Clone Detection and Elimination for Haskell

Christopher Brown Simon Thompson
School of Computing, University of Kent, UK.
chris@techniumcast.com, S.J.Thompson@kent.ac.uk

Abstract

Duplicated code is a well known problem in software maintenance and refactoring. Code clones tend to increase program size and several studies have shown that duplicated code makes maintenance and code understanding more complex and time consuming.

This paper presents a new technique for the detection and removal of duplicated Haskell code. The system is implemented within the refactoring framework of the Haskell Refactorer (HaRe), and uses an Abstract Syntax Tree (AST) based approach. Detection of duplicate code is automatic, while elimination is semi-automatic, with the user managing the clone removal. After presenting the system, an example is given to show how it works in practice.

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

General Terms Languages, Design

Keywords Haskell, refactoring, HaRe, duplicated code, program analysis, program transformation, generalisation

1. Introduction

The existence of duplicated or similar code (often called "code clones") is a well known problem in refactoring and software maintenance. The term *duplicate code*, refers to code fragments that bear a strong relationship: they might be literally identical, or similar up to changing values of literals, or indeed may share a common generalisation. We will explain our particular approach in due course.

Several studies have shown that software systems with code clones are more difficult to maintain than the software systems with little or no duplicated code [Roy, Cordy and Koschke 2009, Monden et al. 2002]. It is therefore beneficial to remove cloned code quickly after its introduction by the use of refactoring technology.

Software clones appear for a variety of reasons; the most obvious of which is the use of the "copy and paste" functions in an editor. Clones introduced by this mechanism often indicate a poor design with a lack of function abstraction or encapsulation [Chou et al. 2001, Li et al. 2006]. Refactoring can be used to eliminate this problem either by first transforming the code to make it more reusable, without duplicating code; or by transforming the code clones at a later stage in the refactoring process [Fowler 1999].

In the last decade, a substantial amount of research has gone into duplicate code detection and removal from software systems; a recent survey and taxonomy gives an encyclopaedic overview [Roy, Cordy and Koschke 2009]. However, few such tools are available for functional programs, and there is a particular lack of clone detection support within existing program environments.

This paper details a clone detection technique for Haskell [Peyton Jones and Hammond 2003] built into the framework of HaRe, the Haskell Refactorer [Li et al. 2003]; the tool described here is able to handle large Haskell programs. In particular, the tool is covers the full Haskell 98 language; works with multiple module programs; preserves layout and is embedded within Emacs [Cameron et al. 2004] and Vim [Oualine 2001].

Fully automating the tool would be dangerous because it could lead to results that the user would not expect; it is better to give the user control over which clones should be removed and extracted. The clone detection stage is fully automatic, while the transformation stage is user controlled. Being part of the programming environments most commonly used by Haskell programmers, the clone detection and elimination tools are more likely to be used than those in a stand-alone utility.

The clone detection technique presented in this paper has a more inclusive criterion for similarity than other clone detection mechanisms: it is concerned with finding clones which are common substitution instances of non-trivial expressions, rather than mere textual duplications. The clone detection technique is able to report code fragments in a Haskell program that share a non-trivial generalisation or *anti-unification*. Syntactically, these code clones are a sequence of well-formed expressions, and therefore this approach makes use of the token stream and AST provided in HaRe as part of Programatica [Hallgren 2003]. User intervention allows clones to be removed in a step-by-step process under the programmer's control, and allows for the preservation of clones of particular kinds (for example very small clones).

In addition to this, the clone detection technique has been extended to look for repeated sequences of monadic "commands". The clone detection and elimination technique as presented here are novel in dealing with a pure functional language with monadic effects. Clone detection systems have been developed in the past for imperative languages, however, due to referential transparency, the scope for detecting and (particularly) eliminating clones for a pure language is much greater.

The work presented here complements that on clone detection in Erlang using Wrangler [Li and Thompson 2009, 2010]. This process uses a hybrid mechanism: a token based technique for identifying code clones, which are then checked for static semantics using the annotated AST. The process here is somewhat different, in using a purely AST based approach. It also goes beyond the purely functional in taking into account monadic sequences within the source program.

PEPM'10, January 18–19, 2010, Madrid, Spain.

Section 2 gives an overview of work relating to duplicated code elimination. Section 3 discusses the analysis stage of the clone removal process for HaRe and Section 4 discusses the transformation stage of the clone removal process.

2. Related Work

A recent overview of clone detection and elimination work is given by Roy, Cordy and Koschke in [Roy, Cordy and Koschke 2009], and their analysis gives a multi-way taxonomy of the substantial body of existing work. First, it is possible to divide techniques according to the mechanisms used to detect clones.

Text-based Techniques

With this approach, the program is considered a sequence of strings (*i.e.* lines), and the mechanism looks for identical sequences of strings across two or more files. So, this mechanism makes no attempt to identify clones which are the same up to differences in, for example, literals, but looks for strict identity of lines. To speed up such approaches it is possible to compute 'fingerprints' of lines, as discussed in, for example, [Johnson 1993].

Token-based Techniques

Programs are naturally sequences of tokens, and so it makes sense to process the input programs into token sequences before further processing. This allows clone-detection to consider more compact input files, as well as being less sensitive to superficial changes such as layout, comments and white space characters.

As well as simple tokenising, it is possible to abstract away somewhat from program structure with this approach. For instance, different identifiiers can be replaced by an 'identifier' token, so that differences in names do not impede the detection of cones which are equivalent up to change of name. Similarly, literal expressions, such as numbers, can also be elided. This approach is used in *CCFinder* [Kamiya et al. 2002], where this processed token stream is then converted into a suffix tree representation for the clone detection itself.

A drawback of this approach is that false positives can be identified, as the binding structure of the identifiers in a program is not necessarily respected when the particular names are hidden. This is avoided by *Dup* [Baker 1995], which uses a similar approach, but without knowing the particular scoping rules of the target language, false positives in the clone identification process are impossible to avoid.

Tree-based Techniques

These techniques search for similar subtrees in the AST, and so have the most structured inputs of the three mechanisms. The AST contains all the information required to do clone detection; variable names and literal values are discarded in the tree representation allowing more sophisticated methods for the detection of clones to be applied. In fact, it is possible to support search for more abstract clones, where common structures internal to the tree can be identified; these represent code which has a non-trivial common generalisation, where sub-trees – such as sub-expressions – are replaced by parameters to the abstraction.

In order to make an AST-based approach more efficient, the *CloneDR* [Baxter et al. 1998] compares nodes in its subtrees by characterization metrics based on a hash function. The source code of similar sub-trees is returned; CloneDR can check for consistent renaming. Our work builds upon some of the ideas outlined in Baxter et al's paper, although the work presented here uses a grouping algorithm rather than a hash function. Futhermore, Baxter et al. also suggest a mechanism for the removal of code clones with the help of macros, but, unlike the approach presented here, they did not carry out any clone removal.

DECKARD [Jiang et al. 2007] is another AST-based language independent clone detection tool; its algorithm is based on the computation of characteristic vectors that approximate structural information within ASTs. Once vectors are computed, they are clustered according to similarity vectors, yielding code clones. There are also some clone detection techniques based on program dependency graphs as demonstrated in [Komondoor and Horwitz 2001].

Hybrid Techniques

The clone detection in Wrangler, [Li and Thompson 2009, 2010], uses a hybrid technique. Clone *cadidates* – which include false positives – are identified using the token stream, rendered as a suffix tree; the AST is then used to check which of these candidates is a true clone.

Other 'facets'

Roy, Cordy and Koschke identify further 'facets' of difference, including the tool environment, the type of similarity (they identify four types), the language-specificity and so forth.

Many of the clone detection techniques discussed earlier are targeted at large legacy systems, and are not tightly integrated into any kind of programming environment. Language independent clone detection tools tend to have much lower precision since they do not – indeed they cannot – take into account static scoping rules of the particular language under analysis by making them less suitable for accurate clone refactoring.

3. The HaRe Clone Detector

There are two separate parts of the clone detection and removal: an *analysis* stage, which looks over a Haskell project and detects clone classes; and a *transformation* stage, which transforms identified clones from the clone classes in the first stage into calls to an appropriate abstraction.

The Clone Detector reports clones by identifying duplicated instances of expressions (and sub-expressions occurring with a "do" block) within a Haskell project. Each clone is reported with a start and end location (in the form of a line number) and clones that are instances, or near instances, of the same expression are grouped together to form clone classes. The analysis stage allows the user to specify the minimum number of tokens for the clones; these clone classes are then delivered in a report.

3.1 Preliminaries

The phrase "code clones" in general refers to a collection of program fragments that are identical or similar to each other. The code fragments can be similar if the program texts are similar or their functionalities are similar without being textually similar. Since semantic similarity is generally undecidable, we only consider clones that are textually similar, which can be compared via their AST representation. By similar code fragments we mean that the code fragments are instances of a well-formed expression, and can be replaced with a call to a generalisation. To give an example of this consider

```
f = (sum [1..100] + 23) + 24

g = foldr (+) 0 [1..10] + (3 + sum [2,3])
```

Both `f` and `g` are instances of the expression `x+y` operator. What makes these fragments code clones in the sense of this paper is that we can create a generalisation

```
add x y = x+y
```

and make `f` and `g` into calls to this generalisation, thus:

```
f = add (sum [1..100] + 23) 24

g = add (foldr (+) 0 [1..10]) (3 + sum [2,3])
```

Note in this example that to identify `f` and `g` as clones it is necessary to identify a common structure *internal* to their ASTs, rather than simply replacing terminal symbols such as names and numbers.

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 that 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.

3.2 AST-Level Clone Analysis

The HaRe clone analysis works over an AST representation of a Haskell program, only using the token stream to identify subexpressions that have a minimum token size (as specified by the user). By using an AST, it is possible to detect and transform clones whilst building upon the existing HaRe framework; source location information is still retained in the AST and whitespace and comments are removed. The clone analysis has five stages.

Inter-module analysis. The clone analysis first calculates the export relations of the module (the modules that import the current module either directly or indirectly). When a new project is initiated within HaRe, Programatica stores the export information in a local directory; it is possible to use this information within HaRe to calculate the modules exported or imported by the current module under refactoring. The export relations are calculated in order for the analysis to determine which modules need to be compared with each other. The clones for a single module project (say module A) are determined by comparing A against itself. If we are comparing a multiple module project that contains the modules A, B and C, we need to be able to compare each different pair of modules, that is $\mathrm{n}(\mathrm{n}+1)/2$ comparisons for an n module project.

It may be more efficient to concatenate the source files together into a separate module and then analyse only that module, rather than reviewing each file and comparing it with the others. However, we didn't consider this approach for a number of reasons:

- It is not straightforward in Haskell to concatenate all modules together due to the binding restrictions of the language. The clone detector would have a further process of renaming all the functions to avoid conflicts in namespaces between modules and in order to retain their identity.
- Programatica uses a project based system whereby all the modules of a Haskell project are parsed in turn, into separate syntax trees. It made sense to re-use the infrastructure of Programatica in this way in order to get a syntax tree for each module in the project. Using separate ASTs makes it easier to implemented the clone analysis as the binding structure relationship is retained throughout each module in the project.

Group. An initial pass over the AST groups together all expressions of the same syntactic form. Originally, before this grouping, the clone analysis took approximately 15 minutes to compute clones for the Huffman example from Thompson's Haskell text [Thompson 1999]. The grouping significantly diminished this time down to 23 seconds. By grouping expressions in this way, a lot of irrelevant analysis can be eliminated (i.e. by comparing an application with a section). For example, all function applications, identifiers, infix applications, and sections, etc. are grouped together. The grouping also takes into account subexpressions, so that the expression,

```
(convert (cd++[L]) t1) ++
(convert (cd++[R]) t2)
```

is grouped into the following:

Function Application. `convert (cd++[L]) t1`, `convert (cd++[R]) t2`

Infix Application. `(convert (cd++[L]) t1) ++ (convert (cd++[R]) t2)`, `cd++[L]`, `cd++[R]`

List. `[L]`, `[R]`

Identifier. `convert`, `cd`, `L`, `t1`, `convert`, `cd`, `R`, `t2`, `++`

Infix and prefix expressions are currently stored as separate entities in the Programatica syntax tree. It is expected that the clone detector will be extended so that infix application can be compared with prefix applications in the clone analysis.

Type and traversal. For each expression in the module, the type of the expression is determined and the relevant group of expressions is traversed for clone detection. A disadvantage is that some clones are missed as the clone analysis is only able to compare expressions with a similar syntactic form. It is not currently possible to compare infix expressions with function applications for example (and those would be not be clones in the sense used in this paper). However, the clone analysis does detect clones between expressions and parenthesized expressions, i.e. `e` and `(e)`. More information on the comparison of the AST is given in Section 3.3.

Sort and group. The clones that are found are sorted based on their token size and grouped together based on their size; clones that are contained within larger clones are removed.

Print. The clones are pretty printed to a report file. The report file includes the clone classes and their location and module information.

3.3 Abstract Syntax Tree Comparison

Two expressions are compared by traversing their structures. If the two structures contain different constructs at any point during the comparison then the analysis terminates and another expression is chosen by the traversal strategy.

- When comparing two identifiers `i1` and `i2`, `i1` and `i2` are identical if the following holds:
  - `i1` and `i2` both refer to the same top-level identifier. If `i1` and `i2` refer to different top-level identifiers, then they are not duplicate expressions. This eliminates cases such as comparing `1 + 2` with `1 - 2` (`(+)` and `(-)` are different top level identifiers). If `i1` and `i2` are both local variables then they will always match. It could be possible to abstract out top-level identifiers, as they can be passed in as higher-order parameters. However, we chose not to do this, because in most cases it is not a step required by the user. A literal may be compared with any other literal or a locally defined identifier, as literals can be passed in as parameters.
- Local identifiers may be identified with each other when making the comparison of sub-ASTs; local identifiers can be passed in as parameters.
- The analysis allows for the comparison of expressions with parentheses with expressions without parentheses. For example, suppose when comparing `(e1)` with `e2`: `(e1)` and `e2` are duplicates if `e1` compares with `e2` structurally, or either `e1` or

e2 are identifiers or literals. If the expression is being compared against a literal of a locally defined variable, then the expression can be passed in as a parameter. This is a consequence of the fact that the tree structure that the clone analysis uses to represent Haskell programs is in fact more detailed than an abstract syntax tree.

- For expressions that contain patterns, for example lambda abstractions and case alternatives, the expressions are the same if their pattern structures are also the same and their expressions are the same. For example, suppose when comparing the two lambda expressions:

```
e1 = \(x:xs) -> head xs
e2 = \(y:ys) -> head ys
```

If the structure of the patterns are both e1 and e2 equivalent, then the patterns in e2 are implicitly renamed by the analysis to match those in e1. This is done by renaming identifiers in a consistent way temporarily within the AST. Identifiers in the expression head ys are also renamed to match the renamed patterns. Therefore, e2 becomes:

```
e2 = \(x:xs) -> head xs
```

Which of course is equivalent to e1 and therefore duplicated. Lambda bindings are treated in exactly the same way as let/where bindings (in terms of their binding resolution). Consider an example where two lambda expressions would not match.

```
f1 = \(x:z:xs) -> x + length xs
f2 = \(x:z:xs) -> z + length xs
```

In the above example, the two expressions within the lambda bodies are different. The first lambda expression refers to x while the second refers to z. The process used here is exactly the same as the technique used for folding, as described in [Brown 2008] (and briefly in the following section).

Rather than performing an alpha renaming we first considered using de Bruijn indices instead to check equivalence between terms. However, de Bruijn indices do not solve the problem completely. Consider

```
\x.\y -> x + y
```

```
\x.\y.\z -> x + y
```

Using de Bruijn indices, in the first case we have, where we use the notation (n) for the variable bound n lambdas out:

```
\.\.(1) + (0)
```

And in the second

```
\.\.\.(2) + (1)
```

But this does not get us any closer to solving the problem, as the common pattern here is _+_, and this is not literally identified by the de Bruijn representation, necessitating some further analysis.

- The analysis looks for duplicated monadic sequences and duplicated monadic bindings or a combination of the two. For example, given the following two monadic blocks:

```
f = do
    x <- return 565; y <- k 56
    putStrLn (y ++ show x)
```

```
h = do
    a <- return 1111; b <- k 56
    putStrLn (b ++ show a)
    return 4
```

The clone analysis first determines that the two binding structures are identical up to changes in literals (indicated by the highlighting) by implicitly renaming the patterns in the second block so that they match the patterns in the first block.

```
f = do
    x <- return 565; y <- k 56
    putStrLn (y ++ show x)
```

```
h = do
    x <- return 1111; y <- k 56
    putStrLn (y ++ show x)
    return 4
```

After this renaming, the analysis then determines that the two expressions putStrLn (y ++ show x)
and putStrLn (y ++ show x) are also duplicates and therefore labels the binding generators x and y together with the qualifying statement putStrLn (y ++ show x) in both blocks as being clones.

4. Refactoring Support for Clone Removal

Already implemented in HaRe are the following refactorings [Brown 2008]:

Function folding. Folding replaces all sub expressions in a program, which are substitution instances of the right hand side of an identified equation, with a call to that equation, passing in expressions substituted as actual paramters. It is possible that folding in this sense could eliminate duplicate code by identify instances of common higher-order functions, such as foldr. For example, suppose that we have the following:

```
1 + (sum [1..10])
```

We can select the expression sum [1..10] and choose to fold against the definition of foldr. This would produce the expression:

```
1 + (foldr (+) 0 [1..10])
```

It is important to note that sum is an instance of foldr and is not a clone.

As-pattern folding. This refactoring replaces particular sub expressions occurring on the right hand side of an identified equation with calls to an as-pattern, provided that the sub-expression refers to a pattern binding that is defined in the same scope. For example, consider the following:

```
f (x:xs) = length (x:xs)
```

An as-pattern can be introduced for the first argument of f and substituted in the body of f:

```
f a@(x:xs) = length a
```

Merging. Merging creates a new tuple-returning definition; the constituent components of the tuple are extracted from a number of identified definitions introducing sharing. For example, consider the two definitions:

```
take :: Int -> [a] -> [a]
```

```
alphaMerge :: [(Char,Int)] -> [(Char,Int)]
           -> [(Char,Int)]
alphaMerge xs [] = xs
alphaMerge [] ys = ys
alphaMerge ((p,n):xs) ((q,m):ys)
```

(A)

```
  | (p==q) = (p,n+m) : alphaMerge xs ys
```

(B)

```
  | (p<q) = (p,n) : alphaMerge xs ((q,m):ys)
```

(C)

```
  | otherwise = (q,m) : alphaMerge ((p,n):xs) ys
```

Figure 1. Duplicated instances of the same expression

```
take 0 _ = []
take _ [] = []
take n (x:xs)
  | n > 0 = x : take (n-1) xs
take _ _ = error "take: negative argument"

drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop _ [] = []
drop n (x:xs)
  | n > 0 = drop (n-1) xs
drop _ _ error "drop: negative argument"
```

Merging allows the definitions of `take` and `drop` to be merged together, to form a new, recursive, definition:

```
splitAt 0 xs = ([],xs)
splitAt _ []  = ([],[])
splitAt n (x:xs)
  | n > 0 = (x:ys, zs)
    where
      (ys, xs) = splitAt (n-1) xs
splitAt _ _
    = (error "take: negative argument",
       error "drop: negative argument")
```

These refactorings, —as a side-effect— help to remove duplicated code, but rely on the user identifying particular problematic areas, and then invoking the refactorings accordingly. The clone detection tool is designed to perform the analysis automatically and to allow the user to choose whether or not to create an *abstraction* for particular members of a given clone class. Furthermore, the *undo* feature of HaRe allows the user to recover their original program if they change their mind.

The trivial examples in the remainder of this section are used to explain the concept of clone detection and extraction, and may result in more difficult to understand code. A larger example of clone detection and elimination is given in Section 5.

4.1 General Function Abstraction

This subsection introduces a new refactoring for HaRe in the form of function abstraction. This refactoring abstracts expressions within a clone class into a call to a new function. The refactoring takes into account instances of an expression together with exact duplicates of the expression. To show how this abstraction works, consider the code fragments in Figure 1. In this example, the HaRe

(A)

```
x+(y+z)
```

(B)

```
((x+y)+z)+w
```

(C)

```
(x+(y+z))+w
```

Figure 2. Duplicated instances showing different generalisations

clone detector has singled out the code fragments as duplicated instances of each other. In each code fragment *(A)*, *(B)* and *(C)* similar calls to the function `(:)` occur. The HaRe clone detector reports these as a clone class to the user, and asks whether or not to form an abstraction over this expression. A new function is created automatically with the expression indicated in the clone class as the function body; any formal parameters are added to the function abstraction; for example, if the clone differs in a literal at each instance this is made into a parameter. The expression instances are then replaced with a function call. The following gives an example of the abstraction that is created for the instances highlighted in Figure 1:

```
abs_1 p_1 p_2 p_3 p_4
   = (p_1, p_2) : (alphaMerge p_3 p_4)
```

Figure 3 shows the transformed program after the abstraction is created and the expressions are replaced with calls. It is interesting, however, that —in general— if there is a common generalisation for three or more sub-expressions, then the common generalisation for two of them compared to three or more may be different. Consider the code blocks in Figure 2. The common generalisation for (A), (B) and (C) is:

```
abs_1 p_1 p_2 = p_1 + p_2
```

Selecting only (B) and (C) however would produce the following abstraction:

```
abs_1 p_1 p_2 p_3 = (p_1 + p_2) + p_3
```

Generalising the expressions in this way has the advantage of removing duplicated code from within the program and encourages code reuse and maintenance. In Figure 3, the programmer now only needs to worry about maintaining one call to `(:)`, while before there were three calls to `(:)` to maintain. It is interesting to note, however, that in the example above the introduction of the abstraction does not shorten the expressions. It does, however, abstract away a common sub-expression, allowing the expression to be more easily maintained at a later stage in the program development cycle. It is important to note that HaRe has a renaming refactoring [Li 2006], that allows more appropriate names to be introduced for the abstraction and its arguments.

4.2 Step-by-step Clone Detection

Clone detection is designed to be fully automatic. Clone classes that are detected are presented to the user in the form of a report, where the user can then step through the instances, deciding whether or not they should be abstracted; this can be done by tabbing through the instances in an editor.

To put this into context, consider the code fragments in Figure 5. The clone detector has detected two clone classes: the expressions on lines 3 and 6 are identified to be instances of the same expression. The clone detector shows this by delivering a report to the user, stating that the sub-expression on line 3 and the sub-expression on line 6 are both instances of a non-trivial generalisa-

```
module A where
import B

emitJump p j i =
  emitByte p (j) >|>
  emitByte p (show l) >|>
  emitByte p (show h)
 where
   (h,l) = i `divMod` 256
```

```
module B where

emitOp12 p op i =
   case (-i) `divmod` 256 of
    (0,l) -> emitByte p (op ++ "_N1") >|>
             emitByte p (show l)
    (h,l) -> emitByte p (op ++ "_N2") >|>
             emitByte p (show h)

emitByte x y = ...
```

Figure 4. A multiple-module project demonstrating clone instances

(A)

```
abs_1 p_1 p_2 p_3 p_4
 = (p_1, p_2) : (alphaMerge p_3 p_4)

alphaMerge :: [(Char,Int)] -> [(Char,Int)]
           -> [(Char,Int)]
alphaMerge xs [] = xs
alphaMerge [] ys = ys
alphaMerge ((p,n):xs) ((q,m):ys)
  | (p==q) = abs_1 p (n + m) xs ys
```

(B)

```
  | (p<q) = abs_1 p n xs ((q, m) : ys)
```

(C)

```
  | otherwise = abs_1 q m ((p, n) : xs) ys
```

Figure 3. Duplicated instances replaced with a call to `abs_1`

(A)

```
1 showTreeIndent :: Int -> Tree -> String
2 showTreeIndent m (Leaf c n)
3  = spaces m ++ show c ++ " " ++ show n
4 showTreeIndent m (Node n t1 t2)
5  = showTreeIndent (m+4) t1 ++
6      spaces m ++ "[" ++ show n ++ "]"
7              ++ showTreeIndent (m+4) t2
```

Figure 5. Two code fragments showing code instances for step-by-step removal

```
/home/cmb/huffman/CodeTable.hs ((46,5),(46,44)):
                         >spaces m ++ show c ++
                          " " ++ show n ++ "\\n"<

Would you like to extract this expression (Y/N)?
```

Figure 6. Results of running the extraction

tion; this process is known as *anti-unification* [Bulychev and Minea 2008] and is discussed in more detail in Section 4.4. For each of the instance matches, the clone detector asks the user whether or not they would like to replace the sub-expression with a call to an abstraction. Figure 6 shows example of the interaction (the numbers in parentheses indicate the start and end positions in the source file using row and column numbers).

For each clone candidate, HaRe displays the start and end locations of the clone instance. This report is produced automatically for each of the instances in turn; the user only has to answer "yes" or "no" for each question. The user may quit at any time by answering with "Q"; this has the affect of only extracting expressions that the user has answered "yes" to up to the point of quitting; all further expressions are left unchanged. If the user has answered "yes" to a question, HaRe also asks the user for the name of the abstraction and proceeds to transform the code to include the abstraction and the calls. This allows for cases where the user may not want to replace all instances of a particular clone, but only some of them; this behaviour is particularly useful if the clone detector reports clones classes with a large number of clones (the user can bail out without having to go through each candidate in turn). It is worth noting that it may be useful to implement a feature so that the user can reply "A" as shorthand for replacing all clones, as long as the user is sure they want to do that.

4.3 Choosing the Location of the Abstracted Function

In Haskell, there are two cases to consider when placing the extracted function:

- The first is in a single-module project where the extracted function can be placed anywhere at the top-level scope of the module.
- The second is a multiple-module project, where insertion of the extracted function in one of the constituent modules may introduce circular inclusions.

Consider the code blocks in Figure 4; this figure demonstrates a multiple-module project where we have, on the left hand side, a module that imports the module B, which is shown on the right hand side. The highlighted expressions in the figure show that the HaRe clone detector has discovered some cloned expressions. If the user has decided they would like to replace these sub-expressions with an function application the question is where does HaRe place the abstracted function for these sub-expressions?

In Figure 7, HaRe has placed the abstraction in the module that contains the instance of the first duplicated sub-expression that is selected for extraction. The problem here is that a cyclic dependency has been introduced: module A imports module B and B imports module A. Cyclic inclusions must be avoided during the transformation due to the fact that transparent compilation of mutually-recursive modules is not supported by current Haskell compil-

```
module A where
import B

divMod256 i = i `divMod` 256
emitJump p j i =
  emitByte p (j) >|>
  emitByte p (show l) >|>
  emitByte p (show h)
 where
   (h,l) = divMod256 i
```

```
module B where
import A (divMod256)

emitOp12 p op i =
   case divMod256 (-i) of
     (0,l) -> emitByte p (op ++ "_N1") >|>
              emitByte p (show l)
     (h,l) -> emitByte p (op ++ "_N2") >|>
              emitByte p (show h)

emitByte x y = ...
```

Figure 7. A multiple-module project demonstrating circular inclusions

ers/interpreters; even though mutually-recursive modules are a part of the Haskell 98 standard.

A possible solution to this problem is to put all the abstractions from a clone detection process into a separate module, but this then risks having problems when the abstractions depend on other functions themselves. The HaRe clone detector solves this problem by performing an analysis over the project under clone detection. If it is safe to place the abstraction into the module where the first highlighted expression occurs within the clone class, then the replacement is performed. Otherwise, the abstraction is placed in the first available module that does not introduce circular inclusions. This module is calculated by Programatica, which stores a list of available modules, and the modules that are used in the clone elimination are extracted into a separate list. This list is then traversed, until a module is found where it is possible to introduce the abstraction without introducing a circular inclusion. If it is impossible to avoid a circular inclusion, then the clone detection reports an error to the user. In the case that expressions over multiple-modules are transformed into calls to the abstraction, the import relations of the modules containing those expressions are modified to import the module containing the abstraction. Suppose the module containing the abstraction is A and the module importing the abstracted module is B, the design issues for placing the abstraction are as follows:

- If module A has an explicit export list, the abstraction name is added to the export list.
- If module B has an explicit import list for module A, then the abstraction name is added to the explicit list of imports.

Otherwise an `import` statement is added to module B so that it imports module `A` with the abstraction name added to the list of imports for module A.

4.4 Calculating the Abstraction

The abstraction is calculated only when the transformation process can determine which sub-expressions can be passed in as an argument. This is done by comparing the identified expressions for transformation from the clone class. If, during the comparison, the sub-expressions are different, then they can be passed into the abstraction as a parameter. If the sub-expressions are the same (either as literals or identifiers referring to the same top-level identifier) then they do not need to be passed in as arguments. Consider the following three clones:

```
(p,n+m) : alphaMerge xs ys
(p,n) : alphaMerge xs ((q,m):ys)
(q,m) : alphaMerge ((p,n):xs) ys
```

The transformation stage determines that from the three clones, the expressions `(:)` and `alphaMerge` occur in the same place in each expression. All other sub-expressions must be passed in as an argument. This process is called "anti-unification" as it is finding the least general generalisation of a set of expressions [Bulychev and Minea 2008].

4.5 Abstracting Monadic Sequences

There are two cases to consider when abstracting duplicated monadic sequences. The first is where the abstraction contains pattern bindings that will be used in the remainder of the function containing the clone instance. The second is where the pattern bindings are not required in the remainder of the function containing the clone instance. If any pattern bindings are needed by the function then the values that they bind must be returned by the abstraction, and a new pattern binding is created to bind the variables to the appropriate values returned by the abstraction. Consider, for example (where the highlighted expressions show the clones):

```
f = do
    x <- return 565; y <- k 56
    putStrLn (y ++ show x)

h = do
    a <- return 1111; b <- k 56
    putStrLn (b ++ show b)
    return 4
```

A new abstraction is created with a `return` statement, returning the values of the pattern bindings as a tuple in the abstraction:

```
abs1 p_1
  = do
      return (p_1,k 56)
```

The clone instances are then replaced with pattern bindings, so that the patterns bound in the abstraction can be further used in the program:

```
f = do
    (x,y) <- abs1 565
    putStrLn (y ++ show x)

h = do
    (a,b) <- abs1 1111
    putStrLn (b ++ show b)
    return 4
```

If the abstraction contains no pattern bindings, or the pattern bindings in the abstraction are not needed outside the abstraction to be created, then no `return` statement is added to the abstraction. If only a subset of the variables are needed, then only these values are returned. Consider that HaRe has detected the following highlighted clones:

```
f = do
      putStrLn (show 56 ++ " " ++ show 42)

h = do
      putStrLn (show 42 ++ " " ++ show 56)
    return 4
```

The following abstraction is created:

```
abs p_1 p_2 = putStrLn (show p_1 ++ " "
                        ++ show p_2)
```

The clone instances are then replaced with calls to the abstraction.

4.6 An Issue with Generalisation

There is an issue to consider when dealing with abstracting over common sub-expressions (generalisation). Consider the following piece of code:

```
f x y = if (\z -> z) x then y else (\z -> z) (y+1)
```

As it can be seen from the code, the common code to be abstracted is a lambda expression:

```
(\z -> z)
```

but it is used at two different monotypes within the body of `f`. If we abstract over the lambda expression we force it to have the type `forall a. a -> a` which cannot be unified. If we introduce a new definition, `g`, as follows:

```
g x y = 1+ (if (id.id) x then y else (id.id) (y+1))
```

We see the common generalisation of `f` and `g` as

```
gen x y h = if h x then y else h (y+1)
```

This results in both `f` and `g` being transformed to call the new abstraction, `gen`:

```
f x y = gen x y (\z -> z)

g x y = 1 + gen x y (id.id)
```

This now forces the argument of `h` within `gen` to be polymorphic, which obviously gives a type error. This is a problem with lambda lifting [Johnsson 1985] in general where we introduce functions (or lambdas) to abstract over a polymorphic entity.

This problem with generalisation could be overcome if the clone detection was extended to take into account the fact that the types of the clones chosen for abstraction within a class must be unifiable. If this was the case, the above example could not occur. A solution to this problem is to introduce an existentially qualified type. In our example, the type checker has inferred the type of `gen`:

```
gen :: Num a => Bool -> a -> ( b -> b )-> a
```

Implicitly the type checkers infers that `a` and `b` are qualified as existential types:

```
gen :: forall a. forall b. Num a => Bool
       -> a -> ( b -> b )-> a
```

The problem is because the `forall a . forall b` is on the left most side, we need to introduce rank-2 polymorphism in all cases, so that `b` can be unied with both the types `Bool` and `Num a => a`:

```
gen :: forall a. Num a => Bool
       -> a -> (forall b . ( b -> b) )-> a
```

Many of the refactorings in HaRe already have support for type analysis, and, although this is an extreme case of generalisation, it is expected that work to extend the clone analysis to take types into account will be an aspect of future work.

5. Case Study

To evaluate the tool we have applied it to a large-scale application written in Haskell 98. We were unable to to perform clone elimination on HaRe due to the fact that HaRe is written in non-Haskell 98 code, and yet is currently only capable of refactoring Haskell 98 programs. Instead, the clone detection was performed over a different test case.

The experiment was to run the clone analysis over `nhc` [Röjemo 1995]: a Haskell 98 compiler system. For the purposes of the experiment, the clone analysis was configured so that there was no limit in the size of the clone classes and the minimum number of tokens appearing in a cloned expression is 5. Due to performance restrictions that are discussed in Section 5.1, the clone analysis was only run over the `Main` module to show the clone analysis as a proof of concept.

The first step was to extract some of the clones reported into abstractions. Figure 8 shows an extract of a clone class that was considered for extraction. The expression:

```
mixLine (map show
        (listAT (getSymTab state)))
```

Within HaRe the *extract expression* is selected from the HaRe drop-down menu. After the clone detection process, the clone detector steps through each clone candidate. It seems appropriate for all expressions to be extracted. HaRe then creates an abstraction (here, `state` is a local variable that must be passed in as a parameter):

```
abs_1 p_1
    = mixLine (map show (listAT
              (getSymTab p_1)))
```

And replaces the expressions in the clone class with:

```
abs_1 state
```

The name is chosen by the refactoring automatically; however the choice of the uninformative name can easily be fixed by a renaming. The same is performed for the clone expressions identified in the two former clone classes of Figure 8. The expressions occuring in the remainder of the report (not shown in the figure) are selected in turn, and extracted into calls to the following abstractions. For example, the expressions of the form:

```
mixLine (map show (listAT
        (getRenameTableIS impState)))
```

are converted into calls as in the following

```
abs_2 impState
```

Likewise, expressions of the form:

```
mixLine (map show (listAT
        (getSymTabIS impState)))
```

are converted into calls as in the following

```
abs_3 impState
```

The following abstractions are introduced at the top of the module:

```
abs_2 p_1
  = mixLine (map show (listAT
            (getSymTabIS p_1)))

abs_3 p_1
  = mixLine (map show (listAT
            (getRenameTableIS p_1)))
```

It now makes sense to convert `abs_3` into a call to `abs_2`. This can be done manually using some of the current refactorings already implemented in HaRe: the definition of `abs_2` is *generalised*

```
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((191,10),(191,59)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((218,11),(218,60)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((232,13),(232,62)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((280,11),(280,60)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((312,12),(312,61)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((333,12),(333,61)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((344,12),(344,61)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((372,7),(372,56)):
   >mixLine (map show (listAT (getSymTab state)))<
/home/cmb21/caseStudy/noHier/stage3/src/compiler98/MainNew.hs  ((390,7),(390,56)):
   >mixLine (map show (listAT (getSymTab state)))<
9 occurrences.
```

Figure 8. An extract from the clone report for nhc

so that `getSymTabIS` is made a parameter (this also updates all calls to `abs_2 localState` with `abs_2 getSymTabIS localState`). A *fold* is then performed against the (generalised) definition:

```
abs_2 p_0 p_1
= mixLine (map show (listAT (p_0 p_1)))
```

Transforming `abs_3` into a call for `abs_2`:

```
abs_3 p_1
   = abs_2 getRenameTableIS p_1
```

An interesting observation in performing clone analysis over `nhc` is that few of the clones appear to be true duplicates, or duplicates that are worth extracting. The main problem is that the analyser cannot distinguish between useful and non-useful clone entities. Indeed, the analysis also compares expressions of the form `f e1 e2` with `f e1 e2 e3`.

The reason for this is mainly due to the fact that the clone analysis is tree-based. `f e1 e2` is actually represented as `(f e1) e2` and, if `e2` is a variable that is not declared at the top-level, it will consequently match against anything.

Monadic clone detection could not be performed in this case study. `nhc` only has a very small amount of monadic code which is not appropriate for clone extraction. A better example may be to apply the clone analysis over a monadic parser example, as long as the example uses the `do` notation.

5.1 Performance

In order to test the true performance of the clone analysis, the HaRe clone detector was ran over a number of Haskell 98 programs varying in program size. These programs are found as examples used in Thompson's text [Thompson 1999] and as test applications for `nhc` [Röjemo 1995]. The performance study was executed on a Linux machine running Fedora 11 with Intel(R) 2.66GHz Quad Core Processor and 4GB RAM.

Table 1 shows the results of applying the HaRe clone detector to the Haskell programs mentioned above. The first column shows the program's size by the number of tokens over the whole program. It seems to make more sense to show size rather than the number of lines of program code in each example, as the number of lines of code is not a good indication as to how many expressions need to be compared.

The number of clones and the time taken in seconds is shown in Table 1 for the clone detection of expressions ≥ 3, ≥ 6 and ≥ 10 respectively. The `nhc Main` module is not included in the results here. The clone detector is currently designed to run over all modules in the program, re-implementing it to work over only the `Main` module would give inaccurate and misleading results. However, computing clones for the whole of `nhc` is known to take a considerable amount of time under the current algorithm.

For all applications, the clone detector was able to finish the detection in a reasonable time. The running time, obviously, seems to be affected by the size of the program, and the number of candidates grouped as clones. This suggests that the algorithm for detecting clones is $O(n^2)$ in the worst case. Interestingly, the clone detector reported more clone candidates when comparing all expressions with a token size ≥ 3 and many fewer candidates for expressions with a token size ≥ 10. This suggests that the clone detector gives more accurate results without giving false positives when expressions with a token size ≥ 6; this is also shown in Table 1.

As most of the clone detection is done by comparing subtrees, the performance could be greatly enhanced by making the comparison parallelizable. Another approach to improving performance would be to use suffix trees to do an initial comparison, and then use the more costly AST to check clone candidates, as in Wrangler [Li and Thompson 2009, 2010].

6. Conclusions and Future Work

In this paper, we have presented a clone detection and elimination system that is embedded within the HaRe framework. The clone detector makes use of an AST-based approach to detect clones, and allows the user to select which clones to eliminate. The benefit of using an AST based approach is that it tends to lead to more accurate results over token-based methods. The usefulness of the tool was demonstrated on a real-world case study.

Two stages of clone removal were presented. The first stage as discussed in Section 3 was an automatic clone detection system, where expressions in a multiple-module project are compared against each other. Clones are reported to a text file. The second stage, as discussed in Section 4, allows the user to highlight a particular instance of a clone. HaRe then asks the user whether or not they would like to replace the clones with a call to an abstraction in the identified class. The location of the abstraction is discussed

Name	Density	Clones ≥ 3	Time ≥ 3	Clones ≥ 6	Time ≥ 6	Clones ≥ 10	Time ≥ 10
Simulation	111	8	40	4	16	0	8
Pictures	174	5	4	2	2	2	1
PhoneChess	212	16	23	6	7	2	3
Huffman	738	23	95	10	26	2	13
Minesweeper	910	28	156	28	56	4	17
Minimax	968	46	213	21	79	0	21
Fish	1157	142	740	8	114	6	49
Calculator	1168	18	40	2	21	2	15
Grep	1358	34	210	5	98	0	52
Expert	2349	82	699	82	20	2	39
RegEx	2858	101	1542	35	254	4	97
Compress	3212	23	103	10	27	0	14
PolyGP	5279	355	16406	187	5562	17	2164

Table 1. Clone Detection Results

in Section 4.3. Monadic clone elimination was discussed in Section 4.5. Finally, the clone detection and elimination was used to eliminate duplicate code from `nhc` in Section 5.

In the future it is expected work to continue on the tool in a number of directions:

- We would like to make use of interactive visualisation techniques to improve the presentation of the clone results.
- We would like to extend the tool to be used to identify the redefinition of library functions. This could be particular useful for newcomers to Haskell to learn library functions with greater ease.
- We expect to look at a hybrid approach comparable to the Wrangler approach [Li and Thompson 2009, 2010] work, as this may make the tool scalable to larger Haskell projects.
- We would like to apply the tool to a substantial monadic project, allowing us to evaluate the usefulness of the monadic clone detection and elimination process.
- We expect to extend the clone analysis to take type information into account. Using types in the analysis would help to avoid problems with generalisation (as discussed in Section 4.6) and could build upon the existing type analysis infrastructure of HaRe [Brown 2008].
- Finally, fully automated clone removal could be supported by *scripted* refactorings, based on the clone results.

We would like to thank Dave Harrison for his editorial advice, and the anonymous referees for their very valuable comments.

References

B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*. IEEE Computer Society.

Ira D. Baxter, *et. al.* Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, 1998. IEEE Computer Society.

Christopher Brown. *Tool Support for Refactoring Haskell Programs*. PhD thesis, School of Computing, University of Kent, UK, 2008.

Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Third International Workshop on Detection of Software Clones*, 2008.

Debra Cameron, James Elliott, and Marc Loy. *Learning GNU Emacs*. O'Reilly, 2004.

Andy Chou, *et. al.* An Empirical Study of Operating Systems Errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001. ACM.

Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, USA, 1999.

Thomas Hallgren. Haskell Tools from the Programatica Project. In *ACM SIGPLAN workshop on Haskell*, 2003. ACM Press.

Lingxiao Jiang, *et. al.* Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, 2007. IEEE Computer Society.

J. Howard Johnson. Identifying Redundancy in Source Code Using Fingerprints. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1993.

Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. Springer-Verlag, 1985.

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):2002.

Raghavan Komondoor and Susan Horwitz. Tool Demonstration: Finding Duplicated Code Using Program Dependences. In *ESOP*, volume 2028 of *Lecture Notes in Computer Science*, Springer, 2001.

Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, School of Computing, University of Kent, UK, 2006.

Huiqing Li and Simon Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In *PEPM*, ACM, 2009.

Huiqing Li and Simon Thompson. Similar Code Detection and Elimination for Erlang Programs. In *PADL*, ACM, 2010 (to appear).

Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In *ACM SIGPLAN 2003 Haskell Workshop*, ACM, 2003.

Zhenmin Li, Shan Lu, and Suvda Myagmar. Cp-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, 32(3):2006.

Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):1976.

Akito Monden, *et. al.* Software quality analysis by code clones in industrial legacy software. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, 2002. IEEE Computer Society.

Steve Oualine. *Vim (Vi Improved)*. Sams, 2001.

Simon Peyton Jones and Kevin Hammond. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.

Niklas Röjemo. Highlights from nhc—a space-efficient haskell compiler. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, 1995. ACM.

Chanchal K. Roy, James R. Cordy and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7), 2009.

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.