Static Program Checking

Bounded Verification – Other Ideas

Juniorprof. Dr. Mana Taghdiri

Thursday – June 26, 2014
Incremental bounded verification

Problems of bounded verification:
- The formulas generated for non-trivial programs are complex
- They often choke the solver
  - When the solver times out, there’s no feedback (on coverage of the analysis or likelihood of correctness)

Solution:
- Divide the program into several sub-programs
- Check the property in each sub-program
  - Hopefully each sub-program generates a smaller sub-formula

Approach:
- Can partition the program based on control flow
- Or based on data flow (variable definitions)
Program partitioning

- Proposed for bounded executions
  - Loops are unrolled
- Partition the set of program paths to multiple subsets:

  $$\text{path(Proc)} = \bigcup_{i=1}^{n} \text{path(Sub}_i)$$

- Then, instead of checking $\text{Pre} \land \text{translate(Proc)} \land \neg \text{Post}$
- We can check

  $$\{\text{Pre} \land \text{translate(Sub}_1) \land \neg \text{Post}\} \land \ldots \land \{\text{Pre} \land \text{translate(Sub}_n) \land \neg \text{Post}\}$$
Partitioning based on control flow

- Splitting algorithm is based on vertices of the computation graph
- Given a vertex, construct two subgraphs
  - Go-through subgraph
  - Bypass subgraph

- Rationale
  - Number of branches is a heuristic metric for complexity
  - Pick a vertex that results in subgraphs with fewer branches

- The splitting can be done recursively as much as desired
Example

class IntList {
   Entry header;
   class Entry {
      int value;
      Entry next;
   };

   boolean contains(int key) {
      Entry e = this.header;
      while (e != null) {
         if (e.value == key)
            return true;
         e = e.next;
      }
      return false;
   }
}
Example after two loop unrollings

```java
public boolean contains(int key)
{
    1: Entry e = this.header;
    2: if (e != null){
    3:     if (e.value == key){
    4:         return true;
    }
    5:     e = e.next;
    6:     if (e != null){
    7:         if (e.value == key){
    8:             return true;
    }
    9:     }
    10:     assume(e == null);
    11:     return false;
    0 :}
```
Partition based on node 11

```java
public boolean
go-through(int key)
{
    1: Entry e = this.header;
    2: if (e != null){
        3': assume !(e.value==key);
    4:

    5: e = e.next;
    6: if (e != null){
        7': assume!(e.value==key);
    8:

    9: e = e.next;
    10: assume(e == null);
    11: return false;
    0:}
```

Gray: branch converted to assume
Black: removed statements
Partition based on node 11

```java
public boolean bypass(int key)
{
  1 : Entry e = this.header;
  2' : assume(e != null);
  3 : if (e.value == key){
  4 :    return true;
  }
  5 : e = e.next;
  6' : assume (e != null);
  7" : assume(e.value == key);
  8 : return true;
  
  9 :
  10:

  11:
  0 :}
```

Gray: branch converted to assume
Black: removed statements
Data flow partitioning

- Control-flow partitioning
  - Is limited to syntactical structure of program
  - Doesn’t exploit program semantics

- Data-flow partitioning is based on variable-definitions
  - Fewer definitions of a variable result in fewer intermediate variables
  - Thus, reduces the number of frame conditions encoding data flow
  - Thus, there are fewer variables in the resulting formula
  - (uses a Jalloy-like translation of code)

- Pick a variable in the computation graph
  - Split the graph into multiple subgraphs s.t. each subgraph has at most one definition for that variable, that can reach the exit statement
  - The definition of this variable is different in each subgraph
Example after two loop unrollings

```java
public boolean contains(int key) {
  1: Entry e = this.header;
  2: if (e != null) {
  3:   if (e.value == key) {
  4:     return true;
  }
  5:   e = e.next;
  6:   if (e != null) {
  7:     if (e.value == key) {
  8:       return true;
  }
  9:   e = e.next;
  10:   assume(e == null);
  11: return false;
  0:
```

Definition set of this = {}
Definition set of key = {}
Definition set of return = {4, 8, 11}
Definition set of e = {1, 5, 9}
All of these definitions can reach the exit statement
Splitting based on “e”

```java
public boolean sub1(int key)
{
1: Entry e = this.header;
2: if (e != null){
3': assume (e.value==key)
4: return true;

5 :
6 :
7 :
8 :
9 :
10:
11: return false;
0 :}
```

Now we have exactly one definition of e (line 1) (doesn’t include 5 or 9)
Set the branch conditions s.t. unwanted nodes are not visited
Splitting based on “e”

```
public boolean
sub2(int key)
{
1 : Entry e = this.header;
2' : assume (e != null);
3' : assume !(e.value==key);
4 :
5 : e = e.next;
6 : if (e != null){
7' : assume(e.value==key);
8 : return true
}
9 :
10 :
11: return false;
0 :)
```

Again exactly one definition of e reaches exit (line 5) 
(1 or 9 can’t reach the exit)
Splitting based on “e”

```java
public boolean sub3(int key) {
    1: Entry e = this.header;
    2': assume(e != null);
    3": assume !(e.value==key);
    4 :

    5 : e = e.next;
    6': assume (e != null);
    7": assume !(e.value==key);
    8 :

    9 : e = e.next;
    10: assume(e == null);
    11: return false;
    0 :}
```

Only the definition in line 9 reaches exit
Discussion

- Limited experiments done so far

- Substantial speedup in higher scopes (around 6, 7)
  - Two rounds of splitting

- Small speedup when the complexity of the specification is more than the code formula
  - The benefit will be reduced by the overhead of multiple checking

- Because sub-graphs are independent, they can be checked in parallel
ESC/Java

- Extended Static Checker for Java
  - Finds common programming errors (not a prover!)
  - Compile-time checker
    - Catches more errors than a typical type checker
    - Examples:
      - Null dereference, array out of bound, type cast error
    - Examples of concurrent problems:
      - Race conditions
      - Deadlocks
      - Can also check user-defined design decisions (pre/post conditions)
  - Based on
    - Verification-condition generation
    - Automatic theorem proving
- Uses its own annotation language
ESC/Java features

- ESC/Java is modular
  - Operates on one procedure at a time
  - Advantage: scalability
  - Disadvantage: user-provided annotations

- Is more lightweight than a full verification tool
  - Annotations are smaller

- Has to make a trade-off between
  - Missed errors (unsoundness)
  - False alarms (incompleteness)
  - Annotation overhead
  - Performance
Running ESC/Java – example

class Bag {
    int size;
    int[] elements; // valid: elements[0..size-1]

    Bag(int[] input) {
        size = input.length;
        elements = new int[size];
        System.arraycopy(input, 0, elements, 0, size);
    }

    int extractMin() {
        int min = Integer.MAX_VALUE;
        int minIndex = 0;
        for (int i = 1; i <= size; i++) {
            if (elements[i] < min) {
                min = elements[i];
                minIndex = i;
            }
        }
        size --;
        elements[minIndex] = elements[size];
        return min;
    }
}
Running ESC/Java – example

```java
1: class Bag {
2:     int size;
3:     int[] elements; // valid: elements[0..size-1]
4: }
5: 
6: Bag(int[] input) {
7:     size = input.length;
8:     elements = new int[size];
9:     System.arraycopy(input, 0, elements, 0, size);
10: }
11: 
12: int extractMin() {
13:     int min = Integer.MAX_VALUE;
14:     int minIndex = 0;
15:     for (int i = 1; i <= size; i++) {
16:         if (elements[i] < min) {
17:             min = elements[i];
18:             minIndex = i;
19:         }
20:     }
21:     size --;
22:     elements[minIndex] = elements[size];
23:     return min;
24: }
```

Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;

Bag.java:15: Warning: Possible null dereference (Null)
if (elements[i] < min) {

Bag.java:15: Warning: Array index possibly too large (... if (elements[i] < min) {

Bag.java:21: Warning: Possible null dereference (Null)
elements[minIndex] = elements[size];

Bag.java:21: Warning: Possible negative array index (... elements[minIndex] = elements[size];

Bag.java:21: Warning: Possible null dereference (Null)
elements[size] = elements[size];
Running ESC/Java – example

```java
1: class Bag {
2:     int size;
3:     int[] elements; // valid: elements[0..size-1]
4: 
5:     Bag(int[] input) {
6:         size = input.length;
7:         elements = new int[size];
8:         System.arraycopy(input, 0, elements, 0, size);
9:     }
10: 
11:     int extractMin() {
12:         int min = Integer.MAX_VALUE;
13:         int minIndex = 0;
14:         for (int i = 1; i <= size; i++) {
15:             if (elements[i] < min) {
16:                 min = elements[i];
17:                 minIndex = i;
18:             }
19:         } 
20:         size--; 
21:         elements[minIndex] = elements[size]; 
22:         return min; 
23:     }
24: }
```

Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;

Bag.java:15: Warning: Possible null dereference (Null)
if (elements[i] < min) {

Bag.java:15: Warning: Array index possibly too large (... 
if (elements[i] < min) {

Bag.java:21: Warning: Possible null dereference (Null)
elements[minIndex] = elements[size];

Bag.java:21: Warning: Possible negative array index (... 
elements[minIndex] = elements[size];

1 needs a pre-condition for constructor (or fixing the code)

`4a: //@ requires input != null`
Running ESC/Java – example

2 and 4 are there because elements is not private
– making it private doesn’t remove warnings
– ESC can’t check all methods to ensure elements is not assigned null
– (it’s modular)
Running ESC/Java – example

```java
1: class Bag {
2:     int size;
3:     int[] elements; // valid: elements[0..size-1]
4: }
5: Bag(int[] input) {
6:     size = input.length;
7:     elements = new int[size];
8:     System.arraycopy(input, 0, elements, 0, size);
9: }
10: int extractMin() {
11:     int min = Integer.MAX_VALUE;
12:     int minIndex = 0;
13:     for (int i = 1; i <= size; i++) {
14:         if (elements[i] < min) {
15:             min = elements[i];
16:             minIndex = i;
17:         }
18:     }
19:     size --;
20:     elements[minIndex] = elements[size];
21:     return min;
22: }
```

Bag.java:6: Warning: Possible null dereference (Null)
size = input.length;
^ {highlighted}

Bag.java:15: Warning: Possible null dereference (Null)
if (elements[i] < min) {
   ^ {highlighted}

Bag.java:15: Warning: Array index possibly too large (... if (elements[i] < min) {
   ^ {highlighted}

Bag.java:21: Warning: Possible null dereference (Null)
elements[minIndex] = elements[size];
   ^ {highlighted}

Bag.java:21: Warning: Possible negative array index (... elements[minIndex] = elements[size];
   ^ {highlighted}

3 is because other code might mutate size

2a: //@ invariant 0 <= size && size <= elements.length {highlighted}
Running ESC/Java – example

```java
1: class Bag {
2:     int size;
3:     int[] elements; // valid: elements[0..size-1]
4:
5:     Bag(int[] input) {
6:         size = input.length;
7:         elements = new int[size];
8:         System.arraycopy(input, 0, elements, 0, size);
9:     }
10:
11:     int extractMin() {
12:         int min = Integer.MAX_VALUE;
13:         int minIndex = 0;
14:         for (int i = 1; i <= size; i++) {
15:             if (elements[i] < min) {
16:                 min = elements[i];
17:                 minIndex = i;
18:             }
19:         }
20:         size--;
21:         elements[minIndex] = elements[size];
22:         return min;
23:     }
24: }
```

Even with the invariant, it complains about index too large (line 15)
Running ESC/Java – example

```java
1: class Bag {
2:     int size;
3:     int[] elements; // valid: elements[0..size-1]
4: }
5: Bag(int[] input) {
6:     size = input.length;
7:     elements = new int[size];
8:     System.arraycopy(input, 0, elements, 0, size);
9: }
10: int extractMin() {
11:     int min = Integer.MAX_VALUE;
12:     int minIndex = 0;
13:     for (int i = 1; i <= size; i++) {
14:         if (elements[i] < min) {
15:             min = elements[i];
16:             minIndex = i;
17:         }
18:     }
19:     size --;
20:     elements[minIndex] = elements[size];
21:     return min;
22: }
23: }
```

Last warning: procedure can be called when bag is empty (size = 0)
Running ESC/Java – example – 2\textsuperscript{nd} run

```java
1: class Bag {
2:     int size;
3:     int[] elements; // valid: elements[0..size-1]
4: }
5: Bag(int[] input) {
6:     size = input.length;
7:     elements = new int[size];
8:     System.arraycopy(input, 0, elements, 0, size);
9: }
10: int extractMin() {
11:     int min = Integer.MAX_VALUE;
12:     int minIndex = 0;
13:     for (int i = 1; i <= size; i++) {
14:         if (elements[i] < min) {
15:             min = elements[i];
16:             minIndex = i;
17:         }
18:     }
19:     size--;
20:     elements[minIndex] = elements[size];
21:     return min;
22: }
```

```
Bag.java:26: Warning: Possible violation of object invariant
} ^
Associated declaration is "Bag.java", line 3, col 6:
// invariant 0 <= size && size <= elements.length ^
Possibly relevant items from the counterexample context:
brokenObj == this
(brokenObj* refers to the object for which the invariant is broken.)
```

```
19a: if (size > 0) {
20:    size--;
21:    elements[minIndex] = elements[size];
21a: }
```

Line 26 is the old line 20.
Size may become negative
Running ESC/Java – example – 3rd run

```java
class Bag {
    int size;
    int[] elements; // valid: elements[0..size-1]

    Bag(int[] input) {
        size = input.length;
        elements = new int[size];
        System.arraycopy(input, 0, elements, 0, size);
    }

    int extractMin() {
        int min = Integer.MAX_VALUE;
        int minIndex = 0;
        for (int i = 1; i <= size; i++) {
            if (elements[i] < min) {
                min = elements[i];
                minIndex = i;
            }
        }
        size--;  
        elements[minIndex] = elements[size];
    return min;
    }
}
```
What did we learn

- Warnings resulted in
  - One pre-condition (inputs != null)
  - Two rep invariants (on size and elements)
  - Two bug fixes (wrong index range, missing case of empty bag)

- Using pre-conditions:
  - When checking a procedure foo, assumes that its pre-conditions hold
  - When encountering a call to foo, checks whether the pre-conditions hold or not

- Using object invariants (rep invariants):
  - Assumes that they hold in the pre-state
  - Checks whether they hold in the post-state or not
Architecture

![Architecture Diagram]

Annotated Java Program

Front End

Abstract Syntax Trees (ASTs)

Translator

Guarded Commands (GCs)

VC Generator

Verification Conditions (VCs)

Theorem Prover

Prover Results

Postprocessor

Universal Background Predicate (UBP)

Output to User
Front-end

- Generates abstract syntax tree (AST)
- Generates type-specific background predicate
  - A formula in first-order logic
  - Generated for every class whose routines are to be checked
  - Encodes information about types and fields that routines use
  - Example: for a final class T
    - \( \forall S :: S <: T \Rightarrow S = T \)
Translation

- Generates Dijkstra’s guarded commands (GC)
- Insert commands of the form `assert E` (E is a boolean expression)
- Ideal translation of a procedure R is to get a guarded command G s.t.
  - If there is a way that R starts from a state satisfying its precondition and behave erroneously (violate post conditions), G has at least one execution that starts in a state satisfying the precondition and then violates some assertion
  - If there is no way that R can start from a state satisfying its precondition and then behave erroneously, then G has no execution that starts in a state satisfying the precondition and then violates some assertion

- ESC Translation is neither sound nor complete
  - Neither of the above conditions holds
Translation

Sources of inaccuracy:

- Modularity
  - replacing calls with specs (usually under-specifications). We may report a bug that is not feasible in the code
  - Especially for ESC/Java, the specs are lightweight, supposed to encode only as much as needed for analysis

- Overflow
  - We ignore arithmetic overflows. We may miss errors

- Loops
  - unroll them (misses errors that need more iterations)
  - asking for loop invariants is unrealistic for practical code
  - default is one unrolling, but user can provide more
VC generation

- Generates verification conditions for each guarded command $G$
  - Is a predicate in first-order logic that holds for exactly those program states from which no execution of the command $G$ goes wrong.
  - Computation similar to computing weakest pre-conditions + optimizations to avoid exponential blow-up

- An execution of a guarded command is said to “go wrong” if control reaches a subcommand of the form assert $E$ when $E$ is false
Thorem proving

- Uses Simplify
- Solves $\text{UBP} \land \text{BP}_T \Rightarrow \text{VC}_R$
  - UBP: universal background predicate
  - BP: type-specific background predicate
  - VC$_R$: verification condition for procedure R

- Universal background predicate
  - Encodes facts about the semantics of Java
  - E.g. that the subtype relation is reflexive, anti-symmetric, and transitive
Post-processing

- Takes the theorem prover’s output and generates warnings when proofs fail

- Simplify allows for
  - Labeled constraints
  - Can track back the source context corresponding to each constraint

- Since the formulas are in FOL (undecidable), the runtime of simplify is limited by some threshold
  - It might report something as a bug that could’ve been proved in longer time
  - (more false warnings)
Annotation language

- Similar to JML, but small differences
  - JML is intended for full specification of programs
  - ESC/Java is intended for lightweight specifications
  - So small syntactic and semantic differences

- Cost:
  - Mostly small annotations (argument non-null, etc.)
  - 40-100 annotations per 1000 LOC (4-10%)
  - In the experiments, they were inserted interactively:
    - First annotated based on a rough understanding of code
    - Then ESC/Java ran, then more annotations added
  - Expensive on users
  - Prohibitively costly when running ESC on existing codebase
Program verification

- Construct a logical formula whose solutions are executions of the code that violate the property \((f)\)
- Now solve \((f)\)

Either translate the code precisely, or ..
Modular analysis

- Replace a procedure with its specification
- Makes the technique better scalable
- But, is very costly for the user

Ask for user-provided annotations, or..
Specification inference

- User provides only the top-level property
- This substantially reduces the human cost

Infer intermediate annotations automatically
ESC/Java annotations

- Simplest annotation-based analysis
  - **Type checker** is a limited program analysis tool
  - It is modular and requires type annotations from users

- ESC/Java is like an advanced type checker
  - Checks for null-dereference, array bounds, etc.
  - So it doesn’t need extensive annotations like full verification
  - Still the amount of annotations can be up to 10% of the code size

- Houdini:
  - Generates intermediate annotations automatically
Generating candidates is done by looking at program text
It uses **heuristics** about what annotations might be useful
Example:
  - all preconditions of the form `argument != null`
Houdini – Annotation assistant for ESC/Java

Input: An unannotated program P  
Output: ESC/Java warnings for an annotated version of P

Algorithm:
- generate set of candidate annotations and insert into P;
- repeat
  - invoke ESC/Java to check P;
  - remove any refuted candidate annotations from P;
- until quiescence;
- invoke ESC/Java to identify possible defects in P;

- To identify incorrect annotations:
  - Invoke ESC/Java
  - Ignore warnings about runtime errors (e.g. null dereference)
  - If there is a warning about an annotation not true at some program point (e.g. a method’s precondition doesn’t hold at a call site), then remove that annotation from the candidate set
  - Removing one annotation may make others invalid, so repeat until fixpoint
Algorithm properties

- Remaining annotations are a subset of the initial candidate set
- Are guaranteed to be valid as much as ESC can tell
- They represent a maximal valid subset of the candidate set

- After the check-refute cycle, Houdini runs ESC/Java again
- This identifies potential run-time errors in the new annotated program
- These warnings are output to the user
Candidate annotations

- Ideally, the initial set must contain "all" possible annotations.
- But, the set cannot be too big because of performance.
- Following heuristics are based on experiments.
- For a field $f$, we generate the following invariants:

<table>
<thead>
<tr>
<th>Type of $f$</th>
<th>Candidate invariants for $f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>integral type</td>
<td><code>//@ invariant f cmp expr;</code></td>
</tr>
<tr>
<td>reference type</td>
<td><code>//@ invariant f != null;</code></td>
</tr>
<tr>
<td>array type</td>
<td><code>//@ invariant f != null;</code></td>
</tr>
<tr>
<td></td>
<td><code>//@ invariant \nonnullelements(f);</code></td>
</tr>
<tr>
<td></td>
<td><code>//@ invariant (\forall int i; 0 &lt;= i &amp;&amp; i &lt; expr</code></td>
</tr>
<tr>
<td></td>
<td><code>=&gt; f[i] != null);</code></td>
</tr>
<tr>
<td></td>
<td><code>//@ invariant f.length cmp expr;</code></td>
</tr>
<tr>
<td>boolean</td>
<td><code>//@ invariant f == false;</code></td>
</tr>
<tr>
<td></td>
<td><code>//@ invariant f == true;</code></td>
</tr>
</tbody>
</table>
Generated invariants

- Integral invariants
  - Mainly to check array index out of bound
  - Comparison operators: `<`, `<=`, `==`, `!=`, `>=`, `>`
  - Comparison expression:
    - an integer field declared earlier in the same class
    - Or an interesting constant: `-1, 0, 1, array dimensions (new int [4])`
  - Contradicting invariants are no problem \((x < 0 \text{ and } x \geq 0)\)
    - One of them gets refuted very fast

- Reference invariants
  - To check pointer `!=` null
  - Array pointers non-null
  - Array elements non-null
  - Array elements up to expr (a field or a constant) non-null
    - Useful in checking stack implemented by array
Other annotations

- **Candidate pre-conditions**
  - Comparison of two arguments
  - Relating an argument to a field declared in the same class
  - Also `//@requires false`
    - Any unfuted precondition of this form shows the procedure is never called
    - To identify dead code

- **Candidate post-conditions**
  - Relate the `\result` to an argument
  - Relate the `\result` to a field
  - Also `//@ensures \fresh(\result)`
    - That result is a newly allocated object
Experimental results

- Houdini is applied to a few programs of various sizes, up to 36kLOC
- It reduces the number of warnings of ESC/Java substantially

<table>
<thead>
<tr>
<th>Type of annotation</th>
<th>Preconditions guessed</th>
<th>% valid</th>
<th>Postconditions guessed</th>
<th>% valid</th>
<th>Invariants guessed</th>
<th>% valid</th>
<th>Total guessed</th>
<th>% valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>f == expr</td>
<td>2130</td>
<td>18</td>
<td>985</td>
<td>18</td>
<td>435</td>
<td>14</td>
<td>3550</td>
<td>17</td>
</tr>
<tr>
<td>f != expr</td>
<td>2130</td>
<td>35</td>
<td>985</td>
<td>35</td>
<td>435</td>
<td>38</td>
<td>3550</td>
<td>35</td>
</tr>
<tr>
<td>f &lt; expr</td>
<td>2130</td>
<td>26</td>
<td>985</td>
<td>27</td>
<td>435</td>
<td>24</td>
<td>3550</td>
<td>26</td>
</tr>
<tr>
<td>f &lt;= expr</td>
<td>2130</td>
<td>31</td>
<td>985</td>
<td>32</td>
<td>435</td>
<td>36</td>
<td>3550</td>
<td>33</td>
</tr>
<tr>
<td>f &gt;= expr</td>
<td>2130</td>
<td>25</td>
<td>985</td>
<td>21</td>
<td>435</td>
<td>19</td>
<td>3550</td>
<td>32</td>
</tr>
<tr>
<td>f &gt; expr</td>
<td>2130</td>
<td>31</td>
<td>985</td>
<td>36</td>
<td>435</td>
<td>35</td>
<td>3550</td>
<td>33</td>
</tr>
<tr>
<td>f != null</td>
<td>509</td>
<td>92</td>
<td>229</td>
<td>79</td>
<td>983</td>
<td>72</td>
<td>1721</td>
<td>79</td>
</tr>
<tr>
<td>\nonnulllelems(f)</td>
<td>54</td>
<td>81</td>
<td>21</td>
<td>62</td>
<td>36</td>
<td>64</td>
<td>111</td>
<td>72</td>
</tr>
<tr>
<td>(\forall ...)</td>
<td>841</td>
<td>27</td>
<td>260</td>
<td>37</td>
<td>125</td>
<td>59</td>
<td>1226</td>
<td>32</td>
</tr>
<tr>
<td>f == false</td>
<td>47</td>
<td>36</td>
<td>51</td>
<td>25</td>
<td>39</td>
<td>10</td>
<td>137</td>
<td>20</td>
</tr>
<tr>
<td>f == true</td>
<td>47</td>
<td>28</td>
<td>51</td>
<td>24</td>
<td>39</td>
<td>8</td>
<td>137</td>
<td>25</td>
</tr>
<tr>
<td>\fresh(\result)</td>
<td>0</td>
<td>0</td>
<td>229</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>229</td>
<td>30</td>
</tr>
<tr>
<td>false</td>
<td>780</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>780</td>
<td>17</td>
</tr>
<tr>
<td>exact type</td>
<td>37</td>
<td>19</td>
<td>11</td>
<td>36</td>
<td>14</td>
<td>57</td>
<td>62</td>
<td>31</td>
</tr>
<tr>
<td>Total</td>
<td>15095</td>
<td>30</td>
<td>6762</td>
<td>30</td>
<td>3846</td>
<td>40</td>
<td>25703</td>
<td>31</td>
</tr>
</tbody>
</table>