Mutual Recursion
• Handle recursion by performing explicit induction
• Handle mutual recursion by performing induction under assumptions
• Choice of assumptions: assume the negation of the (bounded) environment of procedures above the target
• Learn implications among procedure summaries — we call these Environment-Call (EC) lemmas

Call Graph and Bounded Environments

Exploring Procedures and Environments
• Consider each procedure in an environment to learn over- and under-approximate summaries
• Environments represent possible counterexample paths
• Choose a finite path through the call graph:
  • Final call is the target procedure to consider
  • The rest of the procedures in the path make up the environment

Scalability: Use Bounded Environments
• Longer call paths lead to larger queries and poor scalability
• Achieve scalability by approximating the environments
• A b-bounded environment captures at most the body of b procedures above the target procedure in the call graph path

Modular Verification
Goal: Infer procedure summaries that are...
• sufficient for verification
• efficiently computable
• sufficiently abstract and relevant to reason about the whole program

Challenges
• How should procedures be explored and in what environments (if any) should a procedure call be considered?
• How do we ensure procedure relevance and scalability?
• How would mutual recursion be handled?

Learning Procedure Summaries
Update Target’s Over-Approximate Summary
• Perform SMT check for over-approximation of procedure body and over-under-approximation of environment
• If unsatisfiable, learn potential counterexample behaviors in target
• With induction to handle recursion
• With assumptions to handle mutual recursion

Update Target’s Under-Approximate Summary
• Perform SMT check for under-approximation of procedure body and over-under-approximation of environment
• If unsatisfiable:
  • find interpolant I that separates the target and environment
  • learn interpolant as over-approximation

Experimental Results

Clover Spacer Eldarica Holce PCsat Ultimate Uniorhorn

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clover</th>
<th>Spacer</th>
<th>Eldarica</th>
<th>Holce</th>
<th>PCsat</th>
<th>Ultimate</th>
<th>Uniorhorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHC-Comp</td>
<td>81</td>
<td>93</td>
<td>94</td>
<td>92</td>
<td>81</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>Mutual</td>
<td>24</td>
<td>12</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Recursion</td>
<td>16</td>
<td>6</td>
<td>14</td>
<td>14</td>
<td>13</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Montgomery</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Combination</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Arrays</td>
<td>35</td>
<td>20</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>32</td>
<td>20</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

• Clover: our prototype implementation
• Evaluated on three sets of benchmarks:
  1. 101 CHC-Comp 2019 benchmarks
  2. Benchmarks containing mutual recursion, programs based on
     Montgomery encoding and s2n, and combinations of these
  3. Benchmarks containing unbounded arrays
• Compared against other tools [1,2,3,4,5]
• Results demonstrate Clover is very effective for the latter two benchmark sets while remaining competitive with other tools on the first

References