A type system for regular expressions

Eric Spishak   Werner Dietl   Michael D. Ernst
Programming Languages & Software Engineering Group, University of Washington
{espishak,wmdietl,mernst}@cs.washington.edu

ABSTRACT

Regular expressions are used to match and extract text. It is easy for developers to make syntactic mistakes when writing regular expressions, because regular expressions are often complex and different across programming languages. Such errors result in exceptions at run time, and there is currently no static support for preventing them.

This paper describes practical experience designing and using a type system for regular expressions. This type system validates regular expression syntax and capturing group usage at compile time instead of at run time — ensuring the absence of `PatternSyntaxExceptions` from invalid syntax and `IndexOutOfBoundsExceptions` from accessing invalid capturing groups.

Our implementation is publicly available and supports the full Java language. In an evaluation on five open-source Java applications (480kLOC), the type system was easy to use, required less than one annotation per two thousand lines, and found 56 previously-unknown bugs.

Categories and Subject Descriptors

D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification—*Reliability*; D.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features—*Data types and structures*

General Terms

Verification, Languages, Documentation

Keywords

Regular expressions, regex, regexp, type system, Checker Framework, case studies

1. INTRODUCTION

A regular expression is a pattern for matching certain strings of text. In Java, regular expressions are used as follows:

```
Pattern p = Pattern.compile("(.*) ([0-9]+)");
Matcher m = p.match("number 447");
if (m.matches()) {
    System.out.println(m.group(2)); // output: 447
}
```

This code first creates a regular expression with two capturing groups. A capturing group is delimited by parentheses and indicates a subpart of a regular expression. The code then prints the text matched by the second capturing group.

It is easy to make mistakes when using regular expressions. One type of error is a regular expression with invalid syntax. For example, the following code compiles correctly, but it throws a `PatternSyntaxException` at run time because of the unclosed parenthesis at the end of the regular expression:

```
Pattern p = Pattern.compile("(.*) ([0-9]+)(");
```

Invalid regular expressions are a real, serious problem for users of software written in Java. We performed a Google search that yielded over 80,000 bug reports mentioning `PatternSyntaxException`.

Another type of error is the use of an incorrect capturing group number. The following code throws an `IndexOutOfBoundsException`, because group 3 does not exist in the regular expression:

```
Pattern p = Pattern.compile("(.*) ([0-9]+)");
Matcher m = p.match("number 447");
if (m.matches()) {
    System.out.println(m.group(3)); // exception!
}
```

This paper presents a type system that checks the validity of regular expressions and regular expression capturing group access at compile time rather than at run time. This approach helps developers find bugs in syntax and capturing group usage of regular expressions earlier and more effectively than the previous method of relying on run-time errors.

This paper is organized as follows. Section 2 describes the type system for regular expressions. Section 3 describes implementation details of the type system. Section 4 presents results of using this type system on five Java applications. Section 5 discusses related work. Section 6 concludes.

2. TYPE SYSTEM

The regular expression type system expresses two varieties of information. First, the type system distinguishes between `Strings` that are syntactically valid regular expressions and `Strings` that might not be. This enables the type system to verify that a syntactically valid regular expression is used where required (see Section 2.1). Second, the type system tracks the number of capturing groups in each regular expression. This enables the type system to verify correct parameter usage for method calls that take a capturing group number (see Section 2.2).

The type system is conservative: it issues a warning whenever it cannot guarantee correct usage. Most warnings indicate a regular-expression-related bug (see Section 4.2), but some are false positives (see Section 4.3).

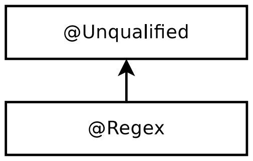

Figure 1: The basic subtyping relationship of the type system's qualifiers.

2.1 Expressing valid regular expression syntax

The type system uses a `@Regex` type qualifier to qualify a type that is a syntactically valid regular expression. In Java 8, a type qualifier is expressed as a "type annotation" [12], which begins with `@` and appears before a type name, as in "`@Regex String`". Most `String`s in an application are (implicitly) annotated with `@Unqualified`, which indicates no knowledge about whether the value is or is not a syntactically valid regular expression. The `@Regex` qualifier is a subtype of the `@Unqualified` qualifier (Figure 1).

As an example of use, the first argument to `String`'s `replaceAll` method must be a valid regular expression. To signify this, the first argument's type is `@Regex String`:

```
public String replaceAll(@Regex String regex, String replacement)
```

Following are additional examples of use of the `@Regex` qualifier:

```
@Regex String regex2 = "(reg)(ex)*"; // regex literal
@Regex String regex3 = regex2 + "(re)"; // concatenation
@Regex String nonRegex = "non ("; // compile-time error
```

The third line shows an example of a compile-time error. The `String` literal is not a valid regular expression, therefore its type is `@Unqualified String` and it cannot be assigned to a `@Regex String`.

2.2 Expressing valid capturing group usage

The `@Regex` type qualifier takes an optional parameter, indicating that the given regular expression has at least that many capturing groups (Figure 2).

Here is an example of the `@Regex` qualifier used with the capturing groups parameter:

```
@Regex(2) String regex2 = "(reg)(ex)*";
@Regex(3) String regex3 = regex2 + "(re)";
// legal declaration, though the type is not as precise as possible
@Regex(1) String regex1 = regex3;
// compile-time error, @Regex(2) is not a subtype of @Regex(4)
@Regex(4) String regex4 = regex2;
```

The `@Regex` qualifier with capturing groups can also be used on `Pattern` and `Matcher` types to indicate the number of capturing groups in the type.

2.2.1 `@Regex(groups)` *Type Hierarchy*

Moving down the qualifier hierarchy of Figure 2 are `@Regex` annotations with increasing group counts. Any regular expression that has at least 3 groups also has at least 1 group, so the regular expression has both the types `@Regex(3)` and `@Regex(1)`.

As a concrete example, consider this code from Lucene/solr/src/-java/org/apache/solr/core/Config.java:

```
  parsedMatchVersion.replaceFirst("^(\\d)\\.(\\d)\$",
    "LUCENE_\$1\$2");
```

The signature of `String.replaceFirst` is `replaceFirst(@Regex String regex, String replacement)`. In the above call, the first argument has type `@Regex(2)`. The call is permitted because `@Regex(2)` is a subtype of `@Regex`.

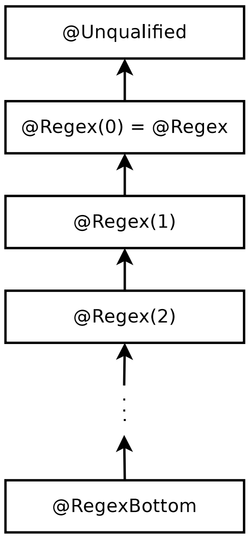

Figure 2: The full subtyping relationship of the type system's qualifiers. Because the parameter to a `@Regex` qualifier is a lower bound on the number of capturing groups in a regular expression, `@Regex` qualifiers with more capturing groups are subtypes of `@Regex` annotations with fewer capturing groups. `@Regex(0)` and `@Regex` are synonyms, because every regular expression has at least zero capturing groups.

2.2.2 `@RegexBottom` *Qualifier*

The `@RegexBottom` qualifier is used only for the `null` literal and is a subtype of all other qualifiers in the type system; this allows the `null` literal to be assigned to any variable whose type has any qualifier. Therefore, a `NullPointerException` is possible when invoking `Pattern.compile` — our type system verifies only correct use of regular expression syntax. A programmer can use it in conjunction with a type system for detecting null pointer exceptions [27, 11].

3. REGEX CHECKER

The type system described in Section 2 is implemented as a pluggable type system called the Regex Checker. It is built on and distributed with the Checker Framework [27, 11, 7]. The implementation consists of 605 non-comment, non-blank lines of Java code, of which 263 lines contain only an `import` statement, annotation (`@Override`, etc.), or curly brace (`{`, `}`).

This section describes additional features that are provided by the implementation.

3.1 Defaulting

As a convenience to developers, the Regex Checker adds default qualifiers to types, preventing the need to explicitly qualify each type. Except as described in this section, all unannotated types are treated as `@Unqualified`.

The Regex Checker implicitly adds a `@Regex` annotation with the correct capturing group count to any `String` literal that is a syntactically valid regular expression. It does so by analyzing the literal string at compile time.

The Regex Checker also implicitly adds a `@Regex` annotation with the correct capturing group count to a concatenation of syntactically valid regular expressions. When two `@Regex String`s are concatenated, the result is a `@Regex String` with a group count that is the sum of the initial two group counts.

The @Regex annotation on a String is propagated from the String to the Pattern produced by compiling the String and to the Matcher produced by matching the Pattern. Because the @Regex annotation is propagated to the Matcher, the Regex Checker can verify that calls to the group method are passed valid capturing group numbers.

Here is an example that illustrates these features. The commented type annotations are implicitly added by the Regex Checker and need not be written by a programmer.

```
/*@Regex(2)*/ String regex2 = "(.*) ([0-9]+)";
/*@Regex(3)*/ String regex3 = regex2 + "(.*)";
/*@Regex(3)*/ Pattern p = Pattern.compile(regex3);
/*@Regex(3)*/ Matcher m = p.match("number 447a");
if (m.matches()) {
    System.out.println(m.group(3)); // output: a
    // compile-time error, m has type @Regex(3)
    System.out.println(m.group(4));
}
```

3.2 Validating User Input

An application that reads a regular expression from an external source (such as user input or a file) must validate it before using it. The Regex Checker supports this use case, and in fact ensures that the developer does not forget it, preventing potential run-time errors. For example, consider this code:

```
String s = getInputFromUser();
Pattern p = Pattern.compile(s);
```

If the user input is not a valid regular expression, this code throws a run-time exception, which may terminate the application with a confusing and unprofessional stack trace. If the user input happens to be a valid regular expression (say, it is a string that contains a period ".") , then the application will not crash, but will produce different results than the user may have intended.

The Regex Checker issues a compile-time error for this code, because s cannot be guaranteed to be a regular expression.

The Regex Checker supplies helper routines to aid the developer in doing the necessary input checking. The most relevant is RegexUtil.isRegex(String), which returns true if its argument is a valid regular expression. There are also versions of the helper routines that take an additional capturing group count parameter. The Regex Checker flow-sensitively refines a variable's type when isRegex is used in a conditional [27]. See the Regex Checker documentation for details [7].

Depending on the programmer's intention, two better ways to write the above code are:

```
String s = getInputFromUser();
// At this point, the type of s is "@Unqualified String".
if (!RegexUtil.isRegex(s)) {
    ... issue user-friendly message about invalid input
} else {
    // At this point, the type of s is "@Regex String",
    // so there is no compile-time error.
    Pattern p = Pattern.compile(s);
}
```

and:

```
String s = getInputFromUser();
// At this point, the type of s is "@Unqualified String".
if (!RegexUtil.isRegex(s)) {
    ... issue user-friendly message about invalid input
    and terminate
}
// At this point, the type of s is "@Regex String",
// so there is no compile-time error.
Pattern p = Pattern.compile(s);
```

3.3 Partial Regular Expressions

When two regular expressions are concatenated, the result is a regular expression, and the Regex Checker supports this case. Sometimes, concatenation of *non*-regular-expression strings yields a regular expression. The most common case is embedding a legal regular expression between string literals that are not themselves regular expressions. The Regex Checker also supports this case. This is illustrated by the following example, taken from Lucene/solr/src/java/org/apache/solr/spelling/SpellingQueryConverter.java:

```
final static @Regex String NMTOKEN;

... // NMTOKEN assigned here

final static @Regex String PATTERN =
     "(?:(?!(" + NMTOKEN + ":|\\d+)))[\\p{L}_\\-0-9]+";
```

The Regex Checker handles this concatenation pattern by substituting an arbitrary regular expression (say, "e") for the @Regex String variable, then analyzing the resulting string. The string literals balance one another's parentheses (and NMTOKEN does not introduce further unbalanced parentheses), so variable PATTERN is a valid regular expression.

In general, for a string literal that is not a valid regular expression (for example, a string literal that has unbalanced parentheses) the Regex Checker stores the value of the string literal in a qualifier on the type. The Regex Checker can then use this stored value to determine if a concatenation results in a string that is a valid regular expression. If the concatenation does result in a valid regular expression a @Regex qualifier is added to the type. Otherwise, the value stored in the qualifier is updated to reflect the concatenation.

3.4 Pattern.LITERAL Flag

So far, we have assumed that Pattern.compile requires a regular expression as its argument. But, when invoked with Pattern.-LITERAL as its second argument, as in:

```
Pattern.compile(anyString, Pattern.LITERAL);
```

the first argument does not have to be a syntactically valid regular expression, because the Pattern.compile method escapes it before it is compiled.

The dependent type [34] of Pattern.compile is not expressible in our type system, so the Regex Checker has a special case for the Pattern.compile method when called with the Pattern.-LITERAL flag allowing the Pattern.compile method to take any String.

3.5 @Regex char

The Regex Checker also supports the use of chars as regular expressions. For example, the char 'c' is a syntactically valid regular expression and will be implicitly annotated as such by the Regex Checker. Concatenation works the same way with chars as it does with Strings.

3.6 Library Annotations

We added 18 @Regex annotations to library methods that take or return regular expressions. The library annotations are trusted, not checked.

Some of the more interesting annotated library methods are shown below.

```
class Pattern {
    public static Pattern compile(@Regex String regex);
    // escapes and returns argument
    public static @Regex String quote(String s);
}

class String {
    public boolean matches(@Regex String regex);
    public String replaceAll(@Regex String regex,
```

Application	NCNB LOC	@Regex annotations	Bugs	False positives
Apache Chukwa	37k	45	7	3
Apache Lucene	154k	11	6	5
Daikon	166k	8	6	1
Plume-lib	13k	40	2	9
YaCy	112k	71	35	7
TOTALS	482k	175	56	25

Figure 3: Case study statistics. NCNB LOC is non-comment, non-blank lines of code. `@Regex` annotations is the number of annotations we added to each application. Bugs is the number of bugs we found in each application by using the Regex Checker (see Section 4.2). False positives are caused by a weakness in either the type system or the Regex Checker implementation; we suppressed these by writing a `@SuppressWarnings` annotation (see Section 4.3).

```
                          String replacement);
    public String[] split(@Regex String regex);
}
```

The annotations permit the Regex Checker to verify correct usage of these methods.

4. CASE STUDIES

4.1 Methodology

We evaluated the Regex Checker using the applications of Figure 3, all of which are mature and under active maintenance. Apache Chukwa [1] monitors distributed systems. Apache Lucene [2] is a search engine library. Daikon [10, 13, 14] is a dynamic invariant detector. Plume-lib [28] is a utility library. YaCy [35] is a peer-to-peer search engine.

Our methodology was to repeatedly run the Regex Checker, examine its warnings, and add `@Regex` annotations where necessary, until the Regex Checker issued no more warnings. Along the way, we sometimes had to correct errors that the Regex Checker had discovered in the subject applications (Section 4.2), or suppress false positive warnings (Section 4.3).

4.2 Bugs

Overall we found 56 previously-unknown bugs in the five applications. We found two types of bugs: bugs caused by failure to validate or escape input before using it as a regular expression (see Section 4.2.1) and bugs for incorrect quoting of input that is used in a regular expression (see Section 4.2.2).

We found no bugs due to string literals that were syntactically invalid regular expressions, nor from invalid use of capturing groups (see Figure 4 for statistics about use of capturing groups). We hypothesize that this is because such bugs would be revealed during testing. However, the Regex Checker would be useful during development in finding these bugs at compile time, instead of waiting for tests to find these bugs at run time.

4.2.1 Non-validated Input

50 of the 56 total bugs were caused by failure to validate or escape input before using it as a regular expression. These bugs could lead to run-time errors from invalid regular expression syntax, or to silently incorrect behavior.

Here is an example bug, from class Lookup in Plume-lib:

```
for (String keyword : keywords) { // command-line arguments
    ...
    // embed argument in regular expression
    keyword = "\\b" + keyword + "\\b";
    // compile regular expression
    if (Pattern.compile(keyword).matcher(search).find())
        ...
}
```

The Regex Checker reports an error at `Pattern.compile(keyword)`, since `keyword`, which was extracted from the command-line arguments, cannot be guaranteed to be a syntactically valid regular expression. When informed of the bug, the developer changed the code to quote `keyword` (by calling the `Pattern.quote` method to escape all special regular expression characters) before embedding it in a regular expression.

Here is another example, from class TsProcessor in Chukwa:

```
datePattern = jobConf.get("TsProcessor.time.regex." +
     chunk.getDataType(), datePattern);
...
Pattern pattern = datePattern != null ?
     Pattern.compile(datePattern) : null;
```

In this example, a regular expression is retrieved from a configuration file. Its syntax needs to be verified before use. Additionally, after this regular expression is compiled the first capturing group is extracted from text that matches this regular expression. That shows that this input does need to be a valid regular expression and should not be quoted. The problem can be eliminated as follows:

```
datePattern = jobConf.get("TsProcessor.time.regex." +
     chunk.getDataType(), datePattern);
// Verify datePattern is a legal regular expression
// with at least 1 capturing group.
if (!RegexUtil.isRegex(datePattern, 1)) {
   ... inform user of invalid value in configuration file
}
...
Pattern pattern = datePattern != null ?
     Pattern.compile(datePattern) : null;
```

In all of these 50 bugs, if the application was supplied with invalid regular expression input, the application would throw a `PatternSyntaxException` and eventually print a stack trace. This is not very helpful to a user of the application. Requiring the developer to validate input also encourages the developer to produce a useful diagnostic message highlighting the specific value in a configuration file that is an invalid regular expression. This clarifies that the user has mis-used the application, and makes it much easier for the user to fix the error in the regular expression.

4.2.2 Incorrect Quoting

The remaining 6 bugs (all in YaCy) were when input was incorrectly quoted before being used as a regular expression. The following example is from class Load_RSS_p:

```
String messageurl;
...
messageurl = row.get("url", "");
if (r == null || !r.get("comment", "").matches(
   ".*\\Q" + messageurl + "\\E.*")) {
...
```

In a regular expression, any text between "`\Q`" and "`\E`" is quoted.[1] However, if the `messageurl` variable itself contained a `"\E"` the quoting would stop earlier than the developer intended, the regular expression would then have an extra `"\E"`, and the application would terminate with a `PatternSyntaxException`. This type of bug appeared six times in YaCy. Instead of using the `"\Q"` and `"\E"` constructs to quote the input, the `Pattern.quote` method should be called:

```
String messageurl;
```

[1]The `String` representation of a backslash is two backslashes; `"\\Q"` is a two-character string containing one backslash and one Q.

	Number of calls	
Method	Total	With `int` literal parameter
`group`	172	171
`start`	3	1
`end`	8	6
Total	183	178

Figure 4: Use of `int` literals in calls to `Matcher`'s methods. For the Regex Checker to verify the correct parameter to a method that takes a capturing group number, the parameter must be a compile-time constant. Of the 183 total calls, only 5 are not passed `int` literals. 3 of the 5 calls that received a non-`int`-literal were in deprecated code and were handled by a single warning suppression (see Figure 5).

Category	Reason	No.
Type system	Substring operation	3
	Variable group count	2
	String represents a character class	1
Subject application	Tests whether s is a regex	8
	Deprecated code	1
Regex Checker implementation	StringBuilder, StringBuffer, char[]	8
	flow-sensitivity bug	1
	line.separator property is a legal regex	1
	Total	25

Figure 5: False positives. Those in the top part are weaknesses of our type system (Section 4.3.1). Those in the middle part could be eliminated by improving the code of the subject applications (Section 4.3.2). Those in the bottom part are weaknesses of our implementation (Section 4.3.3).

```
...
messageurl = row.get("url", "");
if (r == null || !r.get("comment", "").matches(
    ".*" + Pattern.quote(messageurl) + ".*")) {
...
```

4.2.3 *Bug Reports*

The Daikon, Plume-lib, and YaCy developers have fixed all the bugs in their applications, added our annotations to their source code, and are now regularly running the Regex Checker to prevent similar problems in the future. The Chukwa developers have acknowledged and fixed the first bug we reported. The Lucene developers have not yet responded to our bug reports.

4.3 False Positives

The Regex Checker issued 25 false positive warnings in our case studies (Figure 5). In each of these cases, we manually confirmed that the code will never throw an exception at run time, but the Regex Checker was unable to verify that fact. We now discuss the false positives, categorized according to whether the fault lies with our type system (Section 4.3.1), programming paradigms used in the subject application (Section 4.3.2), or the Regex Checker implementation of the type system (Section 4.3.3).

4.3.1 *Type system false positives*

Like every type system, ours reasons about only certain abstractions. This section gives examples of code whose correctness cannot be represented in our type system.

Even if a string is known to be a regular expression, its substrings may not be, and nothing is known about substrings of arbitrary strings. Consider the following substring operation in Chukwa/trunk-/src/main/java/org/apache/hadoop/chukwa/database/Macro.java:

```
public String computeMacro(@Regex String macro) {
  ...
  // first arg to findTableName must have type @Regex String
  table = dbc.findTableName(
    macro.substring(macro.indexOf("(") + 1,
                    macro.indexOf(")")),
    ...);
```

The `computeMacro` method is always called with a String such as `"group_avg(user_util)"`, which is a valid regular expression. Furthermore, the substring call (which extracts the text in the parentheses of the input) always returns a regular expression. The Regex Checker is unable to establish this fact and issues a warning that the the developer must suppress.

The "variable group count" false positives in Figure 5 are ones where the capturing-group argument to a `Matcher` method is a variable (see Figure 4). The Regex Checker does not currently track the ranges of variable values. Furthermore, in some cases a method takes both a regular expression and a group number as an argument. A dependent type system [34] could express that a variable is a valid capturing group number for a regular expression, but our current type system does not. (Our type system also cannot express the precise type of a function that takes any regular expression, and returns a regular expression with one more capturing group by concatenating one more capturing group. No such example came up in our case studies.)

A final false positive is for a computed string that gets embedded in a character class ("`[...]`"). It would be possible to define another type qualifier (orthogonal to `@Regex`) to express legal character class constants.

4.3.2 *Subject application false positives*

Some of the subject applications had their own versions of methods such as `isRegex` (see Section 3.2). The Regex Checker cannot reason about the complex algorithms that quote and test regular expressions. These false positives could be eliminated by using the Regex Checker's own `RegexUtil` methods, which are already correctly annotated and need not be checked along with the user code.

One other false positive was in deprecated code. The developers acknowledged the bad design in this obsolete code, but declined to modify the code.

4.3.3 *Implementation false positives*

The Regex Checker issues some false positives that we intend to correct by improving its implementation.

The Regex Checker currently supports annotations on `String`, `char`, `Pattern`, and `Matcher` types. This should be extended to `char[]`, `StringBuilder`, and `StringBuffer`. For example, here is the proper annotation for this code from Daikon/asm/Operand.java:

```
@Regex StringBuilder b = new StringBuilder();
b.append("eax");
b.append("|ebx");
b.append("|ecx");
b.append("|edx");
...
registers32BitRegExp = Pattern.compile(b.toString());
```

Because the Regex Checker does not support `@Regex StringBuilder`, that annotation must be omitted. As a result, the Regex Checker is unable to determine that `b.toString()` returns a `@Regex String`, and so the Regex Checker issues an error on the call to `Pattern.compile`.

There is an imprecision in the Checker Framework's flow-sensitive type refinement [27] that results in one false positive. We observed the bug when there is an `if` followed by an `else if`, the bodies of both modify program control in some way (for example a `throws` or `continue` statement), and at least one of them includes a call to the `isRegex` method (described in Section 3.2) The code that triggered this bug, from Lucene/solr/src/java/org/apache/solr/-analysis/PatternTokenizerFactory.java, is shown below.

```
String regex = args.get( PATTERN );
if( regex == null ) {
   throw new SolrException( SolrException.ErrorCode.SERVER_ERROR,
       "missing required argument: "+PATTERN );
} else if (!RegexUtil.isRegex(regex, group)) {
   throw new SolrException( SolrException.ErrorCode.SERVER_ERROR,
       "error parsing regular expression "
       + regex + ": " + RegexUtil.regexError(regex));
}
pattern = Pattern.compile( regex, flags );
```

The Regex Checker reports a false positive warning at the call to `Pattern.compile` because it cannot determine that the `regex` variable is a valid regular expression at that location.

The final false positive comes from the fact that the return value of `System.getProperty("line.separator")` is always a legal regular expression (unless the application modifies it). This could be special-cased in a similar way as for `Pattern.LITERAL` in Section 3.4; other pluggable type-checkers built on the Checker Framework have similar special cases.

4.4 Discussion

The first author performed the annotation and type-checking. He found the task simple despite his unfamiliarity with the subject applications. Most parts of the code required no annotations, and the Regex Checker highlighted the places that did. The only difficulty was caused by code with poor design, such as embedding regular expression `String`s in other `String`s and then extracting them later, or storing regular expression `String`s in the same data structures as other `String`s. If the original programmers had used a regular expression verification system, they might have been guided to a cleaner design. We also found use of the `isRegex` method cleaner than other mechanisms used in the applications, such as catching `PatternSyntaxException`.

In addition to detecting 56 bugs, the `@Regex` annotations improve the code's documentation. In many cases, the developer-written documentation had omitted to state the requirement that a given parameter be a regular expression. By contrast, the `@Regex` annotations are machine-verified.

Relatively simple ad-hoc checks would have sufficed to detect many of these bugs. However, a significant benefit of our approach is that it gives a guarantee that no regular expression syntax errors exist anywhere in the application (modulo human inspection at each location where the type checker issues a warning).

5. RELATED WORK

We are not aware of any previous type system nor other formal analysis that helps programmers to verify regular expression syntax.

A simple way to guarantee correct regular expression syntax is to force programmers to create objects, rather than strings, representing the regular expression. Any syntax error is caught at compile time. However, such a syntax is more verbose and less readable than a string representation. Even a language like Haskell that encourages such a style still contains a `Text.Regex.Posix` regular expression library that parses regular expressions at run time. Mainland [25] proposes an analysis that detects errors by parsing the regular expression string at compile time. This only works when the `mkRegex` function is invoked on a string literal. Other work in the functional programming community assumes syntactic correctness. For example, Broberg et al.'s static semantics for "wellformedness of regular expression patterns" actually confirms variable binding and types [4]. Likewise, Fisher and Shivers's static analysis for syntax objects is proposed to be applied to a known, valid regular expression by converting it into a FSM rendered in the C programming language, then analyzing the C program [16].

Regular expression patterns [18] are proposed by the programming language community as an alternative to XPath for specifying elements and attributes in an XML document. Both can be viewed as a variety of regular expression. The existing analyses are complementary to ours, in that they assume correct syntax and try to determine semantic errors, such as subpatterns that will never match [3, 6]. In Wilk and Drabant's type system for Xcerpt, if the application type-checks, it passes a set of test cases; that is, its results are in a given expected set [33].

A more heavy-weight approach than ours is to analyze an application to determine all the strings that it can generate, represented as an automaton [32] or a grammar [26]. This has been most often applied to detection (or generation [23]) of SQL injection attacks, and can be augmented by external string solvers [22]. Regular expressions are often used by the analysis to describe the legal strings, but none of this work addresses applications that use regular expressions. String analysis has also been used to verify sanitization routines, such as to confirm that every blacklisted character has been removed from a given string [36, 5, 31]; such work is complementary to ours.

Our type `Regex(`n`)` could be expressed as a dependent type [34], if n can refer to another program variable. Such a feature was not necessary in our case studies.

5.1 Other practical tools

The FindBugs [15, 19, 20] tool has three bug checks for regular expressions. (1) It reports invalid regular expression syntax in compile-time constants that are passed to `Pattern.compile`. In our case study, the subject applications contained no errors of this variety. By contrast, the Regex Checker is a type system that applies to all `String`s. (2) `File.separator` cannot be used as a regular expression (on Windows). This special case is not needed in the Regex Checker. (3) `"."` may not be used as as regular expression, because a programmer might have intended to escape the period. The Regex Checker does not contain such a check. Overall, FindBugs would find no errors in the subject applications used in our case study.

Spinellis and Louridas [30] extended FindBugs to better verify API call arguments, and found that FindBugs "contained a number of tests for the replace methods that were incorrect".

These other popular Java static checking tools do no checking of regular expressions: Checkstyle [8], Hammurapi [17], JLint [21], Macker [24], and PMD [29, 9].

6. CONCLUSION

We believe that formal methods and verification are important enough to be applied to all aspects of an application. This includes regular expression syntax, which may seem mundane but is of significant importance in practice. Our experience with real-world code indicated both necessary and unnecessary features in a regular expression type system, several of which were not obvious beforehand. Our Checker Framework enables new type systems to be created easily and succinctly, and to be scaled to arbitrary power. This satisfies an essential condition of practicality: that new formal techniques must be easy to implement. Our implementation is publicly available [7].

In case studies on five Java applications, our regular expression type system found 56 new bugs. It was easy to use because it requires low overhead and little understanding of the code being annotated. The type system improves documentation and encourages good design.

7. REFERENCES

[1] Apache Chukwa website. `http://incubator.apache.org/chukwa/`.

[2] Apache Lucene website. `http://lucene.apache.org/`.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP*, pages 51–63, 2003.

[4] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *ICFP*, pages 67–78, 2004.

[5] R. Bubel, R. Hähnle, and U. Geilmann. A formalisation of Java strings for program specification and verification. In *SEFM*, pages 90–105, 2011.

[6] G. Castagna, D. Colazzo, and A. Frisch. Error mining for regular expression patterns. In *Theoretical Computer Science*, volume 3701 of *LNCS*, pages 160–172. Springer, 2005.

[7] Checker Framework website. `http://types.cs.washington.edu/checker-framework/`.

[8] Checkstyle website. `http://checkstyle.sourceforge.net/`.

[9] T. Copeland. *PMD Applied*. Centennial Books, Nov. 2005.

[10] Daikon website. `http://groups.csail.mit.edu/pag/daikon/`.

[11] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE*, pages 681–690, May 2011.

[12] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, Oct. 2011.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 69(1–3):35–45, Dec. 2007.

[15] FindBugs website. `http://findbugs.sourceforge.net/`.

[16] D. Fisher and O. Shivers. Static analysis for syntax objects. In *ICFP*, pages 111–121, 2006.

[17] Hammurapi website. `http://www.hammurapi.biz/`.

[18] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 226–244, 2001.

[19] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, pages 132–136, Oct. 2004.

[20] D. Hovemeyer and W. Pugh. Status report on JSR-305: Annotations for software defect detection. In *OOPSLA Companion*, pages 799–800, Oct. 2007.

[21] JLint website. `http://jlint.sourceforge.net/`.

[22] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *ISSTA*, pages 105–116, July 2009.

[23] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, pages 199–209, May 2009.

[24] Macker website. `http://innig.net/macker/`.

[25] G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *ACM SIGPLAN workshop on Haskell*, pages 73–82, 2007.

[26] Y. Minamide. Static approximation of dynamically generated web pages. In *World Wide Web*, pages 432–441, 2005.

[27] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[28] Plume-lib website. `http://plume-lib.googlecode.com/`.

[29] PMD website. `http://pmd.sourceforge.net/`.

[30] D. Spinellis and P. Louridas. A framework for the static verification of API calls. *Journal of Systems and Software*, 80(7):1156 – 1168, 2007.

[31] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *ISSTA*, pages 166–176, 2011.

[32] G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[33] A. Wilk and W. Drabent. A Prototype of a Descriptive Type System for Xcerpt. In *Principles and Practice of Semantic Web Reasoning*, volume 4187 of *LNCS*, pages 262–275. Springer, 2006.

[34] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, Jan. 1999.

[35] YaCy website. `http://yacy.net/en/`.

[36] F. Yu, T. Bultan, M. Cova, and O. Ibarra. Symbolic string verification: An automata-based approach. In *Model Checking Software*, volume 5156 of *LNCS*, pages 306–324. Springer, 2008.