Clone Detection via Structural Abstraction

William S. Evans* will@cs.ubc.ca
Christopher W. Fraser† cwfraser@gmail.com
Fei Ma‡ Fei.Ma@microsoft.com

Abstract

This paper describes the design, implementation, and application of a new algorithm to detect cloned code. It operates on the abstract syntax trees formed by many compilers as an intermediate representation. It extends prior work by identifying clones even when arbitrary subtrees have been changed. On a 440,000-line code corpus, 20-50% of the clones it detected were missed by previous methods. The method also identifies cloning in declarations, so it is somewhat more general than conventional procedural abstraction.

1 Introduction

Duplicated code arises in software for many reasons: copy-paste programming, common language constructs, and accidental duplication of functionality are some common ones. Code duplication or *cloning* (especially copy-paste programming) makes it harder to maintain, update, or otherwise change the program. For example, when an error is identified in one copy, then the programmer must find all of the other copies and make parallel changes. Also duplicate code can make understanding a system more difficult since the crucial difference in two nearly-identical copies may be obscured. On the other hand, cloning is easier than creating a procedure to perform both the original and a new task, and it can be less error-prone (though many errors result from incorrectly or incompletely modifying copies). Since cloned code appears to be a fact of life, identifying it—for maintenance, program understanding, or code modification (e.g. refactoring [13] or program compaction)—is an important part of software development.

There is much prior work in this area, operating on source code [2, 3, 16, 21], abstract syntax or parse trees [5, 20, 15], program dependence graphs [18], bytecode [4] and assembly code [9, 10, 25, 11]. The methods also use various matching techniques: suffix trees [11, 2, 3, 16, 20], hashing [5, 9, 10, 25], subsequence mining [21], program slicing [18], and feature vectors [19, 23, 15].

Clone detectors offer a range of outputs. Some mainly flag the clones in a graphical output, such as a dot-plot [8]. This strategy suits users who reject automatic changes to their source code. Other clone detectors create a revised source code, which the user is presumably free to modify or decline [18]. Still others automatically perform procedural abstraction [9, 10, 25, 11], which replaces the clones with a procedure and calls. This fully automatic process particularly suits clone detectors that operate on assembly or object code, since the programmer generally does not inspect this code and is thus unlikely to reject changes.

Most clone detectors find not only identical fragments of code but also copies with some differences. These slightly different copies could, in theory, be abstracted into a single procedure taking the differences as parameters. However, most previous methods permit only what we call *lexical abstraction*; that is, a process akin to a compiler's lexical analyzer identifies the elements that can become parameters to the abstracted procedure. Typically, the process treats identifiers and numbers for source code or register numbers and literals for assembly code as equivalent; or, alternatively, it replaces them with a canonical form (a "wildcard") in order to detect similar clones. For example, it treats the source codes `i=j+1` and `p=q+4` as if they were identical. In this simple form, lexical abstraction can generate many false positives. A more precise version, parameterized pattern matching [3], eliminates many of these false positives by requiring a one-for-one correspondence between parallel parameters.

Still, some clones detected using these methods could not be abstracted into procedures because they do not obey the grammatical structure of the program. A clone consisting of the end of one procedure and the beginning of another is not easily abstracted, especially at the source-code level, and perhaps should not be recognized as a clone. Searching for clones within the program's abstract syntax tree (AST), rather than its text, avoids these ungrammatical clones. This is the main motivation for most clone detection approaches

*University of British Columbia, Department of Computer Science, Vancouver, BC, Canada. Supported by NSERC Discovery Grant.

†Some of this author's work on this paper was supported by Microsoft Research.

‡Microsoft, Redmond, WA, USA. Some of this work appeared in this author's MS thesis.

Dagstuhl Seminar Proceedings 08261
Structure-Based Compression of Complex Massive Data
http://drops.dagstuhl.de/opus/volltexte/2008/1678

using ASTs.

Clone detection in ASTs suggests a natural generalization of lexical abstraction in which parameters represent subtrees of an AST. Subtrees of an AST may correspond to lexical constructs (identifiers or numbers) but they may also correspond to more general constructs that capture more complicated program structures. Thus, we call this generalization *structural abstraction*.

There is some prior work on clone detection in ASTs, though not fully general structural abstraction as defined above. One method uses a subset of the AST features as part of a feature vector describing code fragments and searches for clones using feature vector clustering [19]. Another method [5] finds clones in an AST but allows only lexical abstraction. A third method linearizes the AST and looks for clones, using standard techniques, in the resulting sequence of AST nodes [20]. A fourth clusters feature vectors that summarize parse trees [15]. We discuss these and other approaches in more detail in Section 6.

This paper presents the results of applying general structural abstraction to ASTs. Our work has no special treatment for identifiers, literals, lists, or any other language feature. It bases parameterization only on the abstract syntax tree. It abstracts identifiers, literals, lists, and more, but it does so simply by abstracting subtrees of an AST.

The objective of this work is to determine if full structural abstraction on ASTs is affordable and if it improves significantly on lexical abstraction. Structural abstraction seems inherently more costly, and there is no *prima facie* evidence that it finds more or better clones.

To answer these questions, we designed and built a clone detector based on structural abstraction and ran it on over 425,250 lines of Java source and over 16,000 lines of C# source. We both tabulated the results automatically and evaluated selections manually. In these tests, structural abstraction improved significantly on lexical abstraction: 20-50% of the clones we found elude lexical abstraction.

2 Algorithm

Our structural abstraction prototype is called Asta. Asta accepts a single AST represented as an XML string. It has been used with ASTs created by JavaML from Java code [1] and with ASTs created by the C# compiler lcsc [14]. A custom back end for JavaML and lcsc emits each module as a single AST. A simple tool combines multiple ASTs into a single XML string to run Asta across multiple modules.

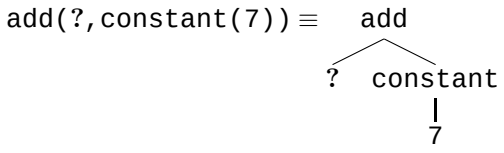
The ASTs are easily pretty-printed to reconstruct a source program that is very similar to the original input. The ASTs are also annotated with pointers to the associated source code. There are thus two different ways to present AST clones to the programmer in a recognizable form.

To explain Asta, we use common graph theoretic terminology and notation. For example, $V(G)$ and $E(G)$ denote the nodes and edges of a graph G. A *subtree* is any connected subgraph of a tree. A subtree of a rooted tree is also rooted and its root is the node that is closest to the root in the original tree. An *ancestor* of a node in a rooted tree is a node on the path from the root to that node. If node u is an ancestor of node v then v is a *descendant* of node u. A *full subtree* of a rooted tree T is subtree of T containing a node of T and all of its descendents in T.

A *pattern* is a labeled, rooted tree some of whose leaves may be labeled with the special wildcard label, `?`. Leaves with this label are called *holes*. A pattern P matches a labeled, rooted tree T if there exists a function $f : V(P) \rightarrow V(T)$ such that $f(\text{root}(P)) = \text{root}(T)$, $(u, v) \in E(P)$ if and only if $(f(u), f(v)) \in E(T)$, and for all $v \in V(P)$, either (1) $\text{label}(v) = \text{label}(f(v))$, and v and $f(v)$ have the same number of children, or (2) $\text{label}(v) =$ `?`. In our case, T is a full subtree of an abstract syntax tree and the pattern P represents a macro, possibly taking arguments. Each hole v in P represents a formal parameter that is filled by the computation represented by the full subtree of T rooted at $f(v)$.

An *occurrence* of a pattern P in a labeled, rooted tree S is a subtree of S that P matches. Multiple occurrences of a single pattern P in an abstract syntax tree represent cloned code. A *clone* is a pattern with more than one occurrence.

In what follows, trees and patterns appear in a functional, fully-parenthesized prefix form. For example,



denotes a pattern with one hole. When a pattern is used to form a procedure, holes correspond to formal parameters in the definition and to actual arguments at invocations. Holes must replace a full subtree. For example,

```
?(local(a),formal(b))
```

is not a valid pattern because the hole replaces an operator but not the full subtree labeled with that operator. This restriction suits conventional programming languages, which generally do not support abstraction of operators. Languages with higher order functions do support such abstraction, so Asta would ideally be extended to offer operator wildcards if it were used with ASTs from such languages. Algorithms and experimental results for the extended version of Asta can be found in [22].

2.1 Pattern generation

Asta produces a series of patterns that represent cloned code in a given abstract syntax tree S. It first generates a set of candidate patterns that occur at least twice in S and have at most H holes (H is an input to Asta.) It then decides which of these patterns to output and in what order.

Candidate generation starts by creating a set of simple patterns. Given an integer parameter D, Asta generates, for each node v in S, at most D patterns called *caps*. The d-cap ($1 \leq d \leq D$) for v is the pattern obtained by taking the depth d subtree rooted at v and adding holes in place of all the children of nodes at depth d. If the subtree rooted at v has no nodes at depth d (i.e. the subtree has depth less than d) then node v has no d-cap. Asta also generates a pattern called the *full cap* for v, which is the full subtree rooted at v. For example, if $D = 2$ and the subtree rooted at v is:

```
add(local(a),sub(local(b),formal(c)))
```

then Asta generates the 1-cap `add(?,?)` and the 2-cap `add(local(?),sub(?,?))` as well as the full cap `add(local(a),sub(local(b),formal(c)))`.

The set of all caps for all nodes in S forms the initial set, Π, of candidate patterns.

Asta finds the occurrences of every cap by building an associative array called the *clone table*, indexed by pattern. Each entry of the clone table is a list of occurrences of the pattern in S. Asta removes from Π any cap that occurs only once.

Karp, Miller, and Rosenberg [17] present a theoretical treatment of the problem of finding repeated patterns in trees (as well as strings and arrays). Their problem 1 is identical to the problem of finding all d-caps: "Find all depth d substructures of S which occur at least twice in S (possibly overlapping), and find the position in S of each such repeated substructure." Unfortunately, they present algorithms that solve problem 1 only for strings and arrays. Their tree algorithms are designed to find the occurrences of a given subtree in S (a problem that we solve using an associative array, i.e., hashing).

After creating the set, Π, of repeated caps, Asta performs the closure of the *pattern improvement* operation on the set. Pattern improvement creates a new pattern by replacing or "specializing" the holes in an existing pattern. Given a pattern P, pattern improvement produces a new pattern Q by replacing every hole v in P with a pattern $F(v)$[1] such that (i) $F(v)$ has at most one hole (thus, Q has at most the same number of holes as P), and (ii) Q occurs wherever P occurs (i.e. $F(v)$ matches every subtree, from every occurrence of P, that fills hole v). It is possible that for some holes v, the only pattern $F(v)$ that matches all the subtrees is a hole. In this case, no specialization occurs for hole v.

In order to perform pattern improvement somewhat efficiently, we store with each node u in S a list of patterns that match the subtree rooted at u. The list is ordered by the number of nodes in the pattern in decreasing order. Given a pattern P to improve and a hole v in P, Asta finds an arbitrary occurrence of P (with matching function f) in S and finds the list of patterns stored with the node $f(v)$. Asta considers the patterns in this list, in order, as candidates for $F(v)$. Any candidate with more than one hole is rejected (to satisfy condition (i)). In order to satisfy condition (ii), a candidate pattern must match the subtree rooted at $f(v)$ for all matching functions f associated with occurrences of P. Another way of saying this is that every node $f(v)$ (over all matching functions f from occurrences of P) must be the root of an occurrence of the candidate pattern. Thus Asta looks up the candidate pattern in the clone table and checks that each $f(v)$ is the root of an occurrence in that table entry. (We actually store this list of occurrences as an associative array indexed by the root of the occurrence, so the check is quite efficient.)

Asta repeats the pattern improvement operation on every pattern in Π, adding any newly created patterns to Π, until no new patterns are created.

Pattern improvement is a conservative operation. It only creates a more specialized pattern if it occurs in the same places as the original pattern. Some patterns can't be specialized without reducing the number of occurrences. We may still want to specialize these patterns because our focus is on finding large patterns that occur at least twice. Asta performs a greedy version of pattern specialization, called *best-pair specialization*, that attempts to produce large patterns that occur at least twice. It does this by performing pattern improvement but requires only that the specialization preserves two of the occurrences of the original pattern.

For each pair of occurrences, T_i and T_j ($1 \leq i < j \leq r$) of a given pattern P with r occurrences, Asta produces a new pattern Q_{ij} that is identical to P except that every hole v in P is replaced by a pattern $F_{ij}(v)$ such that (a) $F_{ij}(v)$ has at most one hole, and (b) Q_{ij} matches T_i and T_j. The largest Q_{ij} (over $1 \leq i < j \leq r$) is the best-pair specialization of P. Asta creates the best-pair specialization for every pattern P in the set of patterns, Π, and adds those patterns to Π. It then computes, again, the closure of Π using the pattern improvement operation.

As the final step in candidate generation, Asta removes from Π all *dominated patterns*. A pattern is dominated if it was improved by the pattern improvement operation.

[1] The notation emphasizes the fact that each hole may be filled with a different pattern.

2.2 Thinning, ranking, and reporting

Asta finds many candidate clones, sometimes too many, so the candidates are thinned and ranked before output. Asta supports a wide range of options for thinning and ranking.

Thinning uses simple command-line options that give thresholds for number of nodes and number of holes. All results in this paper omit clones under ten nodes or over five holes. The ASTs average approximately 14 nodes per line, so some sub-line clones are reported. Though sub-line clones are often too small to warrant refactoring, they can yield substantial savings when abstracted for the purpose of code compaction.

Clones may be ranked along several dimensions:

Size: Size is the number of AST nodes or the number of characters, tokens, or lines of source code, in the clone, not counting holes.

Frequency: A clone may be ranked according to its size (option `One`) or its estimated savings, which is the product of its size and the number of non-overlapping occurrences, minus one to account for the one occurrence that must remain. The latter ranking (option `All`) favors clones whose abstraction would most decrease overall code size, but it often produces small, frequent clones. Automatic tools for procedural abstraction are indifferent to clone size, but manual refactoring is not. We provide options to suit both applications.

Similarity: Similarity is the size of the clone divided by the average size of its occurrences. If the clone has no holes, every occurrence is the same size as the clone and the similarity is 100%. Clones that take large subtrees as parameters have much lower similarity percentages. The option `Percent` indicates that clones should be ranked by their similarity.

Ranking does more than simply order the clones for output. The report generator drops clones that overlap clones of higher rank. Thus rankings that favor small clones will list them early and can eliminate larger overlapping clones.

Command-line options select from the options above. For example, the default option string used below is "`Node One`", which counts nodes, favors the largest clone (ignoring the number of occurrences), and doesn't consider how similar the clone and its occurrences are.

Asta is currently a platform to evaluate clone detection on ASTs, and provides only a crude user interface. It produces a list of clones as an HTML document with three parts: a table with one row per pattern, a list of patterns with their occurrences, and the source code. Each part hyperlinks to an elaboration in the next part.

3 Measuring Size

Asta has been run on a corpus of 1,141 Java files (from the `java` directory of the Java 2 platform, standard edition (version 1.4.2)[2]) and 58 C# files (mostly from the lcsc compiler [14]). Figure 1 gives their sizes. For each file (ordered by number of AST nodes along the x-axis), the figures show the number of nodes, characters, tokens, and lines. Since these are (roughly) related by constant factors[3] in what follows, we will use node counts as a proxy for size of source code, avoiding measures that are more influenced by formatting.

4 Clone Distribution

Our primary goal is to report a list of clones that merit procedural abstraction, refactoring, or some other action. What merits abstraction is a subjective decision that is difficult to quantify. It is therefore difficult to quantitatively measure how well a system achieves this goal. Historically, research in clone detection (procedural abstraction) for code compaction used the number of source lines (or instructions) saved after abstraction as a measure of system performance. This goal is easy to quantify. A clone with p elements (lines, tokens, characters, or nodes) and r occurrences saves $p(r-1)$ elements[4]. Subtracting one accounts for the one copy of the clone that must remain.

A focus on savings tempts one to use a greedy heuristic that chooses clones based on the number of, for example, source lines they save. The clones that result may not be the ones that subjectively merit abstraction. For example, the clone that saves the most source lines in an eight-queens solver written in C# is the rather dubious:

```
for (int i = 0; i < ?; i++)
  ? = ?;
```

To our eyes, reporting clones based on the number of nodes in the clone itself (rather than the number in all occurrences) produced better clones, at least from the point of view of manual refactoring. Whenever our ranking factored in number of occurrences, we tended to see less attractive clones. However, it may be that the purpose of performing clone detection is, in fact, to compact the source code via procedural abstraction. For that application, small, frequent clones are desirable.

We explore both our primary goal of finding clones that merit abstraction and the historical goal of maximizing the

[2] http://java.sun.com/j2se/1.4.2/download.html

[3] Let n,c,t, and ℓ be the number of nodes, characters, tokens, and lines in a file. For Java, $n \approx 0.55c \approx 4.0t \approx 13.5\ell$. For C#, $n \approx 0.39c \approx 1.45t \approx 14.9\ell$.

[4] This does not consider the cost of the $r-1$ call instructions that replace $r-1$ of the occurrences.

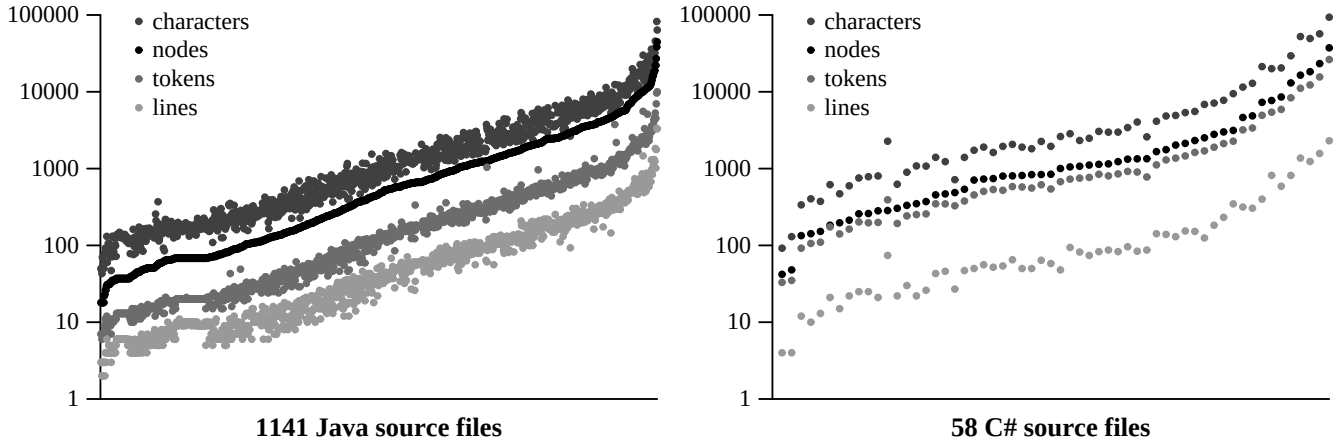


Figure 1. Java and C# source file metrics. Each column of four dots represents the number of characters, nodes, tokens, and lines in one file. The columns are ordered by number of nodes.

number of source lines saved after abstraction. The first goal we equate with finding large clones (with many nodes). To accomplish this, we rank clones by size (number of nodes) and report the size of the non-overlapping clones that we find (Figures 2 and 3). The second, historical goal, we approach by ranking clones by the number of nodes saved and report the percentage of nodes saved after abstraction (Figure 4). In both cases, we follow Asta's ranking of clones to select, in a greedy fashion, those clones that (locally) most increase the measure (eliminating from future consideration the clones they overlap).

4.1 Clones for abstraction

Figures 2 and 3 show the numbers of non-overlapping clones of various sizes found in the largest files of the Java and C# corpora. There are many small clones but also a significant number that merit abstraction.

We hand-checked all 48 clones of at least 80 nodes in the C# examples, and found that 44 represent copying that we would want to eliminate. This high success ratio suggests that many of the smaller clones should also be actionable. The number of significantly smaller clones prohibits grading by hand, but skimming suggests that a 40-node threshold gives many actionable clones and that a 20-node threshold is probably too low, just as 80 is too high.

The size of actionable Java clones is similar. A sampling of 40-node clones revealed many useful clones, while many 20-node clones are too small to warrant abstraction. As one example, the following 59 node pattern with 3 holes occurs 10 times across several Java modules:

```
for (int i=0; i<?1; i++)
  if (?2[i] != ?3[i])
    return false;
```

One of its occurrences (in `java/awt/image/ColorModel.java`) has arguments `numComponents`, `nBits`, and `nb`. Another (in `java/net/Inet6Address.java`) has arguments `INADDRSZ`, `ipaddress`, and `inetAddr.ipaddress`. This is one of the smallest examples of a structural clone that might be worthy of parameterization. Notice that the third hole matches both a lexical and structural parameter.

One of the potential benefits of allowing clone parameters to be larger subtrees than single leaves is the possibility of detecting more than just lexical inconsistencies in copy-paste clones. For example, one of the structural clones found in the C# source contains the following line[5]:

```
return malformed("real-literal", ?);
```

where one copy of the clone has `?` = `tmp.ToString()` and the other copy has `?` = `tmp`. This may be a legitimate difference, but it may also indicate a copy that missed being updated. Clone detectors that merely regularize variable names would not detect the match between these structural parameters and might miss such potential errors.

4.2 Clones for compaction

We now consider the historical goal of maximizing the number of nodes saved by abstraction. Reporting total savings is complicated by the fact that it varies significantly with the threshold on clone size. Figure 4 shows that, for our C# corpus, the total savings drops from 24% to 1% as the threshold for clone size increases from 10 to 160 nodes. If maximizing total savings is our goal, we should al-

[5]The entire clone comprises 231 nodes (21 source lines), contains one hole, and occurs twice.

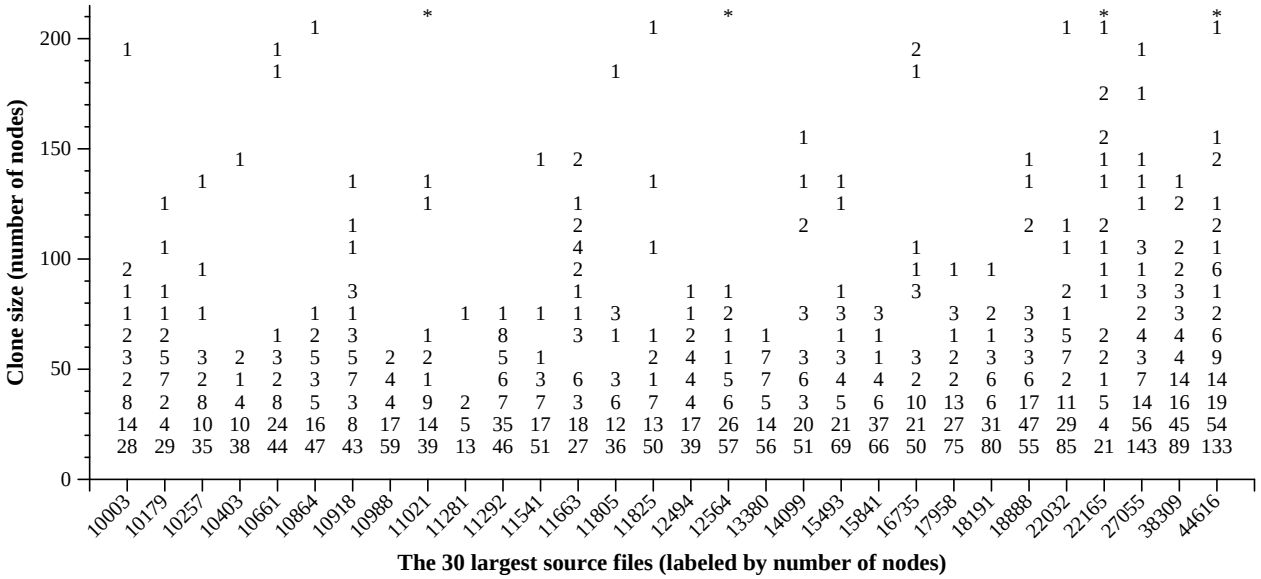


Figure 2. Number of non-overlapping clones in Java source files. For example, the 54 in the rightmost column indicates that the largest source file (44,616 nodes) has 54 non-overlapping clones, each with 20-30 nodes. An asterisk indicates a clone whose size is off the scale. (The maximum size clone has 913 nodes.)

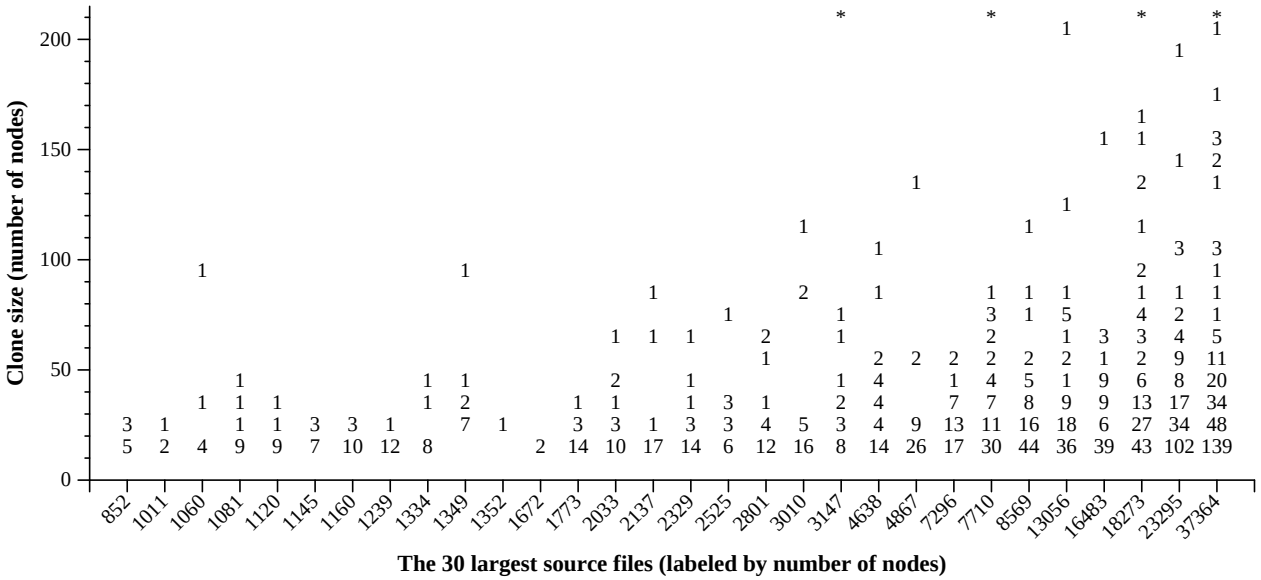


Figure 3. Number of non-overlapping clones in C# source files. For example, the 48 in the rightmost column indicates that the largest source file (37,364 nodes) has 48 non-overlapping clones, each with 20-30 nodes. An asterisk indicates a clone whose size is off the scale. (The maximum size clone has 605 nodes.)

low the automatic abstraction of small clones, even though these clones may not be large enough to merit abstraction by hand. If we would rather avoid abstracting small clones, thresholds between 20 and 80 nodes eliminate many of the small, dubious clones and still yield savings of 4-16%.

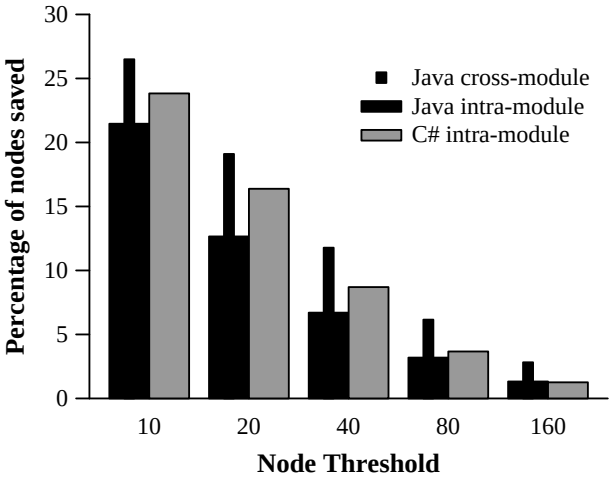


Figure 4. Percentage of nodes eliminated by abstraction

We should emphasize that our results (the wide bars in Figure 4) represent the execution of Asta on each individual file in isolation. If instead, we allow Asta to find clones that occur in multiple files, we obtain greater savings. Figure 4 shows the difference for the Java corpus (the narrow bars). The savings across multiple modules is obtained by finding clones that occur anywhere within the approximately 400,000 lines of Java source.

By comparison, Baker [2] reports saving about 12% by abstracting clones of at least 30 lines in inputs with 700,000 to over a million lines of code; she reports that most 20-line clones are actionable and that most 8-line clones are not. Baxter et al. also report saving roughly 12% on inputs of about 400,000 lines of code; they too use a threshold and conclude that most clones are on the order of 10 lines.

Our threshold of 20 nodes is far smaller than Baker's 30-line threshold. That we still observe mostly actionable clones at this threshold may be understood as a difference in the definition of *actionable*, or as a difference in the corpora, source languages, or abstraction mechanism. Our smaller threshold is matched by our smaller input sizes: our largest module contains about 45,000 lines of source. As mentioned, we can apply our techniques across multiple modules (as shown in Figure 4), but there is also redundancy and duplication within individual files.

Remarkably, the savings we obtain by abstracting actionable clones within isolated files is roughly the same as that obtained by both Baker and Baxter et al. This is somewhat disappointing since our system finds clones based not only on lexical abstraction (as in Baker and Baxter et al.) but also on structural abstraction. Either there are very few clones that are purely structural in nature, or individual files contain fewer clones (that we view as actionable) than the large corpora examined by Baker and Baxter et al. The following section makes the case for the latter interpretation.

5 Lexical versus structural abstraction

Prior clone detection algorithms are based on lexical abstraction, which abstracts lexical tokens. Structural abstraction can abstract arbitrary subtrees and thus should be expected to find more clones. One objective of this research has been to determine if this generality translates into practical benefit and, if so, to characterize the gain.

Clones are easily classified as lexical or structural. An occurrence of a clone is *lexical* if each of the clone's holes is occupied by an actual argument that is an identifier or literal. If a clone has two or more lexical occurrences, then it might have been found by lexical abstraction and is thus called a *lexical clone*; otherwise, it is called a *structural clone*.

In the ASTs produced by JavaML and lcsc, identifiers and literals appear as leaves but, depending on context, can be wrapped in or hung below one or more unary nodes. We classify arguments or holes conservatively: if an argument is a leaf or a chain of unary nodes leading to a leaf, then we count it as a lexical abstraction. Only more complex arguments are counted as structural abstractions. For example, suppose the clone `a[?] = x;` occurs twice:

```
 a[i] = x;
a[i+1] = x;
```

The argument to the first occurrence is lexical because it includes only a leaf and, perhaps, a unary node that identifies the type of leaf. The argument to the second occurrence is, however, structural because it includes a binary AST node.

Asta's HTML output optionally shows the arguments to each occurrence of each clone, and it classifies each argument as lexical or structural. Because Asta can generate clones that a human might reject, we checked a selection of C# source files by hand. Figure 3 includes 48 clones of 80 or more nodes. 32 were structural and 16 were lexical. 28 of the structural clones and all of the lexical clones were deemed useful. Thus a significant fraction of these large clones are structural, and most of them merit abstraction.

There are, of course, too many clones to check all of them by hand, so we present summary data on the ratio of structural to lexical clones. This ratio naturally varies with the thresholds for holes and clone size.

First, fixing the hole threshold at 3 and raising the node threshold from 10 to 160 gives the left half of Figure 5.

As the threshold on clone size rises, Asta naturally finds fewer clones, but note that structural clones account for an increasing fraction of the clones found.

If, instead, we vary the hole threshold, we obtain the right half of Figure 5, in which the node threshold is fixed at 10 and the hole threshold varies from zero to five. The ratio of structural to lexical clones rises because each additional hole increases the chance that a clone will have a structural argument and thus become structural itself.

Clones with zero holes are always lexical because they have no arguments at all, much less the complex arguments that define structural clones. Predictably, the number of structural clones grows with the number of holes. At the same time, the number of lexical clones may decline slightly because some of the growing number of structural clones out-compete some of the lexical clones in the rankings.

Clones with fewer holes are generally easier to exploit, just as library routines with fewer parameters are easier to understand and use. Even if we restrict our refactoring effort to one-parameter macros, we still see that 20% of the opportunities involve structural abstraction, which is significant. Optimizations are deemed successful with much smaller gains, and improving source code is surely as important as improving object code. Figure 5 explores a large range of the configuration options that are most likely to be useful, and it shows significant numbers of structural clones for all of the non-trivial settings.

6 Related work

The most closely related work to ours is by Baxter et al. [5] who perform clone detection in ASTs. They use a hash function to place each full subtree of the AST into a bucket. Then every two full subtrees within a bucket are compared. The hash function is chosen to be insensitive to identifier names (leaves) so that these can be parameters in a procedural abstraction. In order to allow larger subtrees to be parameters, an even more insensitive hash function could be used. However, the cost of this is an increased bucket size and a larger set requiring pairwise comparison. Asta avoids this by growing larger matches from smaller ones, essentially hashing the first few levels of each full subtree (the d-caps) and then extending them as needed. This method finds *any* duplicated subtree not just duplicated full subtrees.

Yang [26] uses a language's grammatical structure (and ASTs in particular) to calculate the difference between two programs via dynamic programming. He addresses a different problem than clone detection, but his method could be used for that purpose and could be used to find the general subtree clones that we find. However, it would require $\Omega(n^4)$ time on an n node AST, which is impractical for all but the smallest programs.

Koschke et al. [20] also detect clones in ASTs. They serialize the AST and use a suffix tree to find full subtree copies. This technique does not permit structural parameters.

Jiang et al. [15] cluster feature vectors that summarize subtrees of a parse tree or AST. The vectors count the number of nodes in each of several categories. By using locality-sensitive hashing, they can promptly identify trees with similar vectors, without comparing all pairs of trees. The trade-off is that the vectors conflate trees with the same summary characteristics but different structures.

The tools CCFinder [16] and CP-Miner [21] also do not find clones with structural parameters. CCFinder is a token-based, suffix-tree algorithm that allows parameterized clones by performing a set of token transformation rules on the input. CP-Miner converts each basic block of a program into a number and looks for repeated sequences of these numbers, possibly with gaps. It also allows parameterized clones by regularizing identifiers and constants. Neither method produces structural parameters.

Tools that automatically perform procedural abstraction, rather than simply flagging potential clones, also permit some degree of parameterization in the abstracted procedure. These tools typically operate on assembly code and most allow register renaming [9, 10, 25]. Cheung et al. [6] take advantage of instruction predication (found, for example, in the ARM instruction set [24]) to nullify instructions that differ between similar code fragments. The parameters to the abstracted representative procedure are the predication flags, which select the instructions to execute for each invocation. One flag setting could select an entirely different sequence of instructions than another, however for the representative to be small, many instructions should be common to many fragments. A shortest common supersequence algorithm finds the best representative for a set of similar fragments [6]. The method is not intended for a large number of fragments with many parameters.

Another generalization uses slicing to identify non-contiguous duplicates and then moves irrelevant code out of the way [18]. This extension catches more clones than lexical abstraction, but parameterization remains based on lexical elements. This extension is orthogonal to this paper's generalization. The two methods could be used together and ought to catch more clones together than separately.

Finding clones in an AST might appear to be a special case of the problem of mining frequent subtrees [7, 27], but closer examination shows that the two problems operate at two ends of a spectrum. Algorithms that mine frequent trees scan huge forests for subtrees that appear under many roots. The size and exact number of occurrences are secondary to the "support" or number of roots that hold the pattern. An AST-based clone detector makes the opposite trade-off. The best answer may be a clone that occurs only twice, if

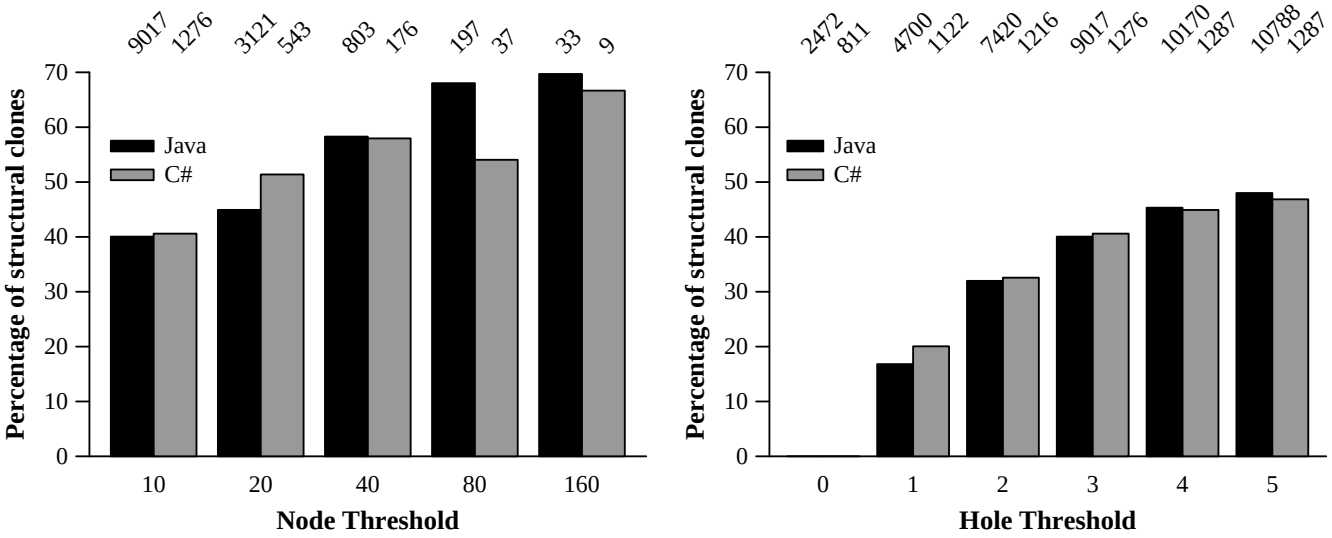


Figure 5. Percentage of structural clones for various node and hole thresholds. The number above each column denotes the total number of clones for the given threshold.

it is big enough. Size and exact number of occurrences are important. Support is secondary; indeed, some interesting clones may occur in only one tree of the forest.

7 Discussion

Asta has been written in Icon [12] and Java. The Icon version takes a few seconds on most corpus modules and about 7 minutes on the largest. Icon is interpreted and dynamically typed, and the program has not been optimized for speed, so these running times are high. The Java version takes a few seconds on all corpus modules, even the largest. Finding all clones across all modules in the 440,000 line corpus took less than one hour.

Our structural abstraction method can benefit from variable renaming (a technique described by Baker [3]) since variables that can be named consistently in all clone occurrences no longer need to be represented as holes in the clone. This reduces the number of parameters that need to be passed to the abstracted procedure in the calls that replace the clone occurrences, and thus these clones save more when abstracted as procedures. Experimental results show an extra savings of about 20% for our Java corpus when combining structural abstraction with variable renaming [22].

In summary, we have designed, implemented, and experimented with a new method for detecting cloned code. Heretofore, abstraction parameterized lexical elements such as identifiers and literals. Our method generalizes these methods and abstracts arbitrary subtrees of an AST. We have shown that the new method is affordable and finds a significant number of clones that are not found by lexical methods.

References

[1] G. J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995.

[3] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, Oct. 1997.

[4] B. S. Baker and U. Manber. Deducing similarities in Java sources from bytecodes. In *Proc. USENIX Annual Technical Conference*, pages 179–190, June 1998.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[6] W. Cheung, W. Evans, and J. Moses. Predicated instructions for code compaction. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, pages 17–32, 2003.

[7] Y. Chi, S. Nijssen, R. R. Muntz, and J. N. Kok. Frequent subtree mining–an overview. *Fundamenta Informaticae*, 66(1–2):161–198, Mar. 2005.

[8] K. Church and J. Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, 1993.

[9] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *ACM Conference on Programming Language Design and Implementation*, pages 139–149, May 1999.

[10] S. K. Debray, W. Evans, R. Muth, and B. de Sutter. Compiler techniques for code compaction. *ACM Trans. Prog. Lang. Syst.*, 22(2):378–415, Mar. 2000.

[11] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.

[12] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, 1996.

[13] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–279, July 1993.

[14] D. R. Hanson and T. A. Proebsting. A research C# compiler. *Software-Practice and Experience*, 34(13):1211–1224, Nov. 2004.

[15] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, May 2007.

[16] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Engineering*, 28(7):654–670, July 2002.

[17] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees, and arrays. In *Proc. ACM Symposium on Theory of Computing*, pages 125–136, 1972.

[18] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the Eighth International Symposium on Static Analysis*, pages 40–56, 2001.

[19] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3:77–108, 1996.

[20] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the IEEE Working Conference on Reverse Engineering*, pages 253–262, 2006.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Engineering*, 32(3):176–192, Mar. 2006.

[22] F. Ma. On the study of tree pattern matching algorithms and applications. Master's thesis, Department of Computer Science, University of British Columbia, Aug. 2006.

[23] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 244–253, Nov. 1996.

[24] D. Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, second edition, 2001.

[25] B. D. Sutter, B. D. Bus, and K. D. Bosschere. Sifting out the mud: Low level C++ code reuse. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 275–291, Nov. 2002.

[26] W. Yang. Identifying syntactic differences between two programs. *Software-Practice and Experience*, 21(7):739–755, July 1991.

[27] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, Aug. 2002.