1 Introduction

The complexity of software analysis scales with program size. In the context of program testing, for instance, a simple approach is to utilize customized test scripts. Such scripts would compare known inputs with expected outputs. However, customized test scripts are inherently difficult and time-consuming to write, and lack scalability with the growing complexity of software architectures that they target.

Alternative automated techniques such as randomized testing have also emerged to address scalability. While randomized testing alleviates some engineering cost of writing customized test scripts, such techniques cannot yield formal proofs nor deterministically guarantee program coverage due to their probabilistic nature.

The difficulty of achieving sound analysis of executional behavior of large programs suggests that a different angle of attack is needed. Symbolic execution is one such approach.

2 Symbolic Execution

The core idea behind symbolic execution is to execute on symbolic input values.

In practice, this is done by substituting concrete values for symbolic ones during program execution. By making this substitution, it is possible to examine how symbolic inputs result in different end states of program execution. As branching instructions such as if-then-else statements are hit, the current execution state duplicates itself to continue execution along all possible branches simultaneously. Each state is tagged with a unique path constraint, a conjunction of logical conditions that its variables must satisfy in order to reach such state.

In contrast, concrete program execution forces input variables to take on concrete values, meaning that all branching from a deterministic program can be known immediately as a
function of the input values. This also means that customized test scripts and randomized testing can only explore one trace of execution per run, with strong risk of redundancy and path overlaps.

To observe symbolic execution in action, consider the following Haskell function `foo` that takes three parameters `a`, `b`, and `c` and branches as a function of input parameters:

```haskell
foo a b c = if a + b < c
  then a + b
  else if c < 5
    then b + c
    else a + c
```

Each end result has a unique path constraint that program variables must satisfy in order to reach it. For example, the result `b + c` requires satisfaction of `a + b < c` and `c < 5`. By keeping track of such path constraints, we can then utilize automated reasoning tools such as SMT solvers to generate satisfiable concrete value substitutions for symbolic ones that would conclude at each end state. This allows us to automatically and deterministically explore program behavior.

### 3 Previous and Current Work

Historically, symbolic execution engines were primarily geared towards imperative language such as C and Java, with less attention put towards functional languages. However, with recent growing interest in robust software analysis and engineering techniques, functional languages have come to occupy an increasingly important space due to features such as equational reasoning, object immutability, and strong static typing.

In particular, I have been working in Professor Ruzica Piskac’s group to develop G2, a symbolic execution engine that targets the Haskell programming language. Haskell is a general purpose functional language with strong static typing and lazy evaluation. The language also serves as a testbed for programming language researchers that work with emerging techniques such as dependent type systems and generalized algebraic data types.

The current engine has many core features of a symbolic execution engine implemented, such as Haskell’s lazy evaluation semantics and integration with an SMT solver. A number of crucial features are still lacking, however, which hurts the overall utility of the engine.
4 Missing features and goals

4.1 Integration with standard Haskell library and GHC quirks

Because G2 interfaces with the Glasgow Haskell Compiler in order to extract an intermediate representation from Haskell source code, it is very much at mercy of the oddities and artifacts of GHC. In particular, because distributions of GHC come with standard libraries as compiled objects rather than source code, it is not possible for G2 to directly import them.

The current proposed approach is to curate by hand a subset of Haskell’s standard library and export G2 along with such set of files. Such included portions of the library would largely deal with a few standard data structures and commonly used functions. Parts of the library that deal with internal GHC representation of numerical values, for instance, would not be included as part of the curated subset.

4.2 Optimizations

A key point of efficiency advertised in lazy evaluation is the need to evaluate shared expressions only once, and only when necessary. Because symbolic execution branches on all paths simultaneously when encountered with a conditional statement, it is possible that the same expression is evaluated redundantly in the child states.

A proposed solution is to share an execution environment across all instances of execution states. Such an approach would allow for an expression to be readily reused by other states after one state has completed its evaluation in the environment. However, engineering difficulties would include working around Haskell’s functional purity and object immutability, which can make sharing environments difficult, as it would require a mutable global copy.

Other optimizations that we are interested in trying include more frequent calls to the SMT solver. Currently, G2 attempts symbolic execution within a bounded number of steps before calling the SMT solver on all terminated states. It has been experimentally demonstrated that a large number of these states may have unsatisfiable constraints. However, unsatisfiability is disregarded during actual execution, since logical values of symbolic variables are never directly examined. As such, G2 may spend a large amount of time running useless calculations.

A possible solution to this is to mix solving with execution. Because G2 internally handles symbolic execution as repeated calls to a reduce function for each until either a normal form is reached or a step counter runs out, it is possible to discretize execution and inject solver calls to path constraints every so often. This would in theory allow for faster rejection of unsatisfiable states, which should save computation time.
4.3 Garbage collection

G2 does not yet have garbage collection features, which means that prolonged runs of the engine is memory intensive. An obvious solution is to implement garbage collection for G2. The approach would most likely be to use an existing algorithm such as Cheney’s algorithm. Such a feature will hopefully reduce G2’s memory footprint.

4.4 Integration with LiquidHaskell

LiquidHaskell is an extension of Haskell’s type system that allows for predicates to be included within Haskell’s types. For instance, we may specify a function to take non-negative integers, thereby emulating natural numbers, which are not a native construct to Haskell.

The researchers behind LiquidHaskell have expressed interest in integrating LiquidHaskell with G2 in order to utilize symbolic execution capabilities as a means of generating counter examples during type-level verification. We are currently in communication with the team at UC San Diego about how to proceed with integration work.

4.5 Type polymorphism

Haskell functions permit polymorphic types, which is problematic to SMT solvers because SMT does not allow arbitrary type polymorphism. However, because programs are finite in length, it is possible to get the type information about all the types that a polymorphic function is invoked with. This should give us a starting point to constrain the set of types that such functions may take when we translate to SMT formulae.

5 Project proposal

In this CPSC 290 project, I will address some of the features currently missing in G2. I will first address import issues with Haskell’s standard libraries. Next, I will work on integration with LiquidHaskell, with G2 as an embedded engine. Thirdly, I will work on revamping G2’s internal execution architecture in order to handle environment sharing. Lastly, if time permits, I will work on other missing features, prioritizing garbage collection.