Building a Symbolic Execution Engine for Haskell

William T. Hallahan
Computer Science
Yale University
william.hallahan@yale.edu

Anton Xue
Computer Science
Yale University
anton.xue@yale.edu

Ruzica Piskac
Computer Science
Yale University
ruzica.piskac@yale.edu

Abstract

Symbolic execution is a powerful software analysis technique that reasons about the possible execution states of a program due to logical branching.

Historically, such techniques are primarily geared towards imperative languages such as C and Java, with less effort in the development of frameworks for functional languages. While such works do exist, many are focused on contract-based analysis, and some lack full support for reasoning about functional expressions.

In this paper we outline the application of symbolic execution techniques to Haskell. Our methods for program model extraction, defunctionalization, execution semantics, and constraint solving have strong implications for static analysis based testing, and for future work in program verification and synthesis.

CCS Concepts • **Software and its engineering → Functional languages**;

Keywords Symbolic Execution, Functional Programming, Higher-Order Functions, Haskell

ACM Reference Format:
William T. Hallahan, Anton Xue, and Ruzica Piskac. 2017. Building a Symbolic Execution Engine for Haskell. In *Proceedings of TAPAS 17, New York, NY, USA, August, 29, 2017,* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The difficulty of software analysis scales with program size: large programs with complex interactions are difficult for humans to reason about.

In the context of testing, a simple approach to checking program behavior is through the use of automated test scripts. However, such scripts do not scale well with development, as additional tests must be written to contain the growing complexity of program logic. Furthermore, writing effective tests is inherently difficult and time consuming to do, placing unnecessary burdens on developers. Techniques such as randomized testing [6] can partially address these issues, but cannot yield formal proofs nor guaranteed coverage due to their probabilistic nature.

The difficulty of achieving sound coverage by simply querying a program with input suggests that a different of angle attack is needed. Symbolic execution is one such approach.

The core idea behind symbolic execution is to use symbolic variables in place of concrete values for program input data. By making this substitution, it is possible to examine how arbitrary input results in different end states of program execution. As branching instructions such as *if-then-else* or *case* statements are hit, the current execution state duplicates itself to continue execution on all possible branches simultaneously. Each state is tagged with a *path constraint*, a conjunction of the logical conditions that its variables are required to satisfy in order to reach the current point of execution. By contrast, during concrete program execution, concrete values for variables forces logical branching to take only one of the several potential paths available. In testing scenarios, this reduces the amount of code coverage that each test case is able to hit.

For instance, consider the following Haskell function `foo`, which has three parameters named `a`, `b`, and `c`:

```
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 required to reach it. For instance, in order to reach `b + c`, we must satisfy `a + b < c` and `c < 5`. By keeping track of these path constraints, we can utilize automated reasoning tools such as SMT solvers [3] to generate satisfiable concrete substitutions for variables that would conclude at each end state.

Additional contract-based assertions and assumptions about program state can also be encoded within path constraints. For example, a general assertion check can be expressed as follows: if there exists a state that satisfies the negation of the assertion, it implies that the assertion has failed.

with paper note.
TAPAS 17, August, 29, 2017, New York, NY, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Symbolic execution is not without its flaws, however. One difficulty is looping or recursion conditions that depend on symbolic values themselves, which can easily result in infinite branching. Furthermore, because multiple states appear per logical branch, *path explosion* can quickly drain memory resources. Other challenges occur at the constraint solving phase: as SMT solvers are currently only equipped to deal with first-order logic, it is difficult for symbolic engines to reason about higher-order functions. Such limitations affect in existing symbolic engines [4], and limits the types of functional programs that can be effectively analyzed.

In this paper we outline our progress towards unassisted symbolic execution of Haskell. We document techniques used for program model extraction from Haskell source, execution semantics under symbolic evaluation, and constraint solving strategies taken.

Furthermore, we illustrate how general symbolic execution problems such as path explosion, execution semantics, and exploration heuristics are tackled in G2. We also demonstrate the feasibility of defunctionalization techniques applied to domain-specific problems in the constraint solving of higher-order functions. The techniques applied in our work extends the power of symbolic execution engines to reason about a larger family of programs than before, and provides a solid base for future development in this area.

2 Design

The core design of a symbolic execution engine for higher-order functional languages consists of program model extraction, defunctionalization, execution semantics, and constraint solving.

2.1 Model Extraction

We now describe the representational model used in the G2 symbolic execution engine that we call G2 Language.

This language is extracted by leveraging the Glasgow Haskell Compiler (GHC) API to perform partial compilation of Haskell programs into a lambda calculus intermediary called Core Haskell. While Core Haskell is a relatively concise representation of Haskell source programs, it nevertheless contains excessive amounts of irrelevant compilation information. Thus, we further translate Core Haskell into G2 Language, which is in approximate one-to-one correspondence with Core Haskell's high-level features. This language is complex enough to express Core Haskell, and thus Haskell's syntactic constructs, yet concise enough to contain only the information relevant to us.

$$
\begin{array}{lcl}
\langle expression \rangle & ::= & \langle variable \rangle \\
& | & \langle constant \rangle \\
& | & \lambda x \;.\; \langle expression \rangle \\
& | & \langle expression \rangle \; \langle expression \rangle \\
& | & \langle datacon \rangle \\
& | & \texttt{case} \; \langle expression \rangle \; \texttt{of} \; \overrightarrow{\langle alt \rangle} \\
& | & \texttt{BAD} \\
\langle variable \rangle & ::= & \langle name \rangle \; \langle type \rangle \\
\langle constant \rangle & ::= & int \mid float \mid char \mid \ldots \\
\langle datacon \rangle & ::= & \langle name \rangle \; \overrightarrow{\langle type \rangle} \\
\langle alt \rangle & ::= & \langle datacon \rangle \; \overrightarrow{\langle variable \rangle} \; \langle expression \rangle \\
\langle type \rangle & ::= & TyInt \mid TyFun \; \langle type \rangle \; \langle type \rangle \mid \ldots
\end{array}
$$

Within an *expression*, a *variable* is paired with an identifying *name* and corresponding *type* that is used during constraint solving. *constants* are values that cannot be reduced further. Because all functions are curried in Haskell, single-parameter lambda functions are sufficient to represent all functions. Next, lambda expression application is treated as the application of the right *expression* to the result of the left *expression*. Data constructors are similar to variables: they are likewise endowed with a unique *name*, and have a list of *type* to denote its parameters. Branching is done by data constructor matching on *alt* cases, which have a parameter in the form of a *variable* list that match to the arguments taken by the *datacon*. All logical decisions can be broken down into this format, even `True` and `False` from *if-then-else* statements. Lastly, `BAD` represents a catch-all error state.

An execution state is defined as $(\mathcal{E}, \mathcal{C}, \mathcal{P})$. $\mathcal{E}$ is an environment that maps *name* to *expression*. $\mathcal{C}$ is the current *expression* under evaluation. $\mathcal{P}$ is the path constraint accumulated so far for the execution state.

2.2 Defunctionalization

Because Haskell is able to natively express higher-order functions that are outside of the reasoning capabilities of the first-order SMT solvers, we apply *defunctionalization* techniques introduced by Reynolds [10] in order to lower higher-order terms to first-order ones.

Let $t_1, \ldots, t_n$ be a list of the function types used as arguments in $\mathcal{E}$. For each t_i, we introduce a new datatype, $apply\_type_i$ and a new function $apply\_func_i$ of type $apply\_type_i \to t_i$. We refer to these as *apply types* and *apply functions*. For each function f of type t_i, we introduce a constructor f_{cons} for $apply\_type_i$. The role of $apply\_func_i$ is to perform pattern matching on the constructors of $apply\_type_i$. When a match on f_{cons} is found, the appropriate function f is invoked with the corresponding arguments that were also passed into $apply\_func_i$.

Every higher-order function is then adjusted in a preprocessing step to accept apply types in place of function arguments. Each call to a higher order function is replaced by a call to an apply function, which is passed an apply type. In this way, we preserve the semantics of the original program, but reduce reasoning about higher-order functions to reasoning about first-order functions.

This transformation enables SMT solvers to reason about higher-order functions by proxy.

2.3 Execution Semantics

The goal of G2 execution semantics is to reduce a state down to a terminal one, in which $\mathcal{C}$ is a normal form achieved through rewrites and $\mathcal{E}$ lookups.

Each expression in G2 Language has a set of corresponding rewrite rules akin to those of similar lambda calculus based languages. These rewrite rules can be applied to step through the execution sequence of a particular state in discrete increments. Because steps are applied incrementally, this allows us to bound the depth of our execution space, and prevent automatic infinite looping that would otherwise cause the engine to run indefinitely. Several execution heuristics can be applied to explore the space, such as depth-first-search or breadth-first-search techniques. We currently use breadth-first-search techniques to apply stepping on all states in a queue until a counter limit is hit. This has the advantage in achieving balanced coverage in execution states, and avoids scenarios that would result in diving too deep into infinite recursion conditions.

Execution branching occurs during the evaluation of *case* expressions when the inspected expression is a symbolic value. In this scenario, the state is duplicated and all *alt* branches are taken simultaneously, with the constraints $\mathcal{P}$ for each state appropriately updated. Symbolic values can be represented as variables who lack a corresponding lookup in the environment $\mathcal{E}$, although other explicit annotations are possible. This technique can be applied to delay evaluation of the *Haskell Prelude*-dependent portions of the program that we cannot easily extract G2 representations for, such as arithmetics, until SMT constraint solving, where direct translation can be done on variable name inspections.

Furthermore, because Haskell has lazy evaluation, similar semantics must be preserved in G2. In particular, this involves favoring the left expression during for evaluation during expression applications.

2.4 Constraint Solving

Each path constraint gives a representational formula that can be fed into the Z3 SMT solver.

The translation is straightforwrad: Haskell datatypes become Z3 *sort*, while arithmetical operations on numerical constants are mapped directly to their equivalents Z3. We currently do not support translation of non-normal form expressions to SMT equivalents, as this significantly increases encoding complexity while yielding little value. Furthermore, due to the breadth-first-search nature of our execution, we tend to find terminal states quickly anyways. Additional exploration of the execution space can be achieved by increasing counter limits on the engine.

3 Related Work

There has been considerable work in the analysis of Haskell programs. Our project is still a work in progress, with a flexible future for testing and implementation of different ideas. As such, we have focused on building a solid, general base for the symbolic execution of Haskell.

Projects such as static Contract Checking [14] and HALO [13], are mostly concerned with verifying pre and post-conditions annotated within the program. HALO, in particular, does not rely on symbolic execution: it uses a direct translation between Haskell functions and first-order logic formulas. While Contract Checking does make use of symbolic execution, it has limitation in its ability to support recursive predicates in pre and post conditions of function calls that we are working to address.

Catch [7] is specifically designed for the identification of pattern matching errors, while Reach [8] is focused on determining reachability in Haskell code. These are actually quite similar problems: a pattern matching error is really just an instance of reaching the end of a *case* expression. The techniques utilized, however, are different. Catch does not rely on symbolic execution, instead it generates constraints that allow it to create proofs that pattern matching statements do not result in errors. Reach does utilize symbolic execution, and appears to have evaluation semantics similar to ours.

For this reason, Reach is probably the most similar existing work to our project. While we are not yet at a stage where a side by side comparison is possible, we hope to be able to both cover a larger subset of Haskell, and be more efficient, than Reach. In particular, Reach does not support Haskell's full standard library, which we are currently investigating handling. Furthermore, Reach also has a more narrow objective in pure reachability testing of Haskell programs, while we also aim our engine to target other challenges in domains such as verification and synthesis.

4 Conclusion and Future Work

G2 integrates and improves upon existing symbolic execution techniques for Haskell that allow us to perform analysis on a larger family of programs.

Similar techniques can be applied to symbolic engines that will be developed for similar functional languages for more effective program coverage. Currently, a number of promising optimizations and configurations are in development.

For instance, although G2 favors left expression during expression applications to emulate lazy evaluation semantics, the current implementation does not support the optimization of single evaluation of shared expressions. Such implementations require a Spineless Tagless Graph Reduction (STG) [9] based approach in order to perform direct stack and heap manipulations, deviating from well-known standard lambda calculus family of evaluation semantics currently implemented. STG execution semantics in theory will also solve a few aggressive memory consumption issues with G2 that result from the need to perform nested symbolic execution during expression application. We have developed a promising STG semantics-based prototype with goals for integration.

Another crucial problem in every symbolic execution engine is speed, as annotated runs are inherently slow and multiple states must be kept track of. As such, the heuristics for exploration is important. While this has not been the immediate focus of our work, we also plan to investigate how different exploration techniques such as bounded depth-first-search work in terms of efficiency trade offs.

5 Acknowledgements

This research was sponsored by the NSF under grant CCF-1553168.

References

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[2] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[3] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[4] Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. Concolic testing for functional languages. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 137–148, New York, NY, USA, 2015. ACM.

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[6] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[7] Neil Mitchell and Colin Runciman. Not all patterns, but enough: An automatic verifier for partial but sufficient pattern matching. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA, 2008. ACM.

[8] M. Naylor and C. Runciman. Finding inputs that reach a target expression. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 133–142, Sept 2007.

[9] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201, 1989.

[10] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[11] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[12] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, April 2003.

[13] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. Halo: Haskell to logic through denotational semantics. *SIGPLAN Not.*, 48(1):431–442, January 2013.

[14] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. *SIGPLAN Not.*, 44(1):41–52, January 2009.