl:56.12-58.20:
   new_fun(Msg, N, a, fun () -> loop_b() end)
c:/cygwin/home/hl/test/pingpong.erl:40.12-42.20:
   new_fun(Msg, N, b, fun () -> loop_a() end)

-1\**-  *Wrangler-Erl-Shell*   50% (31,0)     (Comint:run Compilation)------------------
```

Fig. 1. A snapshot showing similar code detection with Wrangler

application instance of `new_fun`, which is the function call that arises from unifying the class member with the function definition represented by `new_fun`.

One aim of clone detection is to identify them so that they can be eliminated. The general approach to removing a cloned code fragment in the functional programming paradigm is to replace it with a function call to the least-general common abstraction of the clone class to which the code fragment belongs. In theory, it is possible to eliminate all clones found fully automatically without control from the user, however in practice, this is not a desirable way for various reasons. Instead of eliminating clones fully automatically, our framework allows the user to eliminate clones in an *incremental* way. We automate things that should be automated, while allowing the user to have control over the clone elimination process when it is necessary.

A non-incremental similar code detection and elimination framework was first added to Wrangler in 2009 [3]. The contribution of this paper is to provide:

– an incremental algorithm which works substantially more efficiently than the original standalone algorithm for larger projects in analyses;
– a framework that gives the user fine control of the granularity of the clones reported and a clear vision of the code after eliminating a specific clone;
– a method for tracking the evolution of clones during software lifetime, and

– a mechanism for clone detection to be included in the standard workflow and reporting of projects subject to continuous integration or regular builds.

The remainder of the paper is organised as follows. Section 2 gives an overview of Erlang and the refactoring tool Wrangler. Section 3 introduces some terminology to be used in this paper; Section 4 describes the incremental clone detection algorithm. The elimination of code clones is discussed in Section 5, and an evaluation of the tool is given in Section 6. Section 7 gives an overview of related work, and Section 8 concludes the paper and briefly discusses future work.

2 Erlang and Wrangler

Erlang [4] 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. 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.

An Erlang program typically consists of a number of modules, each of which defines a collection of functions. Only functions exported explicitly through the `export` directive may be called from other modules. In Erlang, a function name can be defined with different arities, and the same function name with different arities can represent entirely different functions computationally.

Wrangler [5] is a tool that supports interactive refactoring of Erlang programs. It is integrated with (X)Emacs, as shown in Fig. 1, and as well as with Eclipse through ErlIDE. Wrangler is implemented in Erlang, and downloadable from `http://www.cs.kent.ac.uk/projects/wrangler/Home.html`.

3 Terminology

Anti-unification and Unification Anti-unification, first proposed in 1970 by Plotkin [1] and Reynolds [2], applies the process of *generalisation* to pairs, or sets, of terms. The resulting term captures all the commonalities of the input terms. Given terms $E_1, ..., E_n$, we say that E is a *generalisation* of $E_1, ..., E_n$ if there exist substitutions σ_i for each $E_i, 1 \leq i \leq n$, such that $E_i = E\sigma_i$. E is a *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 θ such that $E = E'\theta$; it is not difficult to see that these are unique (up to renaming of variables) and so we call any of them *the* least-general common generalisation. 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. Furthermore, an AST subtree which exports some of its locally declared variables should not be replaced by a variable either; whereas it is fine to substitute the function name in a function application with a variable because higher order functions are supported by Erlang.

Unification, on the other hand, is the process of finding substitutions of terms for variables to make expressions identical [6]. The unification technique forms the basis of the clone elimination process.

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: $\min\{S_E/S_{E_1}, ..., S_E/S_{E_n}\}$, 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.

Clone Classes. A *clone class* is a set of code fragments in which any two of the code fragments are identical or similar to each other. In the context of this paper, each member of a clone class is a sequence of Erlang expressions. We say a clone class C is maximal if there does not exist a clone class C' such that $|C| \leq |C'|$, and for each class member, E_i say, of C, there exists a clone member, E_j say, of C', such that E_i is a proper sub-sequence of E_j. Only those maximal clone classes are reported by our clone detection tool.

4 The Clone Detection Algorithm

The similar code detector takes a project (that is, a set of Erlang modules), and a set of parameters as input, performs clone detection, and reports the clone classes found. The tool is integrated with an IDE (Emacs or Eclipse), but it can also be run from the command line. Five parameters need to be specified; if no value is supplied a suitable default value is used. The parameters are:

- the minimum number of expressions included in a clone which is a sequence of expressions, denoted by E_{min};
- the minimum number of lexical tokens included in a clone, denoted by T_{min};
- the maximum number of new parameters of the least-general common abstraction function, denoted by P_{max};
- the minimum number of class members of a clone class, denoted by F_{min};
- the similarity score threshold, denoted by $SimiScore$.

With these parameters, the user can have a fine control of the granularity of the clone classes reported. For example, to make the tool only report identical code fragments, the user just need to set the value of P_{max} to 0.

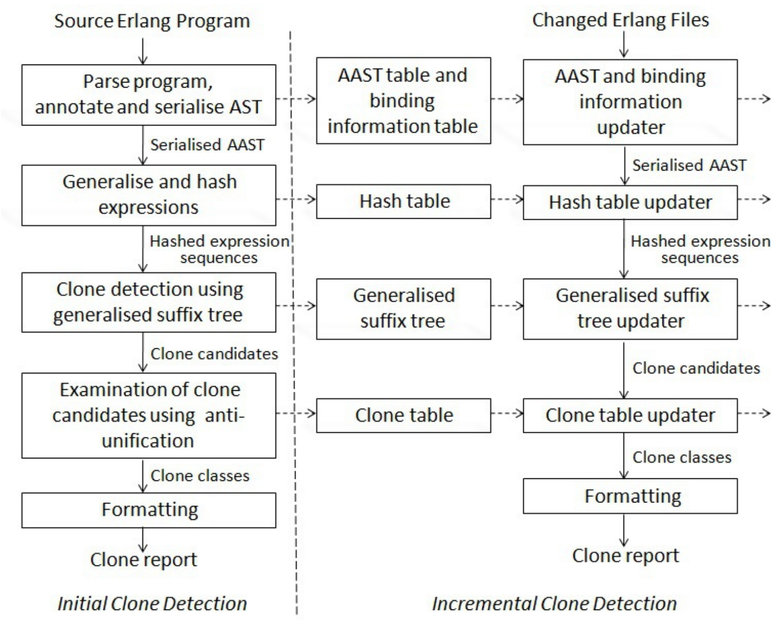

Source Erlang Program
Parse program, annotate and serialise AST
Serialised AAST
Generalise and hash expressions
Hashed expression sequences
Clone detection using generalised suffix tree
Clone candidates
Examination of clone candidates using anti-unification
Clone classes
Formatting
Clone report
Initial Clone Detection
AAST table and binding information table
Hash table
Generalised suffix tree
Clone table
Changed Erlang Files
AAST and binding information updater
Serialised AAST
Hash table updater
Hashed expression sequences
Generalised suffix tree updater
Clone candidates
Clone table updater
Clone classes
Formatting
Clone report
Incremental Clone Detection

Fig. 2. An Overview of the Incremental Clone Detection Process

As shown in Fig. 1, each clone class is reported by giving the number of instances of the cloned code, the least-general common generalisation in the format of a function abstraction named `new_fun`, each clone instance's start and end locations, as well as the application instance of `new_fun`, which is the function call that arises from unifying the class member with function definition represented by `new_fun`. Scalability is tackled from two aspects:

- A two-phase clone detection technique is used. The first phase carries out a quick, semantics-unaware clone detection over a generalised version of the program, and reports initial clone candidates; the second phase examines the initial clone candidates in the context of the original program by means of anti-unification, getting rid of false positives;
- Incremental update of information collected from the various stages of the previous run of the clone detection tool when program code has been changed.

An overview of the algorithm is shown in Fig. 2. Next, we first describe the initial clone detection algorithm, then the incremental part.

4.1 The Initial Detection Algorithm

The initial clone detection algorithm is an extended and adapted version of the standalone algorithm presented in [3]. While both follow the same steps, the algorithm presented here is designed to make incremental clone detection possible, and provides better usability. We explain it in more detail now.

Parse Program, annotate and serialise AST. Erlang files are lexed and parsed into ASTs. In order to reflect the original program text, the Erlang pre-processor is bypassed to avoid macro expansion, file inclusion, conditional compilation, etc. Both line and column numbers of identifiers are kept in the ASTs generated, since location information is used to map between different representations of the same piece of code in the source. Binding information of variables and function names, which is needed during the anti_unification process, is annotated to the AST in terms of defining and use locations.

Location information is needed by the clone detector, but absolute locations are sensitive to changes, i.e. a change made to a particular place of a file could possibly affect all the locations following it. With incremental clone detection in mind, we choose to use relative locations to identify program entities, while recording each function's actual starting line in the file. With relative locations, every function starts from line 1 at column 1. The benefit of this is that we can ensure pure location change does not affect the initial clone candidate result, and all new clone candidates introduced are due to structural change.

The AAST representation of each function is then traversed, and expression sequences are collected. In this way, each function is mapped into a list of expression sequences. The AAST representation of each expression statement is stored in an ETS (Erlang Term Storage) table for use by the later stages of the algorithm and the incremental clone detection. To reduce the time spending on AAST traversal while trying to locate a specific syntax phrase in the AAST, and also to reduce the time on AAST updating when incremental clone detection is concerned, each object in the ETS table represents the AAST of a single expression statement, instead of the AAST of a whole Erlang file.

Generalise and Hash Expressions. This step takes an expression sequence generated from the previous step a time, each expression statement in the sequence is first structurally generalised, then pretty-printed and hashed into an integer value. Only expression statements that share the same generalised form get the same hash value. Therefore, we map each expression sequence into a sequence of integers.

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 replaces 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 `receive` expression, a `try...catch` expression, etc.

Initial Clone Detection using a Generalised Suffix Tree This step turns each integer sequence generated from the previous step into a string, and builds a generalised suffix tree on the strings generated. Cloned strings are then collected from the suffix tree, and each group of cloned strings is then mapped back to a group of expression sequences which share the same generalised representation.

Suffix tree analysis [7] is the technique used by most text or token-based clone detection approaches because of its speed [8,9]. A suffix tree is a representation of a single 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. A generalised suffix tree is a suffix tree that represents the suffix of a set of strings. Fig. 3 shows an example of the generalised suffix tree representation of two strings `ABAB$` and `BAB$`. The numbers in the leaf nodes are string number and starting position of the suffix in the string. We use generalised

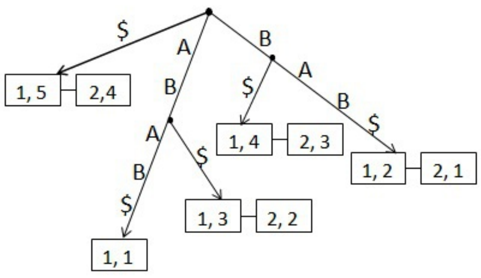


Fig. 3. A generalised suffix tree for two strings `ABAB$` and `BAB$`

suffix tree instead of the standard suffix tree as in [3] for two reasons:

- The standard suffix tree algorithm accepts only a single string as input. To build a suffix tree over a collection of strings, we will have to concatenate all the strings into a single one, and then build a suffix tree over the concatenated string. As a result, clone strings might actually come from two or more separated strings, and therefore need to be further processed;
- With generalised suffix tree, a string can be removed from, or inserted into, the tree easily, which is exactly what we need for incremental clone detection.

Examine Clone Candidates using Anti-unification. The previous step returns a collection of clone classes whose class members are structurally similar, but do not necessarily 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 returns those genuine clone classes. More details follow.

Generation of clone classes from clone candidates. For each clone class candidate, C say, the clone detector takes a class member, A say, and tries pairwise anti-unification with each of the other class members. The anti-unification result is then analysed and processed to derived clone classes that satisfy all the parameters specified by the user. In order to achieve a high recall rate, when two expression sequences do not anti-unify as a whole, their sub-sequences are also examined. If the whole clone class candidate does not form a real clone class, 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. As a result, it is possible to derive none, one, or more clone classes from a single clone candidate. For example, from a clone class candidate with three expression sequences like $\{E_{11}E_{12}E_{13},\ E_{21}E_{22}E_{23},\ E_{31}E_{32}E_{33}\}$, it is entirely possible to derive three clone classes like: $\{E_{11}E_{12},\ E_{21}E_{22},\ E_{31}E_{32}\}$, $\{E_{11}E_{12}E_{13},\ E_{21}E_{22}E_{23}\}$ and $\{E_{21}E_{22}E_{23},\ E_{31}E_{32}E_{33}\}$.

Generation of anti-unifier. The anti_unifier generator takes a clone class and the substitutions inferred during the anti-unification process as input, generalises the expression sequence represented by the first clone class instance over those sub-expressions which are not common to all of the clone instances by replacing those sub-expressions with new variables automatically generated by the clone detector. To ensure that only the minimum number of new parameters are generated, the anti-unifier generator needs to check the static semantics of the sub-expressions to be generalised over, and their corresponding sub-expressions in remaining class instances, so that sub-expressions with the same static semantics and same substitutions are represented by the same new variable. Variables declared by a cloned code fragment in a clone class might be used by the code following it, and the union of these variables should be returned by the anti-unifier so that those variables are still visible to the code following it when the cloned fragment is replaced by an application instance of the anti-unifier.

Generation of application instances. Given the anti-unifier of a clone class and a particular instance of the clone class, the application instance is generated through the unification of the anti-unifier, represented as a function definition named `new_fun`, and the class instance. The application instance gives the user a clear vision of what a cloned code fragment will be replaced with by clone elimination, and therefore helps the user decide whether this clone instance should be eliminated or not. For example, if some of the parameters of the application instance are too complex, the user might want to refactor the code first, then eliminate the clone.

Formatting. Final clone classes are sorted and displayed in three different orders: by the number of duplications, by the size of clone class instances, and by the ranking score of each clone class. The ranking score of a clone class is calculated based on three parameters: the number of parameters of the anti-unifier, the number of terms returned by anti_unifier, and size of the anti-unifier body. Given a clone class anti_unifier, suppose the above three parameters are represented by P, V and L respectively, the ranking score is calculated as: $L/(L+P+V)$.

4.2 The Incremental Detection Algorithm

A change made to a file could affect the existing clone results in two different ways. A structural change could introduce new clone classes, or invalidate some existing clone classes; a location change on the other hand could make the location information of some clone class members out-of-date. Obviously, a structural change to one part of the file is generally accompanied by location changes to the code following it in the file.

To incrementally update the clone result after changes have been made to the program source, our algorithm reuses and updates the intermediate results returned from the previous run of the clone detection as shown on right-hand side of the diagram in Fig. 2.

The algorithm takes a function as a unit to track the changes made to the program source. For a function that is removed, added, or structurally changed, we have no other choices but to remove/add/update the entities associated with it; whereas for a function with only a location change, only its absolute starting location is updated. Taking a function, instead of a file as the unit for tracking changes, we are able to reuse existing results as much as possible. Next we explain in more detail the various intermediate results that are reused and updated during the incremental clone detection phase.

The AAST Table. The AAST table stores the AAST representation of each expression statement. Each entry of the table is a tuple. The first element of the tuple, which serves as the key, is of the format: {`ModuleName, FunctionName, Arity, ExprIndex`}, where `ModuleName`, `FunctionName`, and `Arity` together identify a function, and `ExprIndex` is used to identify an expression statement of this function; the second element of the tuple is the AAST representation of the expression statement, together with the absolute starting line number of the function to which the expression belongs. The AAST updater checks which part of the program has been changed since the last run of the clone detection, and updates the AAST table accordingly. For a function that is deleted, added or modified structurally, the entries associated with that function in the AAST table are also deleted, added, or updated. Only the starting line number is updated if a function only has its location changed.

The Binding Information Table. This table stores the binding structure information for variables of each function. Each entry of the table is a tuple with the first element identifying a function, and the second element representing the binding structure in terms of defining and use locations. This table is updated only if a function has been deleted, added or structurally modified.

The Expression Hash Table. This table stores the mapping from an expression statement to its hash value, therefore from an expression sequence to a sequence of hash values, as well some meta-information about the expression, including the number of lexical tokens, location information, and a boolean flag indicating whether the expression is new. Each expression statement in this table is also identified by a tuple, whose first element is an integer uniquely identifying the expression sequence to which the expression statement belongs, and the second element is a four-element tuple {`ModuleName, FunctionName, Arity, ExprIndex`} identifying the particular expression.

The Generalised Suffix Tree. The reuse of the generalised suffix tree allows us to avoid re-building the suffix tree for the whole program from scratch. The generalised suffix tree is updated by removing those strings deleted/changed from it, and by adding the new strings into it. To avoid the re-calculation of clones from the suffix tree, we annotate each internal node with the clone information represented by that node, which is also updated accordingly.

The Clone Table. This table stores the mapping between each initial clone candidate and the clone classes derived from it. Each entry is a tuple whose first element is the clone candidate, and second element is the clone classes derived from it. A clone candidate is only processed if it does not belong to this table, and the result is then added to the table. By tracking changes of the clone table, we are able to track the evolution of clones during software development.

5 Support for Clone Elimination

Working within the functional programming paradigm, the general approach to removing a cloned code fragment is to replace it with an application of a function whose definition represents an abstraction of the cloned code fragment. In theory, it is possible to eliminate all clones found fully automatically, however, our experience [10] shows that this is undesirable for the following reasons:

– Some cloned code fragments logically do not represent a clearly defined functionality; or a cloned code fragment might contain code that logically belongs to the code before, or after, it in the program source.
– The least-general common abstraction generated by the clone detector has to be given a proper name to reflect its functionality; and the parameters of the least-general common abstraction might need to be renamed or re-ordered.
– In the case that a clone class contains code fragments from multiple modules, a proper module has to be selected as the host module of the least-general common abstraction to avoid introducing bad modularity smells.
– Fully automatic clone detection could introduce too many changes to the code base in one go, and makes it harder for the user to follow.

Our clone elimination framework tries to automate things that should be automated, while allowing the user to have control over the clone elimination process when it is necessary. With this framework, the following steps can be followed to eliminate some, or all, cloned code fragments from a given clone class.

1. Copy and paste the anti-unifier of the clone class into an 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 ‘fold expressions against a function definition’ to the new function.

Renaming, *reordering of function parameters* and *folding expressions against a function definition* are all refactorings supported by Wrangler. *Folding* is the refactoring which actually removes code clones. It searches for, and highlights, clone instances of the function clause selected, and replaces those clone instances which the user chooses to eliminate with application instances of the function selected. This refactoring carries out its own clone instance search, and can therefore be applied independently of the clone detection process.

Given a clone report containing a list of clone classes, the user has a number of decisions to make as to which clone classes, or even which instances of a specific clone class, to eliminate. We believe that by showing the least-general common abstraction of each clone class and the application instance associated with each clone class instance, we make this process much easier.

6 Experimental Evaluation

The tool has been applied to various Erlang application code and testing code. In this paper, we take three codebases to evaluate the efficiency and accuracy of the approach. The first codebase is Wrangler itself; the second codebase is an Erlang test suite written with Erlang's Common Test framework from a mobile industry; and the third codebase includes the application and testing code of both the Erlang compiler and the Erlang stdlib. The experiments were conducted on a laptop with Intel(R) 2.27 GHz processor, 4.00 GB RAM, and running Windows 7. The tool is evaluated in two criteria: runtime efficiency and the number of clones detected. To contrast the performance, we run both the incremental and the standalone clone detection for each codebase and version.

The default parameter setting for the tool, i.e. 5 for E_{min}, 40 for T_{min}, 4 for P_{max}, 2 for F_{min} and 0.8 for $SimiScore$, was used throughout the experiments. The experimental results are shown in Table 1. The first column of the table

Wrangler	KLOC	Files Changed	Incremental		Standalone	
			Time	Clones	Time	Clones
0.8.7	42.5	70/70	15	18	15	18
0.8.8	44.2	59/80	10	21	15	21
0.8.9	47.4	44/83	8	26	16	26
0.9.0	48.0	9/84	2	26	16	26
0.9.1	48.1	3/84	3	26	17	26
Test Suite						
V0	24.0	26/26	560	371	560	371
V1	23.9	3/26	90	361	550	361
V2	23.9	1/26	54	357	550	357
V3	23.8	2/26	90	346	550	346
V4	23.7	2/26	80	338	540	338
Erlang						
R13B-03	244.3	306/306	94	78	94	78
R13B-04	245.5	71/311	36	79	97	79
R14A	250.8	108/327	40	82	95	82
R14B	251.9	39/321	28	81	94	81

Table 1. Incremental vs. Standalone Clone Detection

shows the codebases and their versions we used. For both Wrangler and the Erlang compiler/stdlib, the version numbers are the release numbers, therefore the amount of changes made between two consecutive versions could be large; for the test suite, we use the original test suite as version V0, and each version following represents the codebase after some clones have been eliminated. The second column shows the size of each version of codebase in terms of the number of lines of code. The third column shows the number of files that are changed since the previous version out of the total number of files. The time is measured in seconds, and the clones are measured by the number of clone classes reported.

It is obvious from this table that the processing time can be reduced significantly especially when the amount of changes made between two consecutive versions is small. The tool performance is affected by both the size of the code and the number/size of the initial clone candidates detected. Table 1 shows that the processing time for the test suite is considerably long compared to the other two codebases. This is due to the large amount of clones, and clone candidates, detected. For example, the clone report for the test suite V0 says that the longest clone consists of 86 lines of code, and is duplicated twice; and the most frequently cloned code consists of 5 lines of code, and is duplicated 83 times.

The precision of the tool should be 100% due to the use of static semantics aware analysis during the clone candidate examination phase, and this has been verified through various case studies during the development of the tool. Any false positives reported simply imply a bug in the tool implementation. As to the recall rate, given a set of parameter settings, our tool in theory should be able find all those classes whose members do not textually overlap; but because it is practically impossible to examine this manually with large codebases, and there are no other clone detection tools for Erlang which we can use for comparison, at this stage we cannot give a concrete recall rate for the results reported here.

Compared to other clone detection tools, our tool takes precision and usability of the tool as the top priority. A recent study [11] has shown that up to 75% of clones detected by state-of-the-art tools are false positives, and this has hindered the adoption of clone detection techniques by software developers, no matter how fast the tool is and how large of a code base the tool can work with, as inspection of false positives is a waste of developer time.

7 Related Work

A survey by Roy et. al. provides a qualitative comparison and evaluation of the current state-of-the-art in clone detection techniques and tools [12]. Overall there are text-based approaches [13,14], token-based approaches [8,15], AST-based approaches [16,17,18,19] and program dependency graph based approaches [20]. AST-based approaches in general are more accurate, and could report more clones than text-based and/or token-based approaches, but various techniques are needed to make them scalable. A comparison and evaluation of these techniques in terms of recall and precision as well as space and time requirements has been conducted by Bellon et. al., and the results are reported in [21].

Closely related work to ours is by Bulychev et al. [19] who also use the notion of anti-unification to perform clone detection in ASTs. 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.

In [22], Brown and Thompson describe an AST-based clone detection and elimination tool for Haskell programs. While their approach works on small Haskell programs, scalability is still the problem for the authors to address.

ClemanX [23] is an incremental tree-based clone detection tool developed by Nguyen et al. Their approach measures the similarity between code fragments based on the characteristic vectors of structural features, and solves the task of incrementally detecting similar code as an incremental distance-based clustering problem. In [24], Göde and Koschke describe a token-based incremental clone detection technique, which makes use of the technique of generalised suffix trees.

8 Conclusions and Future Work

We have presented an incremental clone detection and elimination technique which can be used interactively during a clone inspection and elimination process, but also of particular value both to the working programmer and the project manager of a larger programming project, in that it allows clone detection to contribute to the 'dashboard' reports from nightly builds, for instance.

For a tool to be used in an interactive way, performance and efficiency are especially important. This goal is achieved by our tool to incrementally update the clone report after changes have been made to the program. Being able to specify various parameter settings to the clone detector, the user has more control of the granularity of the clones reported. Using the AST as the internal representation of Erlang programs, and being static semantics aware, the clone detection tool achieves 100% accuracy, which is essential when clone elimination is concerned. To support clone elimination, our tool reports not only the common abstraction of a clone class, but also the application instance of each clone class member. The tool has been used in various industry case studies [10], during which its usability has been improved, and usefulness has been demonstrated.

As future work, we plan to extend the tool to detect expression sequences which are similar up to a single insertion or detection of an expression, or similar up to a number of expression-level edits. Our overall approach is language-independent, and we also plan to apply our techniques to clone detection and elimination to test languages such as TTCN-3.

This research is supported by EU FP7 collaborative project ProTest (`http://www.protest-project.eu/`), grant number 215868.

References

1. Plotkin, G.D.: A Note on Inductive Generalization. Machine Intelligence **5** (1970)
2. Reynolds, J.C.: Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence **5** (1970) 135–151
3. Li, H., Thompson, S.: Similar Code Detection and Elimination for Erlang Programs. In: Practical Aspects of Declarative languages 2010. 104–118
4. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media, Inc. (2009)
5. Li, H., et al.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada
6. Baader, F., Siekmann, J.H.: Unification Theory. Handbook of logic in artificial intelligence and logic programming (1994) 41–125
7. Ukkonen, E.: On-Line Construction of Suffix Trees. Algorithmica **14**(3) (1995)
8. Baker, B.S.: On Finding Duplication and Near-Duplication in Large Software Systems. In Wills, L., et al., eds.: WCRE, Los Alamitos, California (1995)
9. 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
10. H. Li, A. Lindberg, A. Schumacher and S. Thompson: Improving your Test Code with Wrangler. Technical Report 4-09, School of Computing, University of Kent
11. Tiarks, R., Koschke, R., Falke, R.: An Assessment of Type-3 Clones as Detected by State-of-the-art Tools. In: SCAM'09, Los Alamitos, CA, USA (2009)
12. Roy, C.K., et al.: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Sci. Comput. Program. **74**(7) (2009)
13. Baker, B.S.: A Program for Identifying Duplicated Code. Computing Science and Statistics **24** (1992) 49–57
14. S. Ducasse, M.R., Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In: Proceedings ICSM99, IEEE (1999) 109–118
15. 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
16. 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)
17. W. Evans, C.F., Ma, F.: Clone Detection via Structural Abastraction. In: the 14th Working Conference on Reserse Engineering. (2008) 150–159
18. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: ICSE '07. (2007) 96–105
19. Bulychev, P., Minea, M.: Duplicate Code Detection using Anti-unification. In: Spring Young Researchers Colloquium on Software Engineering. (2008) 51–54
20. R. Komondoor and S. Horwitz: Tool Demonstration: Finding Duplicated Code Using Program Dependences. Lecture Notes in Computer Science **2028** (2001)
21. Bellon, S., Koschke, R., Society, I.C., Antoniol, G., Krinke, J., Society, I.C., Merlo, E.: Comparison and Evaluation of Clone Detection Tools. IEEE TSE **33** (2007)
22. Brown, C., Thompson, S.: Clone Detection and Elimination for Haskell. In: PEPM'10: Partial Evaluation and Program Manipulation. 111–120
23. Nguyen, T.T., Nguyen, H.A., Al-Kofahi, J.M., Pham, N.H., Nguyen, T.N.: Scalable and Incremental Clone Detection for Evolving Software. ICSM'09 (2009)
24. Göde, N., Koschke, R.: Incremental Clone Detection. In: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering. (2009)