1 Abstract

Symbolic execution is a software analysis technique that seeks to find the possible states that a program may result in provided arbitrary inputs.

In the past, such techniques have focused on imperative language such as C and Java, with less attention given to the development of frameworks for functional programming languages. Among those that do exist, their power tends to be restricted by the limitations of SMT solvers for reasoning about functional expressions. Furthermore, many are focused on contract-based verification and related techniques that require additional program annotations, rather than unassisted methods that can run automated analysis.

In this paper we outline our efforts and progress for the development of symbolic execution techniques to Haskell. Our methods for program model extraction, higher-order functional reasoning, execution semantics, optimizations, and constraint solving have strong implications for static analysis based testing. This work also lays the foundations for future efforts in areas such as program verification and software synthesis.

2 Introduction

A major question in software analysis asks whether a program may enter some error state. Such a state may be linked to assertion violation, resource exhaustion, or whatever is considered to be undesired behavior. In broader terms, we ask what states an execution may result in. This is known as the reachability problem.

A domain in which the reachability problem moves from theory into practice is program testing. As the size of programs grow, the large web of complex interactions between multiple parts of the program becomes difficult for humans to reason about through inspection alone. A simple way to fulfill this need for computer-assisted automation is through the use of automated test scripts that check how the program’s output behaves on a set of fixed inputs with known results. However, such methods do not scale well with development, as additional tests must be written to contain the growing complexity of program logic. Furthermore, writing exhaustive test scripts is difficult and time consuming to do so, placing unnecessary burdens on developers. Other techniques such as randomized testing alleviates the burden of writing manual test cases. In general, such methods generate random samples over the program’s domain of input, and analyze against the output of the execution. However, given that randomized testing is probabilistic in nature, it cannot yield formal proofs nor guaranteed coverage for program correctness.

A key insight into the limitations of manual test scripts and randomized testing is that each execution of the program checks for a single path that a program may take. As a program may have a large number of states, blackbox testing methods that depend on querying programs are limited in their ability to leverage the internal structures of a program, resulting often resulting in a compromise of code coverage. The difficulty of achieving sound coverage through querying a program with some inputs alone suggests that a different angle of attack is necessary. Symbolic execution is one such approach.

The core idea behind symbolic execution is to perform program execution in a controlled environment by applying automated annotations and transformations on the program during compilation. In particular, program inputs are treated as symbolic values in place of concrete ones. Rather than taking on concrete values from, for instance, the integers, a symbolic value may instead be represented as a first-order logic formula over the integers that expresses its possible values of input. By making this substitution and augmenting program execution rules as appropriate, it is possible to examine how arbitrary inputs result in different end states of execution. On branching instructions such as if-then-else or case statements, the current execution state duplicates itself and continues execution on all possible branches. Each state is tagged with a path constraint, a conjunction of the logical conditions that its variables must satisfy in order to reach the current point of execution. As the symbolic input values may be represented as first-order formulae that constrain their range, this provides a natural encoding into the path constraints, and allows for recovery of variable assignments that would the reachable state described by its path constraint.

For instance, we consider the Haskell function \texttt{foo}, which has three parameters named \texttt{a}, \texttt{b}, and \texttt{c} over the \texttt{Int} type, and returns an \texttt{Int}.
foo :: Int -> Int -> Int -> Int
foo a b c =
    if a + b < c
        then a + b
        else if c < 5
            then b + c
            else a + c

Each result of executing this function has a unique path constraint that describes the constraint on the inputs required to reach it. In particular, in order to reach \( b + c \), we must satisfy the conditions that \( a + b < c \) and \( c < 5 \). By keeping track of these constraints, we can utilize automated reasoning tools such as SMT solvers to generate satisfiable concrete substitutions that would conclude at each end state.

Additional contract-based assertions and assumptions about program state and many other forms of verification can be encoded as the state reachability problem that symbolic execution tackles. As such, symbolic execution is a more general framework than that of many other techniques.

Symbolic execution is not without its flaws, however. One difficulty is that looping or recursion may result in infinite branching or cycles, as symbolic execution will execute along paths for which a state’s path constraint is satisfiable indefinitely or until some limit counter is hit. In addition, as multiple states may appear per logical branching, one infamous problem known as the path explosion problem may occur when a large number of reachable states are queued for further execution. Other challenges occur at the solving phase: as SMT solvers are designed to target first-order logic, the ability to reason about higher-order functions is limited, and is where many analysis tools for functional programming languages fall short.

In this paper we outline our progress towards unassisted symbolic execution of Haskell programs. We document techniques used for program model extraction from Haskell source, execution semantics under symbolic evaluation, and constraint solving strategies employed for optimizations. In addition, we illustrate how general symbolic execution challenges such as path explosion, execution semantics, and exploration heuristics, are tackled in our implementation for Haskell, G2. Furthermore, we demonstrate the feasibility of modified defunctionalization techniques applied to domain-specific problems in the constraint solving of higher-order functions. The techniques applied in our work extend the power of symbolic execution engines to reason about a larger family of programs, and provides a solid base for future development in the area.

3 G2

We first define the internal language that G2 uses to represent Haskell programs. We then document how to extract such a representation from Haskell source. Next, we detail program transformations in a preprocessing step that enables eventual reasoning about higher-order functional expressions. Afterwards, we present evaluation semantics for G2, which augment Haskell’s semantics in order to handle symbolic expressions. Finally, we discuss implementations for calling solvers, and miscellaneous optimizations.
3.1 DEFINITIONS

Variables $x, y, f, \ldots$

Literal $\ell :=$ int | double | char | bool

Atom $a := x | \ell$

Data constructor $D := k \tau_1 \ldots \tau_m$

Type constructor $T := t \tau_1 \ldots \tau_n D_1 \ldots D_n$

Expression $e := a$

$\oplus e_1 e_2$

$\lambda x \rightarrow e$

let $\{ x_1 = e_1 \ldots x_n = e_n \} \text{ in } e$

case $e$ as $x$ of $\{ alt_1 \ldots alt_n \}$

$D$

cast $e : \tau_1 \sim \tau_2$

type $\tau$

assume $e_1 e_2$

assert $e_1 e_2$

Alternative $alt := D x_1 \ldots x_n \rightarrow e$

$\ell \rightarrow e$

default $\rightarrow e$

Types $\tau := a$

Int, Float, Double, Char

$\tau_1 \rightarrow \tau_2$

$T \tau_1 \ldots \tau_n$

$\forall a. \tau$

TYPE

$\bot$

Data constructor $k$ that takes types $\tau_1 \ldots \tau_n$

Parametrized type constructor $t$ with data constructors

Variable

Primitive operation

Function application

Lambda abstraction

Let binding

Case matching

Data constructor

Type cast $\tau_1$ to $\tau_2$

Type as an expression

Assumption of $e_1$ about $e_2$

Assertion of $e_1$ about $e_2$

Data constructor matched alternative

Literal matched alternative

Default alternative

Type variable

Literal types

Function type of $\tau_1$ to $\tau_2$

Application of parametrized type constructor

For-all type

Polymorphic type

Bottom type

Figure 1: G2 Syntax.

3.2 Program Model Extraction

G2 utilizes the Glasgow Haskell Compiler (GHC) [2] as a front-end for extracting internal representations of Haskell source. Given its design, it is possible to use GHC as an API from inside a Haskell program to invoke steps in the GHC pipeline that perform tasks such as parsing, type-checking, and code generation on Haskell source. GHC utilizes an internal language called Core Haskell during compilation that is similar to that of G2’s. This is a language that all of Haskell is compiled through, meaning that every construct in Haskell can be represented in terms of this core language.

However, Core Haskell is still far too complicated for our needs, and carries excessive artifacts required for regular Haskell compilation. Furthermore, utilizing Core Haskell risks over-dependence on a particular version of GHC should drastic changes be made. Thus, we opted for our own version of internal representation that we translate Core Haskell from. Doing so means better control over internal program representation, and also means that version upgrades of GHC would just consist of changing the translation interface between Core Haskell and G2’s language.

We further augment our internal language with assume and assert to allow for logical assumptions and assertions that may be used to restrict and direct paths during symbolic execution.

GHC Standard Library

In order to make G2 useful, it is important that we be able to accommodate a large number of standard library functions. GHC’s standard library is known as Base, which contains a large number of modules that implement data structures or provide other programming abstractions. Many of these modules can be imported for use by just importing one file that is known as Prelude, which is itself just a file that imports other modules in Base. During GHC program compilation, Prelude is often considered to be imported by default, although it is possible to disable default behavior.
In order to account for the standard library, we downloaded a copy of Base version 4.9.1.0 from Hackage. The goal was then to compile Base into an internal G2 representation so that it would be possible for G2 to reason about the library functions that many programs would depend on.

Compiling Base was achieved first by commenting out all of the relevant Haskell files in the library. Next, we uncommented the Prelude.hs file and uncommented certain functions in its export list that we intended to support. Most of these were functions related to list operations, data constructor definitions, as well as typeclasses. For the time being, we opted to not handle IO and concurrency functionalities. We then repeatedly ran G2’s translation module against this customized version of Base. At each iteration, we uncomment lines in the library as necessary in order to suppress errors due to the inability to find certain variables. This was done until Base compiled without error.

Because several functions in Base functions either depend on unboxed primitive Haskell operations and data types or native code implementation that is done outside of Haskell, it is not possible to bring these versions into an internal G2 representation. These limitations were stubbed out and allowed to pass the typechecker by utilizing a trick involving recursive let bindings. Assume that `foo` is some function that we are unable to call or whose Haskell implementation is unable to compile through G2’s GHC interface, we may make the substitution as follows:

```haskell
define foo :: a -> b -> c
foo = let x = x  -- Recursive version that typechecks and compiles.
-- foo = nativeCalls -- Original implementation that we cannot compile.
```

The methods that were stubbed out in Base due to incompatibility with G2’s GHC interface can then be replaced injected into the execution state with custom versions implemented in G2’s language.

Another difficulty with incorporating Base into G2 is that variables are assigned unique integer tags during compilation. In particular, G2, views variables as a pair of a name and a type. By contrast, GHC also factors in variable namespace, but the occurrence name, module name, and unique tag together suffices to distinguish two variables in a single compilation instance for our purposes. The name consists of three components: the variable’s occurrence name, the variable’s module, and the variable’s unique tag. For instance, consider the following example involving `fromList` as imported from `Data.Map`:

```haskell
class I a

main = do
  ...
  let a = M.fromList ["one", 1], ("two", 2)]
  ...
```

The variable `fromList` would have an occurrence name of "fromList", a module name akin to "Data.Map", and a unique tag non-deterministically generated by GHC during compilation. Hence, it is possible to have two names that share the same occurrence name and module name, but differ in their unique tag. This appears to be common among alternative branches found in case branching.

Thus, it is possible that the bindings we compile from our customized Base may differ from those that exist in the target program we seek to analyze by a factor of unique tags. In the cases where the same variable differs in our target program and Base by a unique tag, it is a matter of searching out both by their occurrence name and module name, before replacing one unique tag with another.
3.3 Execution Semantics

\[ S := (E, C, \mathcal{PC}, \text{Stk}, b) \]

\[ \text{DataCon Constraint} \]

\[ \ell = e \]

\[ \text{Literal Constraint} \]

\[ \text{isCons } D \ x \]

\[ \text{Constructor Constraint} \]

\[ e : \text{Bool} \]

\[ \neg \text{pred} \]

Current Expression

| Evaluate e |
| Return e |

Path Constraints

\[ \mathcal{PC} := \bigwedge_i \text{pred}_i \]

Frames

| CaseFrame \( x \) of \( \{alt_1 \ldots alt_n\} \) |
| ApplyFrame \( e \) |
| UpdateFrame \( x \) |
| CastFrame \( \tau_1 \tau_2 \) |
| AssumeFrame \( e \) |
| AssertFrame \( e \) |
| CheckAssertFrame |

Figure 2: G2 Execution State

G2 performs symbolic execution by repeated evaluation of a state, \( S := (E, C, \mathcal{PC}, \text{Stk}, b) \), which consists of an Expression Environment \( E \), current expression \( C \), path constraints \( \mathcal{PC} \), a stack of frames \( \text{Stk} \), and a boolean assertion violated flag, \( b \). This evaluation is performed according to the rules in Figures 3, 4, and 5, and are adapted from Haskell’s execution semantics based on graph reduction [3], which we augmented to handle symbolic values.

The expression environment \( E \) is a mapping of variables to expressions. We define \( \text{lookup}(E, x) \) to be the expression \( x \) maps to in \( E \). We define \( E\{x = e\} \) to be the expression environment \( E \), but with \( x \) mapped to \( e \). That is,

\[
\text{lookup}(E\{x = e\}, x') = \begin{cases} 
  e & \text{if } x = x' \\
  \text{lookup}(E, x') & \text{otherwise} 
\end{cases}
\] (1)

The current expression \( C \) contains the expression \( e \) that is being evaluated in the state, alongside a keyword, Evaluate or Return. This keyword helps guide the evaluation rules. We are in an Evaluate state when \( e \) is not yet a value. We are in a Return state when \( e \) is a value, and we are either completely done with evaluation, or we must pop a frame from \( \text{Stk} \).

The path constraints \( \mathcal{PC} \) are a collection of constraints that the input values must satisfy to have reached \( C \). \( \mathcal{PC} \) is a conjunction of clauses, which come from pattern matching in case expressions, or from the left hand side of assume or assert expressions.

The stack \( \text{Stk} \) is used to guide the evaluation of \( C \). During evaluation, it is occasionally necessary to temporarily replace the \( C \), typically to prioritize the evaluation of some subsection of \( C \). At these times, the \( C \) is pushed onto the stack, and can later be popped off.

Finally, the assertion violated flag, \( b \) is used when running with assertions. It is initialized to false, and set to true when an assertion is violated.

\[
\text{isValue}(E, e) = \begin{cases} 
  e = x, & x \notin E \\
  e = \oplus e_1 \ldots e_n, & \land_{i=1}^n \text{isValue}(E, e_i) \\
  e = \ell, & \text{True} \\
  e = D e_1 \ldots e_n, & \text{True} \\
  e = \lambda x, & \text{True} \\
  \text{otherwise,} & \text{False} 
\end{cases}
\] (2)

\( \text{uniqArgs}(x, n) \) is a variable generation function that returns the same list of \( n \) unique variable every time it is given the same \( x \) and \( n \). Formally, it satisfies:

\[ \forall n_1, n_2, x_1, x_2. \exists y. y \in \text{uniqArgs}(x_2, n_1) \land y \in \text{uniqArgs}(x_2, n_2) \iff x_1 = x_2 \]
Figure 3: Non-case Evaluate Rules.
E-CASE-LIT
\[\text{alts} = \ell_1 \rightarrow e_1 \ldots \ell_n \rightarrow e_n, \ldots \quad \exists i. \ell = \ell_i \quad E' = E\{x' = \ell\}\]

(\(E\), Evaluate case \(\ell\) as \(x\) of \(\text{alts}, F, \text{Stk}, b\)) \(\vdash\) \((E', \text{Evaluate } e_i[x'/x], \text{PC}, \text{Stk}, b)\)

for some fresh \(x'\)

E-CASE-LIT-DEF
\[\text{alts} = \ell_1 \rightarrow e_1 \ldots \ell_n \rightarrow e_n, \text{default} \rightarrow e_d \quad \exists i. \ell = \ell_i \quad E' = E\{x' = \ell\}\]

(\(E\), Evaluate case \(\ell\) as \(x\) of \(\text{alts}, F, \text{Stk}, b\)) \(\vdash\) \((E', \text{Evaluate } e_d[x'/x], \text{PC}, \text{Stk}, b)\)

for some fresh \(x'\)

E-CASE-DATA
\[\exists i. D = D_i \quad E' = E\{x' = D_0 \ldots e_n, x_1^{i_1} = e_1, \ldots, x_n^{i_n} = e_n\}\]

\[\text{alts} = D_1 x_1^{i_1} \ldots x_k^{i_1} \rightarrow e_1 \ldots D_m x_m^{i_1} \ldots x_m^{i_m} \rightarrow e_n, \ldots\]

(\(E\), Evaluate case \(D_0 \ldots e_n\) as \(x\) of \(\text{alts}, F, \text{Stk}, b\)) \(\vdash\) \((E', \text{Evaluate } e_i[x'/x], \text{PC}, \text{Stk}, b)\)

for some fresh \(x'\)

E-CASE-DATA-DEF
\[\exists i. D = D_i \quad E' = E\{x' = D_0 \ldots e_n\}\]

\[\text{alts} = D_1 x_1^{i_1} \ldots x_k^{i_1} \rightarrow e_1 \ldots D_m x_m^{i_1} \ldots x_m^{i_m} \rightarrow e_n, \text{default} \rightarrow e_d\]

(\(E\), Evaluate case \(D_0 \ldots e_n\) as \(x\) of \(\text{alts}, F, \text{Stk}, b\)) \(\vdash\) \((E', \text{Evaluate } e_d[x'/x], \text{PC}, \text{Stk}, b)\)

for some fresh \(x'\)

E-CASE-NON-VAL
\[-\text{isValue}(E, e)\]

(\(E\), Evaluate \(e\) as \(x\) of \(\text{alts}, F, \text{Stk}, b\)) \(\vdash\) \((E', \text{Evaluate } e, \text{PC}, \text{CaseFrame } x \text{ of } \text{alts : Stk}, b)\)

E-CASE-LIT-SYM
\[\text{alts} = \ell_1 \rightarrow e_1 \ldots \ell_n \rightarrow e_n, \text{default} \rightarrow e_d \quad E' = E\{x'_2 = x_1\}\]

\[\bigcup_{1 \leq i \leq n} \{\text{Evaluate } e_i[x'_2/x_2], \text{PC} \land x_1 = \ell_i, \text{Stk}, b\}\bigcup \{\text{Evaluate } e_d[x'_2/x_2], \text{PC} \land \bigwedge_{1 \leq i \leq n} x_1 \neq \ell_i, \text{Stk}, b\}\]

for some fresh \(x'_2\)

E-CASE-DATA-SYM
\[E_d = \{x'_2 = x_1\}\]

\[\forall i. 1 \leq i \leq n \implies \exists i. E_i = E\{x'_2 = x_1, x'_1 = x_1^{i_1} \ldots x_k^{i_k} = x_k^{i_k}\}\]

\[\text{alts} = D_1 x_1^{i_1} \ldots x_k^{i_1} \rightarrow e_1 \ldots D_m x_m^{i_1} \ldots x_m^{i_m} \rightarrow e_n, \text{default} \rightarrow e_d\]

\[\bigcup_{1 \leq i \leq n} \{\text{Evaluate } e_i[x'_2/x_2, x'_1/x_1^{i_1} \ldots x_k^{i_k} / x_k^{i_k}], \text{PC} \land D_1 x_1^{i_1} \ldots x_k^{i_k} = x_1, \text{Stk}\}\]

\[\bigcup \{\text{Evaluate } e_d[x'_2/x_2], \text{PC} \land \neg \text{isCons } D_i, x_1, \text{Stk}, b\}\]

for some fresh \(x'_2\), and \(\forall i. 1 \leq i \leq n. \text{uniqArgs}(x_1, k_i) = x_1^{i_1} \ldots x_1^{i_k}\)

Figure 4: Case Evaluate Rules.
The Evaluate rules are for handling a \( C \) that is still in a reducible form.

- **E \(-\) VAL**
  Change the \( C \) from a Evaluate to Return upon the \( C \) reaching a value form.

- **E \(-\) VAR \(-\) VAL**
  If a variable, and the corresponding lookup\((i)\) in the \( E \) is also a value, we may simply go to a Return state.

- **E \(-\) VAR \(-\) NON \(-\) VAL**
  If the \( C \) is a variable and the corresponding lookup\((i)\) in the \( E \) is not a value, we replace the \( C \) with its corresponding lookup\((i)\) and additionally push a UpdateFrame onto the \( Stk \). The purpose of this UpdateFrame is to rewrite the \( E \) binding with whatever the value form if the lookup\((i)\) is after evaluation.
  This is done to save potential computation later, and is one of the optimizations done in lazy evaluation.

- **E \(-\) PRIM**
  During primitive evaluation we want to make sure that the arguments to the primitive operation are evaluated first. This requires us to bypass the lazy evaluation semantics by rewriting certain function definitions.
• **E − APP**
  We push the RHS of the function application onto the stack to be evaluated later, as in lazy evaluation we want to reduce the LHS first.

• **E − LAM**
  Lambda abstractions cannot be evaluated further, and are thus returned.

• **E − LET**
  We want to bind all of the declarations into the $\mathcal{E}$ before proceeding with evaluating the body of the let-expression.

• **E − ASSUME**
  The LHS makes an assumption about the logical conditions of the RHS. We save the RHS onto the $\text{Stk}$ and evaluate the LHS first.

• **E − ASSERT**
  Similar to **E − ASSUME**, we save the RHS onto the $\text{Stk}$ and evaluate the LHS.

• **E − CAST − SPLIT**
  Because we have difficulty reasoning about function-type casting, we perform a lambda abstraction and cast the new lambda parameter and body as appropriate.

• **E − CAST**
  For casting a single type to another, we push the desired cast onto the $\text{Stk}$ first. This is because the expression wrapped within the cast may not have been evaluated to a value form yet, so it is hard to deduce which variables to retype. Thus, we delay this change for later when we have reached a value form.

The case statement splits are used for determining how symbolic execution should branch in a path of execution. In particular, we branch on a particular alternative if we are able to meet the conditions of the alternative. Most notably, this happens when a data constructor or a literal expression matches. Note that because GHC desugars Haskell, complicated structures such as nested pattern matching are transformed into simpler cases.

Furthermore, Core Haskell, and thus also G2’s language, feature an additional variable in each case statement in addition to the inspected expression and alternatives. This additional variable serves to refer to the inspected expression, and is used in cases in Haskell programs where pattern matching with underscores occur.

• **E − CASE − LIT**
  The literal expressions match one of the conditions of the alternative, so we continue down that branch. We take the first non-default alternative branch that matches.

• **E − CASE − LIT − DEF**
  We failed to match any of the literal expressions, meaning that we must take the default branch. During translation, GHC should ensure that a default branch is generated, or if not, have appropriate transformations such that all possible cases of a data constructor are covered. In the case that this does not occur, GHC should be able to automatically detect pattern matching incompleteness and insert a default branch that throws an error.

• **E − CASE − DATA**
  We matched on a data constructor, bind the appropriate parameters that the matching data constructor may carry, and take this branch. For instance, in a Haskell program where something of this may occur:

```haskell
data Color = Rgb Int Int Int | Info String

case expr of
  Rgb r q b -> putStrLn ("red\_hue:\_" ++ show r)
  Info desc -> putStrLn ("description:\_" ++ desc)
```

Would be approximately represented as something of form:
Here the r, g, b, and desc are parameters of the possible alternatives that may be referred to in the body. Whether we take a branch or not will solely depend on if the data constructors match.

- **E − CASE − DATA − DEF**
  If we fail to match with any data constructor the default branch is taken.

- **E − CASE − NON − VAL**
  In the event where the inspected expression is not yet in value form, we must first reduce it to a value form before we may attempt any matching. This feature may actually be abused to force eager evaluation of an expression by applying a transformation. For instance, if we wish to have eager evaluation on an argument such as lazy2:

  ```
  add num1 lazy2
  ```

  We rewrite this within G2’s language as follows:

  ```
  Case (lazy2) (casevar)
  [(DEFAULT, [], add num1 casevar]
  ```

  Because case evaluation semantics requires the inspected expression to be evaluated first, this allows us to force order of evaluation.

- **E − CASE − LIT − SYM**
  When our expression is a symbolic value and required to match on a literal, we are required to split along all alternatives possible. This means that we create a copy of the state for each alternative, where each of the copies has exactly one of the alternative’s conditions satisfied in the \( \mathcal{P} \). If any default alternatives exist, these are accounted for by appending the negation of all other non-default alternative conditions.

- **E − CASE − DATA − SYM**
  Similar to **E − CASE − LIT − SYM**. Note that all of the data constructor parameters in the branched alternatives are treated as symbolic values. This is because a branching condition whose values depend on a symbolic value is itself also considered symbolic.

When we have a Return, this means that we are no longer able to evaluate the C as it is a value form. Instead, we must try to see what frames are on the Stk so that we may update or resolve them as necessary. The terminal state of execution is when we are in a value form and there are no more frames on the Stk to reduce. Note that it is possible to have non-terminating programs, such as in the case of recursive functions, which would have ways of invoking itself again in the expression body.

- **R − IDENTITY**
  Terminating evaluation state. There is nothing to be done.

- **R − UPDATE − FRAME − REDIR**
  The UpdateFrame that we pushed is overwritten with the corresponding value form.

- **R − UPDATE − FRAME − INSERT**
  If there did not previously exist a binding in the \( \mathcal{E} \), there exists one now that binds to the C that is a value.

- **R − CASE − FRAME**
  We re-add the C back into the CaseFrame of that it used to be part of. The case statement may now be evaluated because its inspected expression is in a value form.
• **R − APPLY − FRAME**

A previous ApplyFrame that we pushed onto the stack in favor of evaluating the LHS of an expression application is re-applied. Because expression application relies on the LHS to be a function form, this means that the LHS would be a lambda abstraction.

• **R − CAST − FRAME**

Now that the expression is in a value form, casting is more feasible as we do not have to deal with complicated structures that a general expression may have.

• **R − ASSUME − FRAME**

We append the assumptions made into the path constraints.

• **R − ASSERT − FRAME**

We append the negation of the assertions into the path constraint. This is because when we have assertions, we are looking for assertion failures: that is, the reachability of states where the negation of the assertion is satisfiable are considered errors. We further mark the boolean in the tuple as True to indicate that we have not yet violated an assertion.

• **R − CHECK − ASSERT − FRAME − TRUE**

The CheckAssertFrame checks whether or not we have assertion violation. In this case we have no violation, after confirmation with a solver, and may proceed with execution.

• **R − CHECK − ASSERT − FRAME − FALSE**

In this case there is an assertion violation given by the solver and this means that we have hit an unsatisfiable branch. We prune this state out.

### 3.4 Constraint solving

Here, we present optimizations to solving path constraints generated by G2. To reduce state explosion, we check the satisfiability of the path constraints every time we add a new path constraint, and immediately discard any states with unsatisfiable constraints. But checking the path constraints is itself quite time consuming, and so we aim to optimize it as much as possible.

We adapt the constraint independence optimization introduced by EXE, and refer the reader to [1] for a full description. In short, this optimization involves viewing our path constraints as nodes of an undirected graph, where two nodes are connected by an edge if and only if they share some variable. Then, to check the satisfiability of a newly added constraint, we need only consider those path constraints in it’s strongly connected component.

However, without careful handling, the pervasiveness of Algebraic Datatypes in functional languages can easily lead to very large strongly connected components. counteracting the constraint independence scheme. Consider the function averageF

```plaintext
averageF :: [Float] -> Float
averageF x = sumN x / lengthN x

sumN :: Num a => [a] -> a
sumN xs = case xs of
    x:xs -> x + sumN xs
    _ -> 0

lengthN :: Num b => [a] -> b
lengthN x = case xs of
    _:xs -> 1 + lengthN xs
    _ -> 0
```
averageF must walk over the list twice—once to compute the length, once to compute the sum. Suppose we generated fresh names for data constructor arguments in Rule E-CASE-DATA-SYM. There will be a state where sumN generates the path constraints \( \mathcal{PC}_{\text{sumN}} \):

\[
(\cdot) \quad x_0 \quad s_1 = s_0 \\
(\cdot) \quad x_1 \quad s_2 = s_1 \\
(\cdot) \quad x_2 \quad s_3 = s_2 \\
\neg \text{isCons} \quad (\cdot) \quad s_3 \\
F\# \quad f_0 = x_0 \\
F\# \quad f_1 = x_1 \\
F\# \quad f_2 = x_2
\]

These constraints are satisfiable, modeled by a list of three floats. Now consider the following constraints that will be generated by lengthN:

\[
(\cdot) \quad x'_0 \quad s'_1 = s_0 \\
(\cdot) \quad x'_1 \quad s'_2 = s'_1 \\
(\cdot) \quad x'_2 \quad s'_3 = s'_2
\]

The next constraint generated by lengthN will determine the constructor and arguments of \( s'_3 \). As there are two cases alts, we have two possible sets of constraints:

\[
\begin{align*}
\mathcal{PC}_{\text{length1}} \quad &\quad \mathcal{PC}_{\text{length2}} \\
(\cdot) \quad x'_0 \quad s'_1 = s_0 \quad &\quad (\cdot) \quad x'_0 \quad s'_1 = s_0 \\
(\cdot) \quad x'_1 \quad s'_2 = s'_1 \quad &\quad (\cdot) \quad x'_1 \quad s'_2 = s'_1 \\
(\cdot) \quad x'_2 \quad s'_3 = s'_2 \quad &\quad (\cdot) \quad x'_2 \quad s'_3 = s'_2 \\
\neg \text{isCons} \quad (\cdot) \quad s'_3 \quad &\quad (\cdot) \quad x'_3 \quad s'_4 = s'_3
\end{align*}
\]

Because of their constraints on \( s_3 \) and \( s'_3 \), \( \mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length1}} \) is satisfiable, whereas \( \mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length2}} \) is not. To determine the satisfiability of the new clause in either of \( \mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length1}} \) or \( \mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length2}} \) requires at least eight path components. But in fact, every path constraint on any part of the ADT is in the same strongly connected component, so, with a straightforward approach to constraint dependence, each would require all eleven path constraints. Furthermore, if there were later path constraints involving primitive operations on \( f_0, f_1, \) or \( f_2 \), these constraints would fall into the same strongly connected component as the constraints from the ADT walk. This makes it increasingly expensive to check the satisfiability of each newly added clause.

To reduce this interconnectedness, we use uniqArgs(\( x, n \)), and change our construction of the path constraints graph. Recall from Section 3.3 that uniqArgs returns the same unique \( n \) names, for each \( x \). When using it, we get the same path constraints from sumN. Considering the same two possible sets of constraints from lengthN, we will get:

\[
\begin{align*}
\mathcal{PC}_{\text{length1}} \quad &\quad \mathcal{PC}_{\text{length2}} \\
(\cdot) \quad x_0 \quad s_1 = s_0 \quad &\quad (\cdot) \quad x_0 \quad s_1 = s_0 \\
(\cdot) \quad x_1 \quad s_2 = s_1 \quad &\quad (\cdot) \quad x_1 \quad s_2 = s_1 \\
(\cdot) \quad x_2 \quad s_3 = s_2 \quad &\quad (\cdot) \quad x_2 \quad s_3 = s_2 \\
\neg \text{isCons} \quad (\cdot) \quad s_3 \quad &\quad (\cdot) \quad x_3 \quad s_4 = s_3
\end{align*}
\]

By itself, this has not gained us much—we have added more edges to our implication graph, but not actually made the strongly connected components any smaller. However, we now adjust our graph construction, to only connect DataCon nodes with edges based on the scrutinee, not the arguments. Then, checking the satisfiability of \( \mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length1}} \) requires only checking one constraint:

\[
\neg \text{isCons} \quad (\cdot) \quad s_3
\]

Similarly, determining that \( \mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length2}} \) is unsatisfiable requires only two constraints:

\[
\neg \text{isCons} \quad (\cdot) \quad s_3 \wedge (\cdot) \quad x_3 \quad s_4 = s_3
\]
4 Conclusion and Future Work

G2 integrates and improves upon existing software analysis techniques for Haskell, enabling us to perform analysis on a larger family of programs. Similar techniques can be applied to symbolic engines that can be developed for similar functional languages.

Currently we are working on integration of G2 into existing frameworks such as LiquidHaskell that can leverage G2 to perform utilities such as counterexample generation.

Furthermore, there are a number of optimizations such as garbage collection and execution order heuristics that are still to be tested, but should yield better runtime and memory performance.
Bibliography

