Managing Path Explosion in Lazy Symbolic Execution

Rushyendra Maganty

November 2019

Supervised By:
William Hallahan, PhD Candidate, Department of Computer Science, Yale University
Ruzica Piskac, Associate Professor of Computer Science, Yale University

Abstract

Symbolic execution is a program analysis technique that runs an interpreter which takes all possible paths through a program, continually updating mathematical expressions that constrain the values a variable can take in each branch. Prof Piskac and her group have previously implemented G2, a lazy-reduction based symbolic execution engine for Haskell. Like all such tools, it suffers from the path explosion problem, which arises when the number of branches the interpreter has to explore grows exponentially. To counteract this, we implemented a lazy variant of state merging to merge paths where possible. Among the challenges we encountered in implementing this technique was a devising a way to determine the order of reduction of different paths. We thus conceived and implemented two possible functional data structures, one a variant of a directed acyclic graph, and the other based off of a Zipper on a multiway tree. The key difference between the two is that in the former, at repeated intervals the set of states derived from a specific common branch are merged, whereas in the latter longer branches are spun off into separate trees and are no longer merged with others. We evaluated the performance of both and observed that the Tree Zipper structure performs almost uniformly better than other. In comparison to running G2 without state merging, a large divergence was observed, with the Tree Zipper performing significantly better in some cases and worse in others. Heuristics are thus needed to selectively enable state merging when it would confer a performance boost.

1 Introduction

As software programs balloon in size and complexity, it becomes increasingly difficult for developers to ensure that their code meets specifications and is free of subtle errors. Symbolic Execution tools enable developers to specify high-level properties that their programs should satisfy, and check that their code does not violate any of these properties, generating sample test inputs in the event that they do.

G2 is one such tool developed by Prof. Piskac and her lab [1]. It targets programs written in Haskell, which is a language with non-strict semantics. Like all symbolic execution tools, it suffers from the Path explosion problem, in which the number of paths the engine generates may grow exponentially due to branching, rendering symbolic analyses of certain problems intractable. To counteract this phenomenon, we have over the past half year implemented lazy-reduction based state merging, which aims to merge different paths where possible to mitigate repeated computation. In this report, we specifically describe
the functional data structures implemented to determine the order of execution and merging of different paths, and our initial evaluation of the effectiveness of the tool.

2 Background

2.1 Symbolic Execution

The central idea behind symbolic execution involves replacing test inputs to a program with symbolic values, and then running an interpreter through the code, continually updating a mathematical expression (path condition) that constrains the values these symbols can take. At each point where the executor encounters a branch condition that depends on the symbolic values, the execution splits into two separate states, each with its own path condition. If the execution reaches an assertion statement or error, the path condition is analyzed to see if there exists some combination of concrete input values that violates the assertion statement and fits the constraint. This combination of concrete values is returned as a test input which can be used by the developer to verify the error. The path conditions are usually expressed as a first-order logic formula, and solved using a Satisfiability Modulo Theories (SMT) Solver. For instance, consider the following simple program. Our goal is to figure out if any set of inputs $x,y$ would result in an ERROR.

```plaintext
1. int foo(int x, int y){
2.   int z = x - 10;
3.   if(z > y){
4.     if(2y > x){
5.         ERROR
6.     }
7.   }
8.   return z;
9. }
```

![Symbolic Execution Tree for Function foo()](image)

The symbolic executor runs through the code as shown in Fig 1. At each step, it maintains a heap $H$ that stores the mapping between symbolic values and expressions over the concrete values $x,y$ and $z$, and updates the path condition $P$. When it encounters the ERROR statement, it invokes the SMT solver.
with the path condition $P_4$, which then tries to find a set of values of $x$ and $y$ that satisfies the condition. In this case, there exists such a possible set of values: $x = 24$ and $y = 10$. Hence, the symbolic executor could return such a pair of values as a test input that causes the function to fail.

### 2.2 Path Explosion in Lazy-Reduction Based Symbolic Execution Engine

G2’s current handling of symbolic algebraic data types can cause execution to branch into multiple different branches. When this happens repeatedly, the number of paths grows exponentially and performance of the tool degrades significantly. For example, consider the simple function below which returns the length of a list.

```haskell
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

When G2 evaluates this function on a symbolic list $a$, the symbolic list is pattern matched with both data constructors. G2 thus takes both branches, splitting execution into two. When execution continues on the second branch and the expression $length \ xs$ is reduced, it further splits into another two branches. In this manner, execution can continue indefinitely as all possible values of the symbolic variable are explored.

### 2.3 Prior Status of Project

Professor Piskac and her group have previously developed an operational semantics for G2 that specifies the method by which different Haskell expressions are reduced. G2 represents the symbolic values used and constraints detected at each point in the symbolic execution of a program by a tuple of values called a symbolic state $S$. It is of the form $(E, H, P)$, where the expression $E$ corresponds to the term being evaluated, the heap $H$ is a mapping from bound variables to terms, and the path constraint $P$ is a logical formula representing the constraints on the symbolic values thus far in order for the execution to have reached that state.

In prior months, we have extended these semantics by adding a set of transition rules for G2 that specify when and how state merging occurs. We have also written functions to implement these rules in G2, in the form of a state merging operation. This operation can be abstractly conceived of as a function that takes states $S_1 = (e_1, H_1, P_1)$ and $S_2 = (e_2, H_2, P_2)$, and combines them into a new state $(e_1', H_1', P_1')$, which encodes information from both $S_1$ and $S_2$ in a recoverable manner.

### 3 Functional Data Structures for Merging-Enabled Lazy Reduction

In addition to defining a set of transition rules that determine when and how state merging occurs during lazy reduction, it is also important to specify the order in which states from different paths are picked to be reduced. Constantly picking the same path for further reduction could result in poor performance if the chosen branch does not terminate. On the flipside, constantly switching paths could result in the program taking extremely long to find terminating states on earlier, shorter paths. We thus developed two different search strategies to deal with this problem, and evaluated the performance of both.

The two strategies, which we refer to as the Work Graph and Tree Zipper, make use of two different functional data structures to control the order of evaluation of different paths. The constructors for both
data structures accept two arguments, a reduce function that takes a Reducer and a State, and applies the relevant reduction rule(s) to the State, and a merge function that takes a set of states and attempts to merge them.

reduce :: Reducer \rightarrow State \rightarrow [State]
merge :: [State] \rightarrow [State]

3.1 Work Graph Data Structure

The Work Graph structure creates an implicit directed acyclic graph, with States represented as vertices. There exists edges between each state and its successors (if any) that result from evaluating a Case split statement, as well as between states and the common state they may be merged together to form.

This graph is represented by having each State contain a stack of type \([\text{Int}]\). Let \(\text{edge}(u, v)\) denote an edge between \(u\) and \(v\), \(\text{stack}(S)\) denote the stack of State \(S\), \(\text{head}(sck)\) denote the top element of the stack \(sck\), and \(\text{tail}\) denote its remaining elements. Let \(C(s)\) be a predicate that returns true if and only if state \(s\) results from splitting on a case statement, and \(M(s)\) be a predicate that returns true if and only if \(s\) is the result of merging states together. When two states \(S_1\) and \(S_2\) result from splitting on a case statement while reducing state \(S_3\), a common element is pushed to both their stacks indicating a potential merge point, such that:

\[
\forall S_1. \forall S.C(S_1) \rightarrow (\text{edge}(S, S_1) \iff \text{tail}(\text{stack}(S_1)) = \text{stack}(S))
\]

\[
\forall S_1. \forall S_2. \forall S.C(S_2) \land C(S_1) \rightarrow (\text{edge}(S, S_1) \land \text{edge}(S, S_2) \iff \text{stack}(S_1) = \text{stack}(S_2) \land \text{tail}(\text{stack}(S_1)) = \text{stack}(S))
\]

Likewise, two states \(S_1\) and \(S_2\) are merged to form state \(S_3\) only if both their stacks are equal. When merged, the heads of both stacks are deleted.

\[
\forall S_1. \forall S_3. M(S_3) \land (S_1, S_3) \iff \text{tail}(\text{stack}(S_1)) = \text{stack}(S_3)
\]

\[
\forall S_1. \forall S_2. \forall S_3. \text{edge}(S_1, S_3) \land (S_2, S_3) \land M(S_3) \rightarrow \text{stack}(S_1) = \text{stack}(S_2)
\]

In addition to the Stack in each State, the Work Graph consists of a sequence of indices representing the order in which to evaluate states, called a WorkPlan. In each iteration, the first index \(i\) in the sequence is picked and the states with stacks \(sck\) such that \(\text{head}(sck) = i\) are chosen to be reduced. To enable efficient lookup of such states, a map from indices to sets of states is stored as a WorkMap.

| type WorkMap a = HM.HashMap Int (Sequence a, Sequence a, Sequence a) |
| type WorkPlan = Sequence Int |

To determine this sequence, indices corresponding to the heads of the stacks of each of the initial states are first added. As the first index is picked and a state is reduced to multiple states, indices corresponding to the new states are added to the front of the sequence. In this manner, execution proceeds in a depth-first fashion down a particular path. To prevent infinite iteration down any path, each State also stores the number of predecessor vertices corresponding to each individual unmerged Case split expression. When this count exceeds a threshold for any particular Case Split, reduction on the state is paused, and the index is added to the back of the sequence. In this manner, the data structure balances competing demands for breadth and depth-first execution, while also ensuring that all mergeable states at a particular index are certainly merged at repeated intervals.

The fields of the data structure are thus as follows:
**data** WorkGraph a b = WorkGraph { wPlan :: WorkPlan
    , wMap :: WorkMap a
    , reduce :: a -> b -> IO ([a], b, Status)
    , merge :: Sequence a -> Sequence a -> b
                      -> (Sequence a, Sequence a, Maybe Int, b)
    , add_idx_func :: Int -> a -> a
    , curr_idx :: Int
    , max_idx :: Int
    , context :: b }

**data** Status = WorkNeeded | WorkSaturated | Split | Mergeable | Accept | Discard

### 3.2 Multiway Tree Zipper Data Structure

The second data structure we implemented was a Zipper wrapper on multiway Tree. A Tree is defined recursively using the following constructors:

```haskell
**type** Counter = Int
**data** Tree a = CaseSplit [Tree a]
    | Leaf a Counter
    | ReadyToMerge a Counter
    | Root [a] (Tree a)
    | Empty
```

*CaseSplit [Tree a]* represents a root node corresponding to a State that has been split into multiple States *[Tree a]*. A *ReadyToMerge a Counter* tree indicates that the *a* can be merged with its siblings. The *Root[a] (Tree a)* constructor contains a *Tree a* corresponding to the root of the current execution, and a list of states *[a]* that have yet to be reduced.

To enable efficient traversal of the Tree in a functional setting, we used a Zipper [3]. A Zipper lets one focus on a specific node in the subtree, whilst also keeping a path from the current node to the root. This enables traversal from the node to the rest of the tree without necessarily having to start traversal from the root of the tree each time. The Zipper is defined as follows:

```haskell
**newtype** Cxt a = Cxt [(Tree a, [Tree a])]
**type** Zipper a = (Tree a, Cxt a)
```

Here, *Cxt a* represents a list of *Parent Tree, [Sibling Tree]* pairs corresponding to a path from the current focus of the Zipper to the root of the Tree. The fields of the data structure are thus as follows:

```haskell
**data** ZipperTree a b = ZipperTree { zipper :: Zipper a
    , env :: b
    , reduce :: a -> b -> IO ([a], b, Status)
    , merge :: [a] -> b -> ([a], b)
    , reset_merging_func :: a -> a }
```

With this data structure, the list of initial states to be reduced is added to a *Root [a] (Tree a)* node, and a Zipper is set to focus on this node. Execution proceeds by building out the tree in a depth-first manner. The State(s) in the node focused on by the Zipper are reduced each time. If a split occurs, the
new nodes are added as children and one of them is selected for the Zipper to focus on. Likewise, if states are merged, the nodes are deleted and the Zipper is set to focus on the parent.

Similar to the previous data structure, to prevent infinite iteration down any path, each node also stores the number of predecessor nodes corresponding to each individual unmerged Case split expression. When this count exceeds a threshold for any particular Case Split, the State corresponding to the node is lifted to the root of the Tree, to be evaluated once the rest of the tree is eventually deleted due to states being accepted, discarded or paused.

The key difference between the Tree Zipper and the Work Graph described earlier lies in how infinite or non-quickly terminating paths are dealt with. In the Tree Zipper, if an infinite path is detected when a threshold is exceeded, the subtree corresponding to that path is branched off into a separate tree and marked for execution later. In the Work Graph, only execution on the path is paused, though the path still remains marked as being mergeable with other sibling paths.

4 Evaluation

To evaluate the effectiveness of our tool, we integrated the state merging functionality into the constraint solving library, G2Q [2]. G2Q is built on top of the G2 symbolic execution engine, and takes a Haskell predicate, along with some concrete and symbolic variables. It either returns Nothing if no value is found that satisfies the predicate, or Just values for the symbolic variables.

We assembled a set of miscellaneous satisfiable predicates written in Haskell, and measured the time taken for G2Q to return a set of values with both state merging enabled and without. The results are shown in Figure 2 below.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Arguments</th>
<th>Runtime With Merging</th>
<th>Runtime With Merging</th>
<th>Runtime: No Merging</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- Work Graph (s)</td>
<td>- Tree Zipper (s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sumEventsTest</td>
<td>xs = 5</td>
<td>Timeout</td>
<td>16</td>
<td>390</td>
<td>[g2 (x :: \text{Int}) -&gt; ?(xs :: \text{Int})</td>
</tr>
<tr>
<td>validateLuhn</td>
<td>a = 15</td>
<td>1.1</td>
<td>1</td>
<td>15</td>
<td>[g2 (a :: \text{Int}) -&gt; ?(idn :: \text{Int})</td>
</tr>
<tr>
<td>subseqOfTest</td>
<td>a = [1,2,1,3]</td>
<td>2.1</td>
<td>0.6</td>
<td>7</td>
<td>[g2 (a :: \text{Int}) -&gt; ?(b :: \text{Int})</td>
</tr>
<tr>
<td>foldrTest2</td>
<td>z = 0</td>
<td>Timeout</td>
<td>3</td>
<td>22</td>
<td>foldrTest2 z xs ys</td>
</tr>
<tr>
<td>divTest</td>
<td>a = 5</td>
<td>4.5</td>
<td>8</td>
<td>36</td>
<td>[g2 (a :: \text{Int}) -&gt; ?(b :: \text{Int})</td>
</tr>
<tr>
<td>compressTest4</td>
<td>a = 2</td>
<td>Timeout</td>
<td>18</td>
<td>58</td>
<td>[g2 (a :: \text{Int}) -&gt; ?(xs :: \text{Int})</td>
</tr>
<tr>
<td>replGetTest</td>
<td>i = 3</td>
<td>5.4</td>
<td>11</td>
<td>29</td>
<td>[g2 (i :: \text{Int}) -&gt; ?(j :: \text{Int})</td>
</tr>
<tr>
<td>foldrTest</td>
<td>z = 0</td>
<td>Timeout</td>
<td>5.6</td>
<td>7.4</td>
<td>[g2 (z :: \text{Int}) -&gt; ?(xs :: \text{Maybe Int})</td>
</tr>
<tr>
<td>vectorAdd</td>
<td>a = [1,2,3,2,3,4,5,6,6,6,\ldots,9,9,9,9,9,9,2,3,4,5,6]</td>
<td>0.5</td>
<td>0.4</td>
<td>0.2</td>
<td>[g2 (a :: \text{Int})</td>
</tr>
<tr>
<td>compressTest3</td>
<td>a = 11</td>
<td>Timeout</td>
<td>42</td>
<td>17</td>
<td>[g2 (a :: \text{Int}) -&gt; ?(xs :: \text{Int})</td>
</tr>
<tr>
<td>greaterThan10Less</td>
<td>y = 145</td>
<td>0.3</td>
<td>0.2</td>
<td>0.1</td>
<td>[g2 (y :: \text{Int}) -&gt; ?(x :: \text{Int})</td>
</tr>
<tr>
<td>reverseTest</td>
<td>len = 15</td>
<td>Timeout</td>
<td>13</td>
<td>1</td>
<td>[g2 (len :: \text{Int}) -&gt; ?(a :: \text{Int})</td>
</tr>
<tr>
<td>sumEventsTestSlow</td>
<td>x = 3</td>
<td>Timeout</td>
<td>64</td>
<td>4.4</td>
<td>[g2 (x :: \text{Int}) -&gt; ?(xs :: \text{Int})</td>
</tr>
<tr>
<td>runLengthEncodeTest</td>
<td>a = 4</td>
<td>Timeout</td>
<td>Timeout</td>
<td>1</td>
<td>[g2 (a :: \text{Int}) -&gt; ?(xs :: \text{Int})</td>
</tr>
<tr>
<td>compressTest2</td>
<td>a = 1</td>
<td>Timeout</td>
<td>Timeout</td>
<td>0.7</td>
<td>[g2 (a :: \text{Int}) -&gt; ?(xs :: \text{Int})</td>
</tr>
<tr>
<td>compressTest</td>
<td>ys = [1,2,3]</td>
<td>Timeout</td>
<td>19.2</td>
<td>0.1</td>
<td>[g2 (ys :: \text{Int}) -&gt; ?(xs :: \text{Int})</td>
</tr>
<tr>
<td>rangeAssert</td>
<td>lo = 2</td>
<td>0.4</td>
<td>Timeout</td>
<td>0.4</td>
<td>[g2 (lo :: \text{Int}) -&gt; ?(hi :: \text{Int})</td>
</tr>
</tbody>
</table>

Figure 2: Runtimes for various predicates with and without merging enabled in seconds

It can be observed that the tree Tree Zipper performs significantly better in comparison to the Work
Graph. When compared to running G2 without state merging enabled, a large divergence is observed, with merging significantly improving runtimes in certain cases, and being slower in others. More research is needed to understand the reasons for this divergence.

5 Next Steps

In the upcoming months, we aim to focus our efforts on building tools to visualize and analyze the execution paths taken by G2 on its runs. By generating reports, we hope to identify performance bottlenecks and discover potential modifications to the data structures used that could speed up execution. We then hope to implement and evaluate various heuristics that would selectively enable state merging at specific points of execution when it predicts a speedup, and disables merging at points where it would slow execution down. We have also considered implementing various extensions, including the use of methods reminiscent of taint analysis to cut down on the number of different paths explored in certain cases.

References

