A Symbolic Execution Framework for Haskell

William Hallahan, Anton Xue, Ruzica Piskac
Yale University, New Haven, CT, USA
Email: william.hallahan@yale.edu anton.xue@yale.edu ruzica.piskac@yale.edu

1. Motivation

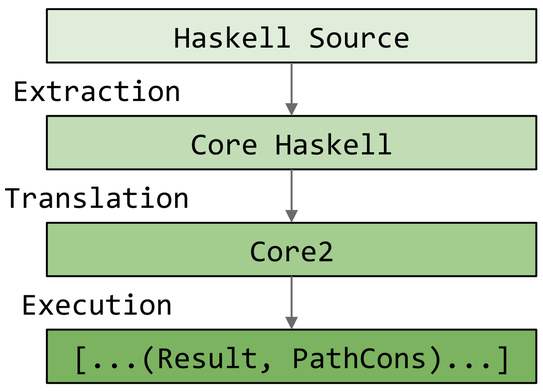
- Large software applications benefit from automated exploration and testing.
- Unassisted symbolic execution is easy to use and general purpose.
- Symbolic execution is easy to describe, but what about implementation?
- A natural complement to Haskell's language design.

Our Results

Generalized statically typed symbolic execution engine for purely functional programming languages with support for algebraic data types and models for higher-order functions.

https://github.com/AntonXue/G2

2. Workflow



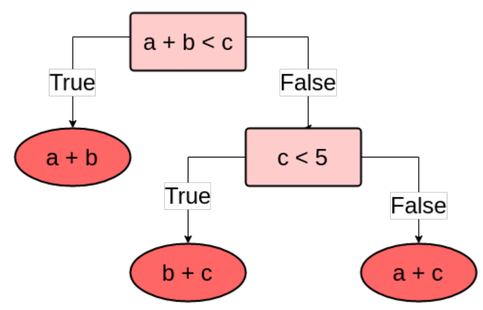
1. Use GHC (8.0.2) API to extract Core Haskell during compilation.
2. Translate Core Haskell to G2 Core, stripping away complex features.
3. Perform symbolic execution on G2 Core for list of result / path constraint pairs.

3. Symbolic Execution

Sample function:

```
> func a b c =
>     if (a + b < c)
>         then a + b
>         else if (c < 5)
>             then b + c
>             else a + c
```

Corresponding execution tree:



Branching at conditional statements.

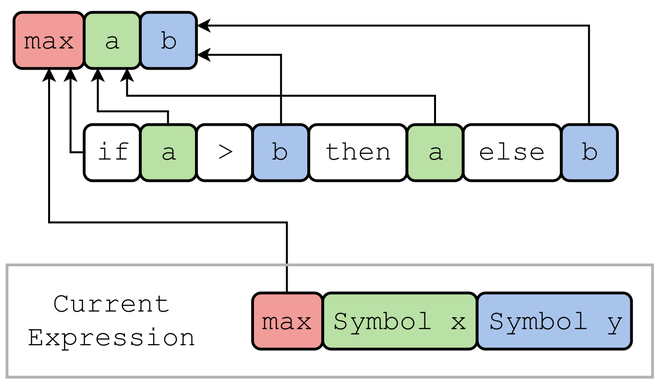
Execution Analysis

1. Assign symbolic values to function parameters.
   > env = {a → a_0, b → b_0, c → c_0}
2. Execute in terms of symbolic values.
   > if (a_0 + b_0 < c_0) then … else …
3. Keep track of conditional constraints.
   > C_A = (a_0 + b_0 < c_0)
   > C_B = (c_0 < 5)
4. Returns a list of results and their path constraints.
   > [P1 → {a_0 + b_0, (a_0 + b_0 < c_0)}
   , P2 → {b_0 + c_0, ¬(a_0 + b_0 < c_0)∧(c_0 < 5)}
   , P3 → (a_0 + c_0, ¬(a_0 + b_0 < c_0)∧¬(c_0 < 5)}]
5. [Optional] Solve each path constraint for values.
   > P1 → {a_0 → 1, b_0 → 1, c_0 → 3} ⇒ 2
   > P2 → {a_0 → 3, b_0 → 1, c_0 → 3} ⇒ 4
   > P3 → {a_0 → 6, b_0 → 1, c_0 → 6} ⇒ 12

4. Graph Reduction

Lazy purely functional programs can be seen as graphs. Variables act as pointers (edges) to other functions (vertices). Our graph reduction strategy is inspired by the Glasgow Haskell Compiler's strategy, but adapted to work with symbolic values.

```
> max :: Int -> Int -> Int
> Max a b = if a > b then a else b
```



Leftmost outermost reductions are repeatedly performed until in Weak Head Normal Form (a constructor, built-in function, or lambda expression).

```
fst :: (a, b) -> a
fst (a, b) = a

square :: Int -> Int
square x = x * x
```

Leftmost innermost reductions

```
fst (square 3, square 4)
 → fst (3 * 3, square 4)
 → fst (9, square 4)
 → fst (9, 4 * 4)
 → fst (9, 16)
 → 9
```

Leftmost outermost reductions

```
fst (square 3, square 4)
 → square 3
 → 3 * 3
 → 9
```

Symbolic execution requires extending the semantics to account for logical variables and track path constraints. A symbolic variable is always treated as already being in WHNF. Case statements on symbolic variables require splitting into multiple states.

Path constraints are always expressed in a normal form, which consist solely of data constructors, primitive types, symbolic variables, and primitive operators such as addition and multiplication. Case and lambda expressions are completely eliminated. This allows us to convert to SMT formulas, and find values satisfying the path constraints.

5. Challenge: Higher-Order Functions

Problem: How can we use SMT solvers with symbolic higher-order functions?

```
> f :: (Int -> Int) -> Int -> Int
> f g x = g . g $ x
```

Haskell has lambda expressions and partial application- these complicate solutions such as defunctionalization.

Potential solution: Introduce symbolic functions as datatypes, and convert these to functions immediately. Suppose we have 3 functions:

```
> f :: Int -> Int
> g :: Int -> Int
> h :: (Int -> Int) -> Int -> Int
```

And aim to symbolically execute h. We introduce a new type and function:

```
> data TIntInt = F | G
> intIntConvert :: TIntInt -> (Int -> Int)
> intIntConvert F = f
> intIntConvert G = g
```

Then we run our symbolic execution with symbolic variables `x :: TIntInt` and `y :: Int` on:

```
> h (intIntConvert x) y
```

Path constraints will be generated based on the introduced type, rather than the function type

Complication: What if we have a datatype with a function parameter?

```
> type IntIntList = [Int -> Int]
```

Potential solution : We create a new type and function to walk over the structure of that type, as such:

```
> type IntIntList' = [TIntInt]
> walk :: IntIntList' -> IntIntList
> walk [] = []
> walk (x:xs) = intIntConvert x:walk xs
```

This research was supported in part by NSF grant CCF-1553168