Clone Detection via Structural Abstraction

William S. Evans

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4, CANADA

Christopher W. Fraser

Microsoft Research
Redmond, WA 98052, USA

August 2005

Technical Report

MSR-TR-2005-104

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 16,000-line code corpus, 20-50% of its clones eluded previous methods. The method also identifies cloning in declarations, so it is somewhat more general than conventional procedural abstraction.

1. Introduction

Programmers clone source code by copying a fragment and then optionally changing the copy. Cloning 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 or refactor [Griswold and Notkin, Fowler et al] the code. For example, the programmer might create a new procedure that abstracts the clones. Differences between the clones require formal parameters to the procedure.

There is much prior work in this area, operating on source code [Baker 1995, 1997], abstract syntax trees (ASTs) [Baxter et al], program dependence graphs [Komondoor and Horwitz], bytecode [Baker and Manber] and assembly code [Cooper and McIntosh; Debray et al; De Sutter et al; Fraser et al]. The methods also range across different scales, from millions of lines of source code [Baker 1995] to a single executable [Debray et al].

Clone detectors offer a range of outputs. Some mainly flag the clones in a graphical output, such as a dot-plot [Church and Helfman]. 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 [Komondoor and Horwitz]. Still others automatically perform procedural abstraction [Cooper and McIntosh; Debray et al; De Sutter et al; Fraser et al], 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.

2. Related work

Most previous methods start with what might be called *lexical abstraction*, because a process akin to a compiler's lexical analyzer identifies the elements that can become parameters to an abstracted procedure. Typically, the process treats identifiers and numbers for source code, or register numbers and literals for assembly code as equivalent; or, alternatively, 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. Lexical abstraction can generate many false positives, but parameterized pattern matching [Baker 1997] compensates by requiring a one-for-one correspondence between parallel parameters.

Lexical abstraction has much to recommend it. It is fast and catches many clones. But does it miss clones? That is, could a more general mechanism catch more clones?

One generalization takes advantage of instruction predication (found, for example, in the ARM instruction set [Seal]) to nullify instructions that differ between similar clones. The parameters to the clones' abstracted representative are the predication flags, which select the instructions to execute for each invocation. A shortest common super-sequence algorithm finds the best representative for a set of similar clones [Cheung et al]. The method is not intended for large clones or many parameters.

Another generalization uses slicing to identify non-contiguous clones and move irrelevant code out of the way [Komondoor and Horwitz]. This extension clearly 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.

Another natural generalization of lexical abstraction allows the creation of parameters that represent entire subtrees of a parser's tree representation of source code. Parse or derivation trees are riddled with artificial duplication that interests only parsers, but abstract syntax trees (ASTs) project out this duplication and are roughly equivalent to source code, as evidenced by AST pretty-printers that reconstruct usable source code. We call this generalization *structural abstraction*.

Where lexical abstraction parameterizes (or wildcards) only AST leaves, structural abstraction parameterizes arbitrary subtrees. Structural abstraction is thus clearly more general but also considers a much larger search space of abstractions or wildcard placements.

There is some prior work on clone detection in ASTs though not fully general structural abstraction as defined above. One method tracks only a subset of the AST features [Kontogiannis et al]. Another method [Baxter et al] uses the full AST, starts with lexical abstraction, and adds a method specialized to detect the common prefix of lists of statements, expressions, etc.

The current 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 16,000 lines of source code. We 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.

Finding clones in an AST might appear to be a special case of the problem of mining frequent subtrees [Chi et al., Zaki], 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 it is big enough. Size and exact number of occurrences are important. Support is secondary; indeed, some interesting forests hold only a single root. The existing subtree mining algorithms are not designed or tuned to find clones in ASTs.

3. Algorithm

Our structural abstraction prototype is called Asta. Asta accepts a single AST represented as an XML string. It has been used with ASTs from the C# compiler lcsc [Hanson and Proebsting]. A custom back end for 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.

lcsc's 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 vertices 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. A *full subtree* is a subtree whose leaves are leaves of the original tree.

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 one-to-one *matching* 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 $v \in V(P)$, $\text{label}(v)=\text{label}(f(v))$ or $\text{label}(v)=?$. In our case, T is a 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,

```
add(?,constant(7))
```

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

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

is not a valid pattern because the wildcard replaces an operator but not the full subtree rooted at 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.

3.1 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 i-cap ($1 \le i \le D$) for v is the pattern obtained by taking the depth i subtree rooted at v and adding holes in place of all the children of nodes at depth i. If the subtree rooted at v has depth $d < D$, then node v has only d caps. 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 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 [Karp et al.] 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, essentially, hashing.

After creating the set 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 r in S a list of patterns that match the subtree rooted at r. 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 preserve two of the occurrences of the original pattern.

For each pair of occurrences, T_i and T_j ($1 < i < j < m$) of a given pattern P with m 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 $1 < i < j < m$) 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 the source pattern, P, for a pattern, Q, created by the pattern improvement operation.

3.2 Thinning and ranking

Asta can find a lot of clones, sometimes too many, so the candidates are thinned and ranked before output. Asta is still at an experimental research stage, so it 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 (very) roughly 12-14 nodes per line, so some sub-line clones are reported.

Clones may be ranked along several dimensions:

1. **Size**: the number of AST nodes or the number of characters, tokens, or lines of source code, in the clone, not counting holes.
2. **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, and 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.
3. **Similarity**: the size of the clone divided by the average size of its occurrence. 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. This option is called `Percent`. It is turned off by the option `Count`.

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 may eliminate larger overlapping clones.

Command-line options select from the options above. For example, the default option string used below is "`Node One Count`", which counts nodes, favors the largest clone, and ignores the number of occurrences.

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

4. Reporting clones

Asta is currently a platform to evaluate clone detection on ASTs, not a tool for general programmers. It thus does not currently offer a polished user interface, and it reports a lot of clones. For evaluation, we want to see all useful clones, even if that means seeing some that aren't useful. Indeed, we particularly want to see cases where awkward clones rank high, because they suggest ways to improve the rankings. Ideally, the top ranked clones will be actionable and the lower rankings will be too small to use. For now, we'd rather include noise than miss something.

The clone report is 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. For example, Figure 1 gives most of the report for an eight-queens solver. The first row of the table presents the top-ranked clone:

```
rows[r]=up[r-c+7]=down[r+c]=?;
```

The hyperlinks in the last column of the table point to the second section, which shows all occurrences of the clone. The second section points, in turn, into the third section, which holds the source code for the original input (mostly elided from Figure 1 to save space), so that the occurrences can be viewed in context.

The first column of the table gives the percentage of similarity. Columns 2 and 3 of the table report the number of nodes (excluding holes) and the number of holes in the clone. Column 4 gives the number of non-overlapping occurrences. The top-ranked clone in Figure 1 has 40 nodes and one hole or, equivalently, one formal parameter. It has two occurrences, which appear in the second section. One occurrence takes `true` as its actual argument; the other takes `false`.

Figure 1. Sample clone report

%sim	#nodes	#holes	#hits	Source
0.952	40	1	2	"rows[r] = up[r-c+7] = down[r+c] = ARGTREE(2);\n\t\t\t"
0.607	24	3	2	"for (int i = 0; i < ARGTREE(1) ; i++)\n\t\t\t ARGTREE(8) = ARGTREE(11);\n\t\t"
0.122	11	3	2	" ARGTREE(1) static void ARGTREE(1) () {\n\t\t ARGTREE(92) }\n"
0.916	11	1	2	" ARGTREE(1) = new Boolean[15]"

Patterns and occurrences:

- "rows[r] = up[r-c+7] = down[r+c] = ARGTREE(2) ;\n\t\t\t"
  - "rows[r] = up[r-c+7] = down[r+c] = true;\n\t\t\t"
  - "rows[r] = up[r-c+7] = down[r+c] = false;\n\t\t\t\t"
- "for (int i = 0; i < ARGTREE(1) ; i++)\n\t\t\t ARGTREE(8) = ARGTREE(11) ;\n\t\t"
  - "for (int i = 0; i < 15; i++)\n\t\t\tup[i] = down[i] = true;\n\t\t"
  - "for (int i = 0; i < 8; i++)\n\t\t\trows[i] = true;\n\t\t"
- " ARGTREE(1) static void ARGTREE(1) () {\n\t\t ARGTREE(92) }\n"
  - "private static void print() {\n\t\tfor (int k = 0; k < 8; k++)\n\t\t\tConsole.Write(\"{0} \", x[k]+1);\n\t\tConsole.Write(\"\\n\");\n\t}\n"
  - "public static void Main() {\n\t\tfor (int i = 0; i < 8; i++)\n\t\t\trows[i] = true;\n\t\tfor (int i = 0; i < 15; i++)\n\t\t\tup[i] = down[i] = true;\n\t\tqueens(0);\n\t}\n\n\t"
- " ARGTREE(1) = new Boolean[15]"
  - "down = new Boolean[15]"
  - "up = new Boolean[15]"

8q.cs:

```
using System;
public class Queens { <see Appendix> }
```

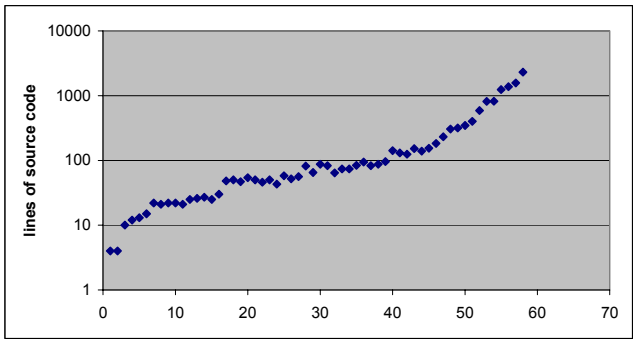
In patterns, ARGTREE(n) flags a hole and gives the number of nodes in the largest argument tree that occupies that hole. For example, the second row abstracts two simple loops with bodies that comprise a single assignment. The ARGTREE(11) reports eleven nodes in the largest right-hand side in either of the two occurrences. The ASTs used in this paper typically have ARGTREE(1) or ARGTREE(2) for identifiers and constants, which lexical abstraction can catch. Larger values generally denote holes that require Asta's more general structural abstraction.

Figure 1 illustrates both pros and cons of structural abstraction. The pattern in the first row catches an important bit of logic from eight queens and arguably merits refactoring. The other rows are less promising and appear as a result of low thresholds. The second row abstracts a specialized loop, but the repeated pattern is smaller (only 24 nodes and two lines) and the similarity percentage is small. The third row is small (only 11 nodes) and has a very small similarity percentage; a threshold on similarity is indicated. The last row improves similarity but not size. A macro would scarcely be worth the effort. We could clearly run with tighter thresholds but we wanted to illustrate the need for them here.

5. Metrics

Asta has been run on a corpus of 58 C# programs. Figure 2 gives their sizes. Most of the code is from the lcsc compiler.

Figure 2. File sizes



Asta can rank clones and report savings in nodes, characters, tokens, and lines. These measures can be sensitive to design or formatting issues that merit elaboration. Node counts are easy to compute but depend somewhat on the design of the abstract syntax tree's node types. Fortunately, lcsc's abstract syntax closely matches the source language definition and introduces little noise.
The other three measures count elements of the source code. Lcsc's ASTs are labeled with line numbers and character positions within each line, though a few nodes are synthesized (such as the null nodes that end lists of statements and parameters), so coordinates must be inferred for them. Also, line and character counts are sensitive to white space, comments, and the fact that some programmers use longer variable names than others. Asta partially compensates by discounting blank lines and comments, though this discounting is an approximation. The token counter gets accurate data from the ASTs and is thus the only source-based measure that is immune to formatting issues.

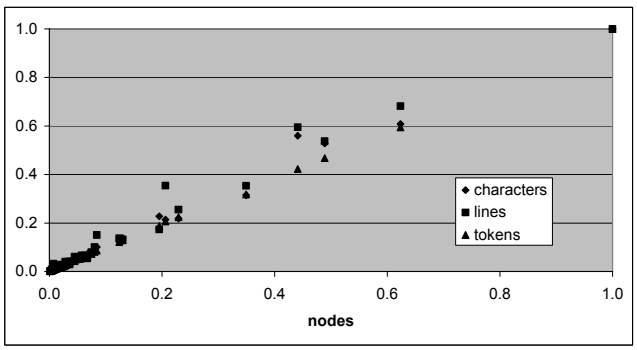
Multiple measures raise the question of which to use. Line counts are a commonly quoted measure of savings, but character counts better reflect the source size, and token counts better reflect the number of primitive elements in the program. For example, is one clone covering some long variable names really any better than the same clone covering short variable names?

Ideally, the number of nodes and tokens—which we can measure precisely—would vary linearly with the number of characters and lines, but do they? Figure 3 addresses these questions with a scatter plot of the number of nodes versus the number of characters, tokens, and lines in the sample corpus, normalized so that three scales can share one graph.

Figure 3 shows that, at least for this corpus, node counts are roughly linear in the number of characters, tokens, and lines. Lines and characters show a few more outliers, but tokens track nodes well. This finding is consistent with the fact that characters and lines involve more formatting issues, which are peripheral to the issue of cloning.

We can thus assume that node and token measures are roughly equivalent and, in what follows, we use node counts as a proxy for size of source code, avoiding measures that are more influenced by formatting.

Figure 3. Source metrics versus nodes



6. Measured Clones

Having addressed the question of what to measure, we turn to measuring the performance of structural abstraction. 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 quantify how well a system achieves this goal. Historically, research in clone detection (procedural abstraction) used the number of source lines (or instructions) saved after abstraction as a measure of system performance. This goal is easy to quantify. A pattern with p elements (lines, tokens, characters, or nodes) and h occurrences saves $p(h-1)$ elements. Subtracting one accounts for the one copy of the pattern 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 `8q.cs` (Figure 1) is the rather dubious:

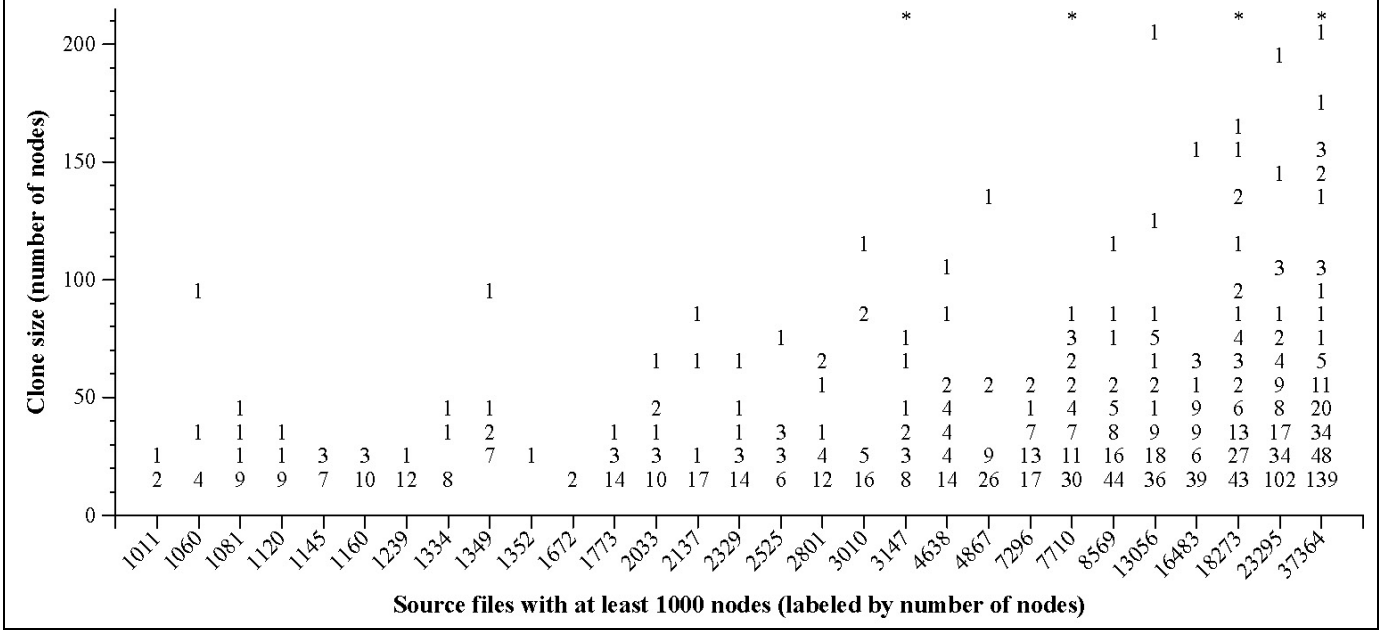
```
for (int i = 0; i < ?; i++)
 ? = ?;
```

To our eyes, reporting clones based on the number of nodes or tokens 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 frequency, 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 source code 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 and number of the non-overlapping clones that we find (Figure 4). 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 5). 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).

Figure 4 shows many small clones but also a significant number that merit abstraction. We hand-checked all 48 patterns of at least 80 nodes, and we found that 44 represent copying that we would want to eliminate. This 91.66% success ratio suggests that many 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 useful clones and that a 20-node threshold is probably too low, just as 80 is too high.

Figure 4. Number of non-overlapping clones of various sizes. 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.)

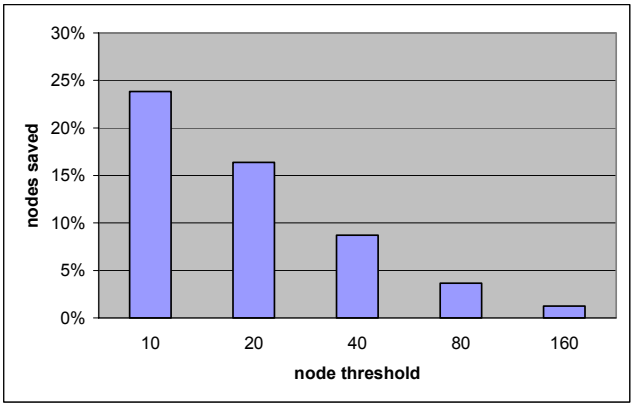


We now consider the historical goal of maximizing number of nodes saved by abstraction. Reporting total savings is complicated by the fact that it varies significantly with the threshold on pattern size. Figure 5 shows that, for our corpus, the total savings drops from 24% to 1% as the threshold for pattern size increases from 10 to 160 nodes. If maximizing total savings is our goal, we should allow the automatic abstraction of small patterns, even though these patterns 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%.

By comparison, Baker [1995] reports saving about 12% when using a threshold of 30 lines; 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%; they too use a threshold but do not specify what its value is.

Our thresholds are lower than Baker's—80 nodes corresponds (very) roughly to six lines of code—but still result in actionable clones. We use a different corpus, different source language, and different abstraction, so we should not be surprised by different results. Remarkably, the savings we obtain by abstracting actionable clones (greater than 20 nodes) is roughly the same as that obtained by both Baker and Baxter et al. This is 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 our corpus contains fewer clones than those examined by Baker and Baxter et al. The following section makes the case for the latter interpretation.

Figure 5. Total savings



7. 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 patterns. 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 is lexical if and only if each of its holes is occupied by an actual argument that is an identifier or literal. If a pattern has two or more lexical occurrences, then it might have been found by lexical abstraction and is thus counted as lexical; otherwise, it is counted as structural.

In lcsc's ASTs, an identifier or literal appears as a leaf but, depending on context, may have a unary parent (specifying its type) or an even longer chain of unary ancestors. 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;
```

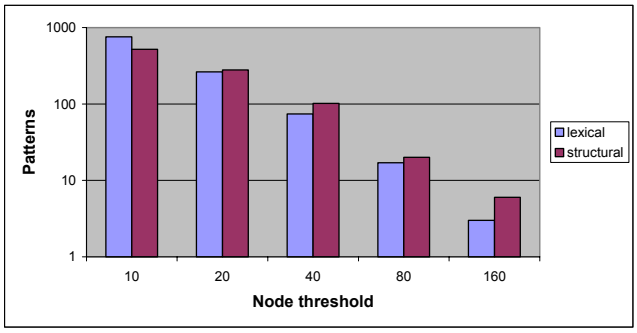
The argument to the first occurrence is lexical because it includes only a leaf and, perhaps, a unary node that identifies the type of leaf. The argument to the second occurrence is, however, structural because it includes a binary AST node.

Asta's HTML output optionally shows the arguments to each occurrence of each pattern, and it classifies each argument as lexical or structural. Because Asta can generate patterns that a human might reject, we checked a selection by hand. Figure 4 includes 48 patterns of 80 or more nodes. 32 were structural and 16 were lexical. 28 of the structural patterns and all of the lexical patterns were deemed useful. Thus a significant fraction of these large patterns are structural, and most of them merit abstraction.

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

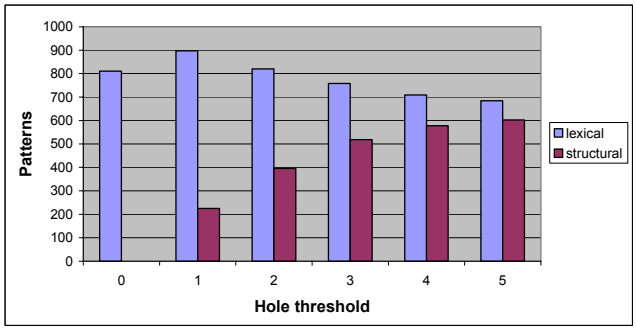
First, fixing the hole threshold at 3 and raising the node threshold from 10 to 160 gives the results in Figure 6. As the threshold on pattern size rises, Asta naturally finds fewer patterns, but note that structural patterns account for an increasing fraction of the patterns found.

Figure 6. Abstraction versus node threshold



If, instead, we vary the hole threshold, we obtain Figure 7, in which the node threshold is fixed at 10 and the hole threshold varies from zero to five. The ratio of structural to lexical patterns rises because each additional hole increases the chance that a pattern will have a structural argument and thus become structural itself.

Figure 7. Abstraction versus hole threshold



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

Patterns with fewer holes are generally easier to exploit—just as library routines with fewer parameters are easier to understand and use—but even one-parameter macros miss roughly 20% of the opportunities without 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. Figures 6-7 explore a large range of the configuration options that are most likely to be useful, and they show significant numbers of structural patterns for all of the non-trivial settings.

8. Discussion

On a 2.6GHz P4, Asta takes a few seconds on most corpus modules and about 7 minutes on the largest, but these numbers exaggerate the costs. Asta is implemented in Icon [Griswold and Griswold], which is interpreted and dynamically typed. Care has been taken with the asymptotic performance, but the implementation has otherwise been tuned to simplify experiments, not to save time. We expect that a compiled implementation would run at least an order of magnitude faster.

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.

9. References

Brenda S. Baker. On finding duplication and near-duplication in large software systems. *Proceedings of the Second IEEE Working Conference on Reverse Engineering*:86-95, 7/1995.

Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing 26*(5):1343-1362, 10/1997.

Brenda S. Baker and Udi Manber. Deducing similarities in Java sources from bytecodes. *Proceedings of the USENIX Annual Technical Conference*:179-190, 1998.

Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. *Proceedings of the International Conference on Software Maintenance*:368-377, 1998.

Warren Cheung, William Evans, and Jeremy Moses. Predicated instructions for code compaction. *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*:17-32, 2003.

Yun Chi, Siegfried Nijssen, Richard R. Muntz, and Joost N. Kok. Frequent Subtree Mining—An Overview. *Fundamenta Informaticae 66*(1-2):161-198, 3/2005.

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.

Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. *Proceedings of the ACM SIGPLAN 1999 Conference on Programming language Design and Implementation:*139-149, 1999.

Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM TOPLAS 22*(2):378-415, 3/2000.

Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Sifting out the mud: Low level C++ code reuse. *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*:275-291, 11/2002.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analyzing and compressing assembly code. *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*:117-121, 6/1984.

Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Peer-to-Peer Communications, 1996.

William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology 2*(3):228-279, 7/1993.

David R. Hanson and Todd A. Proebsting. A research C# compiler. *Software—Practice and Experience*, to appear.

Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays. *STOC '72: Proceedings of the fourth annual ACM Symposium on Theory of Computing*, 125-136, 1972.

Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. SAS 2001. *Proceedings of the Eighth International Symposium on Static Analysis*:40-56, 2001.

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.

David Seal, editor. ARM Architecture Reference Manual. Addison Wesley, second edition, 2001.

Mohammed J. Zaki. Efficiently mining frequent trees in a forest. Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining:71-80, 8/2002.

Appendix: Source code for eight-queens example

```
using System;

public class Queens {
   static Boolean[] up = new Boolean[15],
      down = new Boolean[15],
      rows = new Boolean[8];
   static int[] x = new int[8];

   public static void Main() {
      for (int i = 0; i < 8; i++)
         rows[i] = true;
      for (int i = 0; i < 15; i++)
         up[i] = down[i] = true;
      queens(0);
   }

   private static void queens(int c) {
      for (int r = 0; r < 8; r++)
         if (rows[r] && up[r-c+7] && down[r+c]) {
            rows[r] = up[r-c+7] = down[r+c] = false;
            x[c] = r;
            if (c == 7)
               print();
            else
               queens(c + 1);
            rows[r] = up[r-c+7] = down[r+c] = true;
         }
   }

   private static void print() {
      for (int k = 0; k < 8; k++)
         Console.Write("{0} ", x[k]+1);
      Console.Write("\n");
   }
}
```