# Lazy Counterfactual Symbolic Execution

ANONYMOUS AUTHOR(S)

We introduce *counterfactual symbolic execution*, a new approach that produces counterexample executions that localize the causes of failure of static verification. Furthermore, we develop non-strict semantics for symbolic execution. We combine these two efforts, by applying counterfactual symbolic execution to programs written using LiquidHaskell. Our approach extends the classical notion of symbolic execution in three orthogonal ways. First, we develop a notion of *symbolic weak head normal form* and use it to extend symbolic execution to languages with lazy (non-strict) evaluation. Second, we introduce a *counterfactual branching* operator which is used to denote a choice between two implementations of some computation, and show how to localize errors by pinpointing branches where one choice leads to counterexamples while the other does not. Third, we show how to use symbolic execution to find counterexamples for refinement type errors by translating refinement types into assumes (for input types), asserts (for output types) and (recursive) traversals over inductive data types, and using counterfactual execution to localize whether the code or the specification of the function is to blame. Finally, we implement and evaluate our approach on a corpus of errors gathered from users of the LiquidHaskell refinement type system. We show that counterfactual symbolic execution is able to quickly find counterexamples for 77% of the errors, providing precise explanations of how the code or specifications need to be fixed to enable verification.

## 1 INTRODUCTION

Modular verifiers allow programmers to specify correctness properties of their code using function contracts, such as pre- and post- conditions (*e.g.* EscJava [Flanagan et al. 2002], Dafny [Leino 2010]), or refinement types (*e.g.* DML [Xi and Pfenning 1999], F* [Swamy et al. 2016]). Unfortunately, modular verifiers can be very difficult to use: when verification fails, the hapless programmer is given no feedback about why their code was rejected, let alone how they can fix it.

There are two ways in which (modular) verification can fail when checking if a function $f$ satisfies a contract given by a pre-condition $P$ and a post-condition $Q$. First, the *code* may be wrong. That is, the precondition $P$ may be too *weak*: *i.e.* the postcondition only holds on a smaller set of inputs than those described by the precondition. Alternatively, the postcondition $Q$ may be too *strong i.e.* the function's code is incorrect and establishes a weaker property than stipulated by the postcondition.

Second, more perniciously, the code of $f$ may be correct, but verification may still fail as the library functions' *contracts* may be wrong: the post-condition for some callee (library) function $g$ may not capture enough information about the values returned by that function in order to allow the desired property to be established at the caller (client) $f$. For example, consider the Dafny [Leino 2010] code shown in Figure 1. The verifier rejects this code, complaining that it cannot prove the postcondition for `main`. The problem here is not the code, which is clearly correct, but that as the contracts for `incr` are commented out, the default post-condition `True` is too weak to prove the post-condition that `main` returns a non-negative value. Indeed, if we uncomment those lines, Dafny readily verifies the program.

One might be tempted to use bounded model checking [Biere et al. 2003] or symbolic execution [King 1976] to enumerate paths through the code in order to find execution traces that witnesses the failure *i.e.* to find set of inputs that satisfy the precondition but which produce an output which violates the postcondition [Cadar and Sen 2013]. However, this approach falls short. Standard symbolic execution will be fruitless in the pernicious

```
method incr(x: int) returns (r: int)
// requires 0 ≤ x
// ensures  0 ≤ r
{
   r := x + 1;
}

method incr2(x: int) returns (r: int)
requires 0 ≤ x
ensures  0 ≤ r
{
  var tmp := incr(x);
      r   := incr(tmp);
}
```

Fig. 1. A Dafny program where `main` fails to verify due to a weak specification for `incr`.

case where the code actually satisfies the contract but verification fails due to imprecise specifications for *other* functions.

We introduce *counterfactual symbolic execution*, a new approach that produces counterexample executions that localize the cause of failure of static verification. Furthermore, we apply counterfactual symbolic execution to programs written with LiquidHaskell. However, classical symbolic execution implicitly assumes *eager* (call-by-value) semantics, but Haskell's semantics are non-strict. Consequently, classical symbolic execution fails to find many simple counterexamples under lazy evaluation, and dually, returns spurious counterexamples that are cannot arise with lazy evaluation. Thus, we developed *lazy symbolic execution*, capable of symbolically executing Haskell programs.

Thus, our approach extends symbolic execution in three orthogonal ways.

***1. Lazy Symbolic Evaluation:*** Our first contribution is to extend symbolic execution to languages with non-strict semantics (§ 3). As is standard in symbolic execution, we express contracts via *assume* and *assert* terms. However, we show that a symbolic search that does not account for laziness can miss simple counterexample traces that end with an assert failing. We solve this problem by extending classical lazy graph reduction based semantics with symbolic values. We then generalize the notion of WHNF, that is used to implement call-by-need semantics, to that of *Symbolic Weak Head Normal Form*. This ensures that the symbolic execution only computes values as needed, yielding a lazy symbolic execution framework that is guaranteed to find assert-violating traces if they exist.

***2. Counterfactual Branching:*** Our second contribution is a *counterfactual branching* operator which is used to denote a choice between *two* alternative implementations of some computation (§ 4). The two alternatives could be, for example, the actual *concrete* implementation of a function or an *abstract* implementation that is derived purely from the contract specified for the given function. Our key insight is that we can then determine whether an imprecise library contract is to blame by finding a counterfactual branch where *all* executions selecting the concrete choice are safe, while *some* execution along the abstract choice leads to an assertion failure.

***3. Refinement Types as Contracts:*** Our third contribution is to show how lazy counterfactual symbolic execution can be used to localize the cause of *refinement type* errors on Haskell programs (§ 5). Our key insight is that we can *translate* refinement types into assumes (for

input types), asserts (for output types) and (recursive) traversals over inductive data types, thereby converting the typing problem into one of finding an assertion failure. Following this reduction, plain symbolic execution can be used to synthesize inputs that yield counterexample inputs that show how a function fails to meet its refinement type specification. More importantly, counterfactual executions – where concrete implementation is the code itself, and the abstract implementation is a reification of the type specification – can be used to identify those library functions whose (weak) specifications are the root cause of the error.

***Implementation and Evaluation on LiquidHaskell:*** Finally, we present an implementation and empirical evaluation of our approach over a large dataset of 78 errors gathered from users of the LiquidHaskell refinement type system [Vazou et al. 2014] (§ 6). We show that counterfactual symbolic evaluation is able to quickly – in roughly 5 to 25 seconds per error – find concrete counterexamples for 36% of the errors. Furthermore, our search is able to find abstract counterexamples that pinpoint the correct library function whose specification is too weak for 41% of the errors, demonstrating both the importance and effectiveness of counterfactual counterexamples. Thus in all, our approach is able to synthesize inputs and executions that explain 77% of verification failures, and hence, could point the way towards making modular verification accessible.

## 2 OVERVIEW

We start with an overview of our goals, the challenges posed by lazy evaluation and refinement type error localization, and show how we solve these challenges via lazy counterfactual symbolic execution.

### 2.1 Goal: Symbolic Execution

Our first goal is to implement a symbolic execution engine for non-strict languages like Haskell. Such an engine would take as input a *program* like:

```
intersect :: (Eq a) => [a] -> [a] -> [a]
intersect [] _  = []
intersect _  [] = []
intersect xs ys = [x | x <- xs, any (x ==) ys]

any :: (a -> Bool) -> [a] -> Bool
any _ []     = False
any p (x:xs) = p x || any p xs
```

together with a *property*, specified as an assertion about the behavior of the program over some unknown inputs, *e.g.* that the `intersect` function above was commutative:

```
let xs = ?, ys = ?
in assert (xs `intersect` ys == ys `intersect` xs)
```

Our symbolic execution engine then symbolically evaluates all executions of the above program (up to some given number of reductions) to find a counterexample, that is, values for `xs` and `ys` under which the asserted predicate evaluates to `False`:

```
counterexample assert fails when
  xs = [0, 1]
  ys = [1, 1, 0]
```

```
let xs ! j = case xs of
               h:t -> case j == 0 of
                 True  -> h
                 False -> t ! j-1
    repl n = n : repl (n + 1)
    i = ?, k = ?
    v = repl i ! k
in assert (v == i)
```

Fig. 2. Program with assertion over a lazy list that strict analyzers would struggle with

## 2.2 Challenge: Lazy Evaluation

While there are several symbolic execution engines that can produce the above result [Cadar and Sen 2013], including those for functional languages like $F^{\#}$ [Tillmann and de Halleux 2008], Scala [Köksal et al. 2012], and Racket [Tobin-Hochstadt and Van Horn 2012; Torlak and Bodík 2014], all of these tools implicitly assume *strict* or *call-by-value* semantics. This proves problematic for a non-strict language (like Haskell) as a strict symbolic execution can *miss* assertion failures, and can report *spurious* failures that cannot occur under lazy evaluation.

***Strictness Reports Spurious Failures*** Consider the following code:

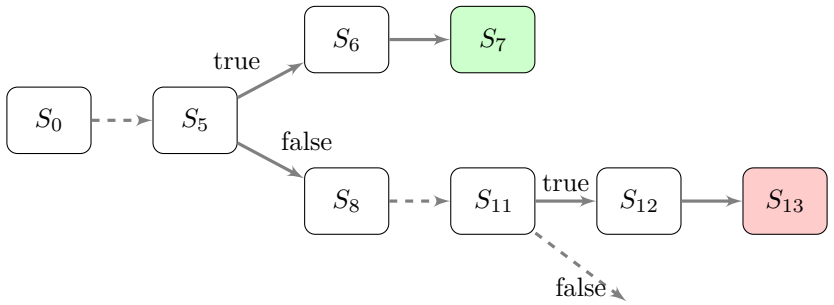
```
let f x = 10
    g _ = assert False
in f (g 0)
```

If this code was evaluated strictly, `g 0` would be evaluated first, triggering an assertion failure. However, under Haskell's nonstrict semantics, `f` is evaluated first, and immediately returns 10 without evaluating its argument. Thus, as `g 0` is never reduced, the assertion is never even evaluated and hence, does not fail!

***Strictness Misses Real Failures*** Even worse, strict symbolic execution can miss errors in code that relies explicitly on lazy evaluation. For example, consider the code in Figure 2. The code uses two functions, `!`, which returns the `j`-th element of the list `xs`, and `repl`, which returns an infinite list starting at `n` and asserts that the `k`-th element of `repl i` should be `i`. A strict symbolic execution will keep unfolding the infinite list corresponding to the term `repl i`, and thus, will *miss* the fact that the assertion can be violated simply by lazily evaluating the asserted predicate on a finite prefix.

## 2.3 Solution: Lazy Symbolic Execution

In this paper we solve the problems caused by strictness, by developing a novel lazy symbolic execution algorithm. At a high-level, our algorithm mimics the lazy graph reduction semantics of non-strict languages like Haskell, where terms are only reduced *by need*, up to *Weak Head Normal Form* (WHNF), *i.e.* enough to resolve pattern-match branches. Our key insight is that we can generalize the classical semantics to account for *symbolic values* that denote unknown inputs, by developing a notion of *Symbolic* WHNF, where terms are reduced up to symbolic variables whose values are constrained by *path constraints* that capture the branch information leading up to that point in the execution.




| | |
|---|---|
| $S_0$ | $E_0 = \texttt{let} \ldots \texttt{in} \, \texttt{assert}(v = i)$<br>$H_0 = \emptyset \qquad P_0 = \top$ |
| $S_1$ | $E_1 = \texttt{assert}(v = i)$<br>$H_1 = \{v \mapsto \texttt{repl}\ i\ \texttt{!}\ k, k \mapsto ?, i \mapsto ?, \texttt{repl} \mapsto \lambda n\,.\,n : \texttt{repl}\ (n+1)\} \qquad P_1 = \top$ |
| $S_2$ | $E_2 = \texttt{assert}(\texttt{repl}\ \texttt{!}\ k = i)$<br>$H_2 = H_1 \qquad P_2 = \top$ |
| $S_3$ | $E_3 = \texttt{assert}((\texttt{case}\ xs_3\ \texttt{of}\ \{\texttt{[]} \to \ldots; h{:}t \to \ldots\}) = i)$<br>$H_3 = \{xs_3 \mapsto \texttt{repl}\ i, j_3 \mapsto k\} \cup H_2 \qquad P_3 = \top$ |
| $S_4$ | $E_4 = \texttt{assert}((\texttt{case}\ (i : \texttt{repl}\ (i+1))\ \texttt{of}\ \{\texttt{[]} \to \ldots; h{:}t \to \ldots\}) = i)$<br>$H_4 = H_3 \qquad P_4 = \top$ |
| $S_5$ | $E_5 = \texttt{assert}((\texttt{case}\ j_3 = 0\ \texttt{of}\ \{\texttt{True} \to h_5; \texttt{False} \to t_5\ \texttt{!}\ (j_3 - 1)\}) = i)$<br>$H_5 = \{h_5 \mapsto i, t_5 \mapsto \texttt{repl}\ (i+1)\} \cup H_4 \qquad P_5 = \top$ |
| $S_6$ | $E_6 = \texttt{assert}(h_5 = i)$<br>$H_6 = H_5 \qquad P_6 = k = 0$ |
| $S_7$ | $E_7 = \texttt{assert}(i = i)$<br>$H_7 = H_6 \qquad P_7 = k = 0$ |
| $S_8$ | $E_8 = \texttt{assert}((t_5\ \texttt{!}\ (j_3 - 1)) = i)$<br>$H_8 = H_5 \qquad P_8 = k \neq 0$ |
| $S_9$ | $E_9 = \texttt{assert}((\texttt{case}\ xs_9\ \texttt{of}\ \{\texttt{[]} \to \ldots; h{:}t \to \ldots\}) = i)$<br>$H_9 = \{xs_9 \mapsto t_5, j_9 \mapsto (j_3 - 1)\} \cup H_8 \qquad P_9 = k \neq 0$ |
| $S_{10}$ | $E_{10} = \texttt{assert}((\texttt{case}\ ((i+1) : \texttt{repl}\ ((i+1)+1))\ \texttt{of}\ \{\texttt{[]} \to \ldots; h{:}t \to \ldots\}) = i)$<br>$H_{10} = H_9 \qquad P_{10} = k \neq 0$ |
| $S_{11}$ | $E_{11} = \texttt{assert}((\texttt{case}\ (j_9 = 0)\ \texttt{of}\ \{\texttt{True} \to h_{11}; \texttt{False} \to t_{11} : (j_9 - 1)\}) = i)$<br>$H_{11} = \{h_{11} \mapsto (i+1), t_{11} \mapsto \texttt{repl}((i+1)+1)\} \cup H_{10} \qquad P_{11} = k \neq 0$ |
| $S_{12}$ | $E_{12} = \texttt{assert}(h_{11} = i)$<br>$H_{12} = H_{11} \qquad P_{12} = k \neq 0 \wedge k - 1 = 0$ |
| $S_{13}$ | $E_{13} = \texttt{assert}(i + 1 = i)$<br>$H_{13} = H_{12} \qquad P_{13} = k \neq 0 \wedge k - 1 = 0$ |

Fig. 3. Symbolic Execution Tree for Example from Figure 2

***Symbolic States*** Symbolic execution evaluates a *State*, which is a triple $(E, H, P)$ comprised of an expression $E$ being evaluated, the heap $H$, mapping variables in $E$ to other (to be evaluated) expressions, and path constraints $P$, which are logical formulas constraining the values of symbolic variables in $E$ and $H$.

***Symbolic Execution Tree*** Figure 3 shows the *tree* of states resulting from performing symbolic execution on the code from Figure 2. Each node in the tree is labeled by a symbolic state, and has children nodes corresponding to the symbolic states that the (parent) node can transition to.

- *Initial State:* The root node of the tree corresponds to the initial symbolic state $S_0$, comprised of $E_0$, the source program expression, $H_0$, the initial empty heap, and $P_0$, the trivial path constraint (`True`).
- *Variable Binding:* The first transition from $S_0 \hookrightarrow^* S_1$ accounts for the sequence of let-bindings, which are *not* evaluated, but which, instead, are *bound* on the heap as shown in $S_1$, where the expression $E_1$ to be evaluated is the assertion, but where the heap $H_1$ contains the bindings. The symbolic variables $k$ and $i$ correspond to the (unknown) input values bound to `k` and `i` respectively.
- *Variable lookup and Application:* The transition from $S_1 \hookrightarrow S_2$ *looks up* `v` on the heap, replacing it with its definition `repl ! k`. The transition from $S_2 \hookrightarrow S_3$ looks up and *applies* the definition of the list index operator `!` to the arguments `repl i` and `k`. Notably, to model laziness, we *do not* evaluate the arguments to the function call, but instead generate fresh binders for the function parameters `xs` and `j` – namely, $xs_3$ and $j_3$ and bind them on the heap $H_3$.
- *Lazy Evalution up to Symbolic WHNF:* The transition from $S_3 \hookrightarrow S_4$ demonstrates how we model laziness. It looks up the definition of $xs_3$ – namely `repl i` and evaluates it only up to *symbolic* WHNF, *i.e.* enough to determine which of the patterns to branch on. That is, our lazy semantics perform exactly enough evaluation to determine that `repl i` corresponds to the non-empty list `i : repl (i+1)`. If our symbolic execution implemented strict semantics, the list would have to be completely evaluated *before* picking a case alternative, and since `repl` generates an inifinite list, this evaluation would never finish, regardless of how deep the symbolic execution searched.
- *Pattern Matching:* The transition from $S_4 \hookrightarrow S_5$ matches the non-empty list against the cons-pattern by introducing fresh binders $h_5$ and $t_5$, corresponding to the pattern variables, and binding them the the respective terms on the heap $H_5$.
- *Symbolic Branching:* At $S_5$ the scrutinized expression is $j_3 = 0$ which, after looking up the binding for $j_3$ in the heap $H_3$, is $i = 0$. This expression contains a symbolic value $k$ and hence, is in SWHNF, meaning that it could evaluate to `True` or `False`. Therefore, there are two possible transitions, to $S_6$ and $S_8$, and in each case we record the condition under which the transition occured by strengthening the path constraints $P_6$ and $P_8$ with the constraints $k = 0$ and $\neg k = 0$ respectively. The transition $S_6 \hookrightarrow S_7$ looks up $h_5$ to reduce the asserted predicate to a tautology $i = i$, meaning the assertion holds.
- *Recursive unfolding:* The symbolic execution continues to explore the other branch, $S_5 \hookrightarrow S_8$. Again, the binders are lazily looked up on the heap, with each application and pattern-match yielding fresh binders. Via a sequence of transitions we arrive at the symbolic branch denoted by $S_{11}$, where the head of the list is bound to the (symbolic) value $h_{11} = i + 1$.
- *Assertion Failure:* Again, at $S_{11}$ we have a symbolic branch on the term $k - 1 = 0$. This time, however, the `True` branch transitions to $S_{12}$ where the asserted predicate has been reduced to $h_{11} = i$. The transition $S12 \hookrightarrow S13$ looks up the value of $h_{11}$ in the heap $H_{11}$ to find that the asserted predicate is $i + 1 = i$ which is not `True`, and

thus, our lazy symbolic execution procedure reports a counterexample to the assertion in Figure 2.

Furthermore, we can obtain a satisfying assignment (*i.e.* a model) for the path constraints at the point of violation to obtain concrete values for the symbolic inputs $i$ and $k$ that lead to the failure. This allows us to determine concrete values that will lead to an assertion violation. For example, here, the SMT solver tells us that the assertion is violated when $k = 1$ and not, *e.g.* when $k = 0$.

### 2.4 Refinement Type Counterexamples

A refinement type constrains classical types with predicates in decidable first-order logics. For example, we can specify that the function `die` should never be called at run-time by assigning it the type:

```
die :: {x : String | false} -> a
die x = error x
```

The refinement type checker will verify that at each call-site, the function `die` is called with values satisfying the condition `false`. As no such value exists, the code will only typecheck if all calls to `die` are in fact, provably unreachable.

A restricted class of functions may be lifted into the refinement types to specify properties of algebraic data. For example, the following function computes the `size` of a list datatype:

```
data List a = Emp | (:+:) a (List a)

size :: List a -> Nat
size Emp         = 0
size (x :+: xs) = 1 + size xs
```

Using the `size` measure, one can write a safe version of the `head` function as:

```
head :: {v : List a | 0 < size v} -> a
head (x:_) = x
head _     = die "head: called with empty list!"
```

The input refinement type of `head` states that it is only called with non-empty lists whose size is strictly positive. As in the second equation the size is equal to 0, the second pattern is *inconsistent* with the input refinement, and hence, provably never reachable.

***Concrete Counterexamples*** It is often not obvious why a refinement type fails. For example, consider the `zipWith` function defined below:

```
zipWith :: (a -> b -> c)
        -> xs : List a
        -> {ys : List b | size xs > 0 => size ys > 0}
        -> List c
zipWith _ Emp         Emp         = Emp
zipWith _ (x :+: xs) (y :+: ys) = f x y :+: zipWith f xs ys
```

The function takes as arguments a function `f` and two lists, iterating over the elements of the lists and producing a new list whose elements are the result of applying `f` to elements of

the input lists. The above program is rejected by the refinement type checker LiquidHaskell [Vazou et al. 2014] with the vexing error:

```
32 | zipWith f (x :+: xs) (y :+: ys) = f x y :+: zipWith f xs ys
                                                            ^^

  Inferred type
    VV : {v : (List a) | v == ys}
  not a subtype of Required type
    VV : {VV : (List a) | size xs > 0 => size VV > 0}
```

This error can be more confusing than helpful. Instead, a counterexample that illustrates an instance where program execution violates the refinement types may provide better insight. Running our tool yields the following:

```
Counterexample:
    zipWith (0 :+: 0 :+: Emp) (0 :+: Emp)
makes a call to
    zipWith (0 :+: Emp) Emp
violating the refinement type of `zipWith`
```

The counterexample illustrates an input that will cause `zipWith` to recursively invoke itself with an input that violates the given precondition. With this information in hand, the user can see how to improve the refinement type (namely it is not enough that the second list be non-empty when the first is, we require that the lists have the *same* size.)

***Symbolic Execution for Refinement Types*** To find concrete counterexamples, our tool converts LiquidHaskell refinement types into `assume` and `assert` expressions and then uses symbolic execution to find inputs that violate the assertions. This conversion proceeds by using the types to generate *wrapped* versions of the functions where: (1) for each input value, we assume that the refinement for that input holds, so that we avoid spurious counterexamples that do not enjoy the given preconditions, (2) at each call-site, we assert that the refinements hold for the arguments, and (3) at each function return, we assert that the output refinements hold for the value being returned [Findler and Felleisen 2002]. The resulting program is then fed into the lazy symbolic execution engine and any counterexamples found are presented to the user as being concrete witnesses demonstrating why refinement typing fails.

## 2.5 Localizing Imprecise Refinement Types

Next, consider the homework exercise of implementing a `concat` function that concatenates a list of lists into a single list, with the goal of verifying that the size of the returned list is the sum of the lists in the input:

```
sumsize :: List (List a) -> Int
sumsize Emp         = 0
sumsize (x :+: xs) = size x + sumsize xs

concat :: x:List (List a) -> {v :List a | size v = sumsize x}
concat Emp                    = Emp
concat (xs :+: Emp)           = xs
concat (xs :+: (ys :+: xss)) = concat ((append xs ys) :+: xss)
```

```
append :: List a -> List a -> List a
append Emp         Emp = Emp
append xs          Emp = xs
append Emp         ys  = ys
append (x :+: xs) ys  = x :+: append xs ys
```

This `concat` implementation is correct, but is rejected by LiquidHaskell. To make verification modular, and hence, scalable, at each function call, LiquidHaskell is only aware of the information that is in the refinement type of the callee, and not the actual definition. Thus, when trying to verify `concat`, LiquidHaskell knows nothing about the value returned by `append`!

Thus, the above example illustrates a common, and most vexing situation where the verifier rejects a program, not because the property being checked does not hold (as in `zipWith`), but because the specifications for called functions are too *weak*. Worse, as the code is correct, we cannot report counterexamples as they do not exist!

***Abstract Counterexamples*** In this situation, the kind thing would be to point the user to function whose type needs to be tightened. We do so by introducing the notion of an *abstract counterexample*, where we show how the overall property can be violated by using an abstract implementation of the callee that is derived solely from the (refinement type) specification for the callee.

For example, when G2 is run on `concat` it reports the following abstract counterexample:

```
Abstract Counterexample:
    concat (Emp :+: (Emp :+: Emp)) = 0 :+: (1 :+: Emp)
makes a call to
    concat ((append Emp Emp) :+: Emp) = 0 :+: (1 :+: Emp)
violating the refinement type of `concat` when
    append Emp Emp = 0 :+: (1 :+: Emp)

Please strengthen the refinement type of `append`
to eliminate this possibility.
```

The abstract counterexample tells the user that the existing specification for `append` permits the call `append Emp Emp` to return the value `0 :+: 1 :+: Emp` which, in turn, causes the evaluation of `concat (Emp :+: (Emp :+: Emp))` to return a value that violates the specification of `concat`.

Crucially, the abstract counterexample points the user to the fact that the above only arises due to the (trivial) refinement type *specification* for `append` and not due to the actual *implementation* of the function. Inspired by this message, a user could improve the type refinement on `append` to:

```
append :: x : List a  -> y : List a
       -> {z : List a | size x + size y = size z}
```

which then lets LiquidHaskell verify `concat`.

***Counterfactual Symbolic Execution*** We can find abstract counterexamples like the above with a new technique called counterfactual symbolic execution. First, we introduce a *counterfactual* branching operator, essentially a non-deterministic choice operator that can evaluate either

| | | | |
|---|---|---|---|
| $e$ | ::= | | **Expressions** |
| | \| | $x$ | variable |
| | \| | $s$ | symbolic variable |
| | \| | $l$ | literal |
| | \| | $\lambda x \,.\, e$ | abstraction |
| | \| | $D$ | data constructor |
| | \| | $e\ e$ | application |
| | \| | $e \oplus e$ | primitive operation |
| | \| | $\texttt{let}\ x = e\ \texttt{in}\ e$ | let |
| | \| | $\texttt{case}\ e\ \texttt{of}\ \{\overline{a}\}$ | case |
| | \| | $\texttt{let}\ x\ = ? : \tau\ \texttt{in}\ e$ | symbolic let |
| | \| | $e \,\square\, e$ | counterfactual branch |
| | \| | $\texttt{assume}\ e\ \texttt{in}\ e$ | assumption |
| | \| | $\texttt{assert}\ e\ \texttt{in}\ e$ | assertion |
| | \| | $\texttt{CRASH}$ | assertion failure |
| | | | |
| $a$ | ::= | | **Alternatives** |
| | \| | $D\ \overline{x} \rightarrow e$ | constructor match |

Fig. 4. $\lambda_G$ grammar

of its two arguments. Second, we replace each function definition with a counterfactual branch that non-deterministically chooses between the actual concrete implementation, and an abstract version derived solely from the refinement type specification of the function.

We can then run symbolic execution as before, and report an abstract counterexample at those counterfactual branches where the concrete choice produces *no* counterexamples, but the abstract one does, in which case, as illustrated above, we can also report exactly how the abstract implementation leads to a property violation.

Next, we formalize lazy symbolic execution (§ 3), show how it can be extended with counterfactual branching (§ 4), and show how to produce concrete and abstract refinement type counterexamples (§ 5).

## 3 LAZY SYMBOLIC EXECUTION

Next, we present our approach by describing a core language $\lambda_G$ (§ 3.1) and formalizing *lazy* symbolic execution as a novel reduction semantics (§ 3.3). We then show how to extend this language with counterfactual branching § 4, and how to use the resulting framework to localize refinement type error localization § 5.

### 3.1 Syntax

Figure 4 summarizes the syntax of our core language $\lambda_G$, a typed lambda calculus extended with special constructs for symbolic execution.

- ***Terms*** include literals, variables, data constructors, function application, lambda abstraction, let bindings, and case expressions.
- ***Case*** expressions $\texttt{case}\ e\ \texttt{of}\ \{\overline{a}\}$ operate on algebraic data types. We refer to $e$ as the *scrutinee*, and to $\overline{a}$ as *alternatives*, each of which maps a *pattern* $D\ \overline{x}$ – comprising a constructor $D$ and a sequence of (bound) pattern variables $\overline{x}$ – to the expression that

| | | | |
|---|---|---|---|
| $S$ | ::= | $(E, H, P)$ | **State** |
| $E$ | ::= | $e$ | **Expression** |
| $H$ | ::= | $\{x \mapsto e\}$ | **Heap** |
| $P$ | ::= | $\wedge_i p_i$ | **Path Constraint** |
| $p$ | ::= | | **Logical Predicate** |
| | \| | $x = D\ \overline{x}$ | constructor binding |
| | \| | $b$ | boolean expression in SWHNF |

Fig. 5. Symbolic States

should be evaluated when the scrutinee matches the pattern. As is standard, Boolean branches correspond to a case-of over the patterns `True` and `False`.

- ***Symbolic Variables*** denote some unknown – typically input – value that will be constrained by various branches or assertions during symbolic execution. We assume that all symbolic binders are to *first order* values: higher-order values are orthogonal and can be handled via the approach of [Tobin-Hochstadt and Van Horn 2012].
- ***Symbolic let*** expressions $\texttt{let}\ x\ = ? : \tau\ \texttt{in}\ e$ bind the variable $x$ to a new symbolic variable that can be used in the expression $e$.
- ***Assume*** expressions $\texttt{assume}\ e_1\ \texttt{in}\ e_2$ *condition* the evaluation of $e_2$ upon whether $e_1$ evaluates to `True` and cause evaluation to halt otherwise. Assume terms are typically used to encode *preconditions* under which some term is evaluated.
- ***Assert*** expressions $\texttt{assert}\ e_1\ \texttt{in}\ e_2$ *check* that $e_1$ evaluates to `True` and cause evaluation to `CRASH` otherwise. Assert terms are typically used to encode the *postconditions* that must be established by some computation.
- ***Counterfactual*** branch expressions $e_1 \square e_2$ nondeterministically evaluates to either $e_1$ or $e_2$.

***Types*** Every expression in G2 has a type. We write $e : \tau$ to denote that $e$ has type $\tau$. Type checking for $\lambda_G$ is standard for polymorphic functional languages, *e.g.* the rules used in System $\mathrm{F}^{\uparrow}_{C}$ [Peyton Jones 1996], and hence are omitted for brevity. The $\texttt{assume}\ e_1\ \texttt{in}\ e_2$ and $\texttt{assert}\ e_1\ \texttt{in}\ e_2$ terms require that $e_1$ have the type `Bool`, and have the type of $e_2$. In the counterfactual branch, both expressions must have the same type.

### 3.2 Symbolic States

Next, we formalize the notion of *lazy symbolic execution* by presenting a new symbolic, non-strict operational semantics for $\lambda_G$ formalized via rules that show how a program transitions between symbolic states. Figure 5 summarizes the syntax of *symbolic states*, $S$ which are tuples of the form $(E, H, P)$. The *expression* $E$ corresponds to the term that is being evaluated. The *heap* $H$ is a map from (bound) variables $x$ to terms $e$. As is standard, the heap is used to store unevaluated thunks until the point at which they are needed. The *path constraint* $P$ is a conjunction of logical formulas that describes the values that (symbolic) variables must have in order for computation to have proceeded up to the given state. We will use the $P$ to capture the conditions under which evaluation proceeds along different case-branches.

***Well-formedness.*** Only symbolic variables may occur free in a state (in the $E$), all other variables are bound, either on the heap, or by a lambda, let, or case expressions. We denote the binding (insertion) of a variable $x$ to an expression $e$ in the heap $H$ as $H\{x = e\}$. We

write $\texttt{lookup}(H, x)$ for the expression to which $x$ is bound in $H$. If there is no binding for $x$ in $H$, $\texttt{lookup}(H, x)$ is not defined.

***Symbolic Expressions.*** The predicate $\mathsf{Sym}(e)$ checks if an expression has symbolic variables:

$$\mathsf{Sym}(e) = \begin{cases} \text{True} & s \\ \text{True} & e = e_1 \oplus e_2 \text{ and } \mathsf{Sym}(e_1) \vee \mathsf{Sym}(e_2) \\ \text{False} & \text{otherwise} \end{cases}$$

***Symbolic Weak Head Normal Form*** The essence of lazy semantics, *e.g.* in Haskell, is to reduce expressions to Weak Head Normal Form (WHNF) [Peyton Jones 1996], *i.e.* a literal, lambda abstraction, or data constructor application. Consequently, the heart of our *lazy* symbolic execution semantics is a notion of *Symbolic Weak Head Normal Form* (SWHNF), that generalizes WHNF to account for (unknown) symbolic values. Formally, we say that an expression $e$ is in SWHNF if the predicate $\mathsf{SWHNF}(e)$, defined as follows, holds:

$$\mathsf{SWHNF}(e) = \begin{cases} \text{True} & e \equiv l \\ \text{True} & e \equiv s \\ \text{True} & e \equiv D\ \overline{e} \\ \text{True} & e \equiv \lambda x\ .\ e \\ \text{True} & e \equiv e_1 \oplus e_2 \wedge (\mathsf{Sym}(e_1) \vee \mathsf{Sym}(e_2)) \\ \text{False} & \text{otherwise} \end{cases}$$

Thus, SWHNF generalizes WHNF to account for symbolic variables and primitive operations where at least one argument is symbolic.

## 3.3 Symbolic Transitions

We formalize lazy symbolic execution via the transition relation $S \hookrightarrow S'$ that says that the state $S$ takes a single step to the state $S'$. The transition relation is formalized via the rules shown in Figures 6 and 7. For some states, more than one rule applies, or there may be more than one way to apply a single rule. From the perspective of a single execution, this requires a nondeterministic decision to apply one of the rules. However, during symbolic execution, we split the state, by applying *each* potential rule, allowing us to explore all possible runs of the program up to some bounded number of transitions.

### Lazy Transitions

We now describe the reduction rules, shown in Figure 6, that formalize lazy execution.

***Bindings and Variables*** are implemented via a non-strict lazy semantics facilitated by the heap. Var looks up concrete variables in the heap, and returns the mapped result, which may be an arbitrary expression. Let and App-Lam both bind an expression in the heap, *without* evaluating the expression. App reduces the function in a function application, without reducing the arguments.

***Primitive*** operations require their arguments be evaluated. Pr-L and Pr-R evaluate the expressions in a primitive to SWHNF. If both the left and right arguments of a primitive are concrete literals, Pr allows that primitive to be evaluated concretely.

***Case*** expressions require the *scrutinee* be evaluated to SWHNF, so that the correct alternative can be picked. This evaluation is peformed by Case-Eval. If the scrutinee is concrete, Case-Data continues evaluation on the correct alternative expression. If the scrutinee is a symbolic variable, Case-Sym-Data nondeterministically chooses an alternative expression to evaluate.

$$\frac{e = \texttt{lookup}(H, x)}{(x, H, P) \hookrightarrow (e, H, P)}\textsc{Var} \qquad \frac{x' \text{ fresh} \quad e_1' = e_1[x'/x] \quad e_2' = e_2[x'/x]}{(\texttt{let } x = e_1 \texttt{ in } e_2, H, P) \hookrightarrow (e_2', H\{x' = e_1'\}, P)}\textsc{Let}$$

$$\frac{\neg\textsf{SWHNF}(f) \quad (f, H, P) \hookrightarrow (f', H', P')}{(f\ e, H, P) \hookrightarrow (f'\ e, H', P')}\textsc{App}$$

$$\frac{x' \text{ fresh} \quad e_1' = e_1[x'/x] \quad H' = H\{x' = e_2\}}{((\lambda x\ .\ e_1)\ e_2, H, P) \hookrightarrow (e_1', H', P)}\textsc{App-Lam}$$

$$\frac{(e_1, H, P) \hookrightarrow (e_1', H', P')}{(e_1 \oplus e_2, H, P) \hookrightarrow (e_1' \oplus e_2, H', P')}\textsc{Pr-L}$$

$$\frac{\textsf{SWHNF}(e_1) \quad (e_2, H, P) \hookrightarrow (e_2', H', P')}{(e_1 \oplus e_2, H, P) \hookrightarrow (e_1 \oplus e_2', H', P')}\textsc{Pr-R} \qquad \frac{l_1 \oplus l_2 = l}{(l_1 \oplus l_2, H, P) \hookrightarrow (l, H, P)}\textsc{Pr}$$

$$\frac{\neg\textsf{SWHNF}(e) \quad (e, H, P) \hookrightarrow (e', H', P')}{(\texttt{case } e \texttt{ of } \{\overline{a}\}, H, P) \hookrightarrow (\texttt{case } e' \texttt{ of } \{\overline{a}\}, H', P')}\textsc{Case-Eval}$$

$$\frac{\overline{x}' = x_1' \ldots \text{ fresh}}{(\texttt{case } D\ e_1 \ldots \texttt{ of } \{D\ \overline{x} \to e, \ldots\}, H, P) \hookrightarrow (e[\overline{x}'/\overline{x}], H\{x_1' = e_1 \ldots\}, P)}\textsc{Case-Data}$$

$$\frac{\textsf{Sym}(e) \quad \overline{x}' = x_1' \ldots \text{ fresh}}{(\texttt{case } e \texttt{ of } \{D\ \overline{x} \to e_a, \ldots\}, H, P) \hookrightarrow (e_a[\overline{x}'/\overline{x}], H, P \wedge e = D\ \overline{x}')}\textsc{Case-Sym-Data}$$

Fig. 6. Lazy Transition Rules

## Symbolic Transitions

We now turn our attention to the reduction rules in Figure 7, which focus on the constructs particular to symbolic execution.

***Counterfactual branches*** proceed by either Choice-L or Choice-R which allowing reduction on either $e_1$ or $e_2$.

***Symbolic binders*** are evaluated using Sym-Let which introduces a fresh symbolic value for $x$ that can be used in the body $e$.

***Assume*** expressions are executed by first reducing the predicate $e_p$ to SWHNF using Assume-Eval; once this is done, the rule Assume simply adds the predicate to the path constraint, thereby recording the constraint that the predicate must hold for computation to proceed.

***Assert*** expressions are handled similarly in that the predicate is first reduced to SWHNF. Next, we *check* that the predicate actually evaluates to `True`– otherwise execution `CRASH`-es due to an assertion violation. To this end, Assert-Crash queries the SMT solver for satisfying assignments of our symbolic variables, that falsify the predicate, *i.e.* which cause the predicate to evaluate to `False`. If the SMT solver finds such an assignment, we can show the user the inputs that cause the assertion violation. If no such assignment can be found, Assert proceeds to evaluate the inner expression $e_b$ under a strengthened path constraint.

$$\frac{}{(e_1 \,\square\, e_2, H, P) \hookrightarrow (e_1, H, P)}\textsc{Choice-L} \qquad \frac{}{(e_1 \,\square\, e_2, H, P) \hookrightarrow (e_2, H, P)}\textsc{Choice-R}$$

$$\frac{x' \text{ is a fresh symbolic variable}}{(\texttt{let}\ x\ = ? : \tau\ \texttt{in}\ e, H, P) \hookrightarrow (e[x'/x], H, P)}\textsc{Sym-Let}$$

$$\frac{(e_p, H, P) \hookrightarrow (e_p', H', P')}{(\texttt{assume}\ e_p\ \texttt{in}\ e_b, H, P) \hookrightarrow (\texttt{assume}\ e_{p'}\ \texttt{in}\ e_b, H', P')}\textsc{Assume-Eval}$$

$$\frac{\texttt{isSWHNF}\, e_p}{(\texttt{assume}\ e_p\ \texttt{in}\ e_b, H, P) \hookrightarrow (e_b, H, e_p' \wedge P)}\textsc{Assume}$$

$$\frac{(e_p, H, P) \hookrightarrow (e_p', H', P')}{(\texttt{assert}\ e_p\ \texttt{in}\ e_b, H, P) \hookrightarrow (\texttt{assert}\ e_p'\ \texttt{in}\ e_b, H', P')}\textsc{Assert-Eval}$$

$$\frac{\texttt{isSWHNF}\, e_p}{(\texttt{assert}\ e_p\ \texttt{in}\ e_b, H, P) \hookrightarrow (e_b, H, e_p' \wedge P)}\textsc{Assert}$$

$$\frac{\texttt{isSWHNF}\, e_p \qquad \texttt{isSMTSat}(\neg e_p \wedge P)}{(\texttt{assert}\ e_p\ \texttt{in}\ e_b, H, P) \hookrightarrow (\texttt{CRASH}, H, \neg e_p' \wedge P)}\textsc{Assert-Crash}$$

Fig. 7. Symbolic Transition Rules

***Concrete Transition Relation.*** We write $\hookrightarrow_c$ for the *concrete transition relation* obtained by replacing the rule Sym-Let with Conc-Let shown below, which replaces a symbolic binding with *some* expression of the suitable type:

$$\frac{H \vdash e' : \tau}{(\texttt{let}\ x\ = ? : \tau\ \texttt{in}\ e, H, P) \hookrightarrow_c (\texttt{let}\ x = e'\ \texttt{in}\ e, H, P)}\textsc{Conc-Let}$$

The concrete transitions correspond exactly to the usual standard non-strict operational semantics; there are *no* symbolic values anywhere, and the path constraint is just `True`.

***Completeness of Symbolic Execution*** Let $\hookrightarrow^*$ and $\hookrightarrow_c^*$ respectively denote the reflexive transitive closure of $\hookrightarrow$ and $\hookrightarrow_c$. We can prove by induction on the length of the transition sequences that if the concrete execution can `CRASH` then so can the symbolic execution:

**Theorem 1.** *$(e, \emptyset, \texttt{True}) \hookrightarrow_c^* (\texttt{CRASH}, \cdot, \cdot)$ iff $(e, \emptyset, \texttt{True}) \hookrightarrow^* (\texttt{CRASH}, \cdot, \cdot)$.*

## 4 COUNTERFACTUAL SYMBOLIC EXECUTION

Modular verifiers allow users to write and automatically check *contracts* (specifications, describing preconditions or postconditions) on functions. Unfortunately, verifications errors can be difficult for users, as error messages typically involve logical formulas, which may not be obviously linked to the written contract.

To help programmers understand why verification fails, we may try to use symbolic execution to find *concrete counterexamples* to contracts as shown in (§ 2.5). A concrete counterexample exists of inputs to a target function that obey the given preconditions (assumptions), but which either call some function in a way that violates its preconditions, or yield outputs that violate the target function's postcondition.

Unfortunately, such inputs do not not always exist. Some code can fail to verify due to modularity, *i.e.* because the contracts for a callee – and not its implementation – are

too imprecise to verify a caller. To give improved feedback in this second case, we introduce *abstract counterexamples*, found via counterfactual symbolic execution. As shown in (§ 2.5), counterfactual symbolic execution finds partial function definitions for directly called functions that obey their function contracts, but demonstrate why the caller's contract is violated.

Next, we show how to extend $\lambda_G$ with counterfactual symbolic execution, in order to generate abstract counterexamples.

***Contracts.*** To this end, we introduce three functions we require on the original contracts:

- the first, pre, returns a contract just specifying the preconditions
- the second, post, returns a contract just specifying the postconditions
- the third, toExpr, converts a contract to a $\lambda_G$ expression

Here, we assume these functions can be implemented for some arbitrary set of contracts. In § 5, we will show how these functions can be implemented for LiquidHaskell refinement types.

***Counterfactual Function Definitions*** To find abstract counterexamples, we create, for each function $f$, a *counterfactual function definition*, based on its implementation and contract. Given the definition of a function $f$ with a contract $c$, where

$$f \equiv \lambda \overline{x}.\ e$$

we define a counterfactual definition:

$$\hat{f} \equiv \lambda \overline{x}.\ (\texttt{assert}\ (\mathsf{toExpr}(c)\ \overline{x}\ e)\ \texttt{in}\ e) \square (\texttt{let}\ s\ = ?:\tau\ \texttt{in}\ \texttt{assume}\ (\mathsf{toExpr}(\mathsf{post}(c))\ \overline{x}\ s)\ \texttt{in}\ s)$$

Now when symbolic execution reduces $\hat{f}$, it binds the arguments to lambdas as usual. Then, due to the counterfactual branch, it splits into two symbolic states. We will refer to these as the *left* and *right* states, cooresponding to the left and right of the counterfactual chouce. The *left* state corresponds to normal execution, with an added assertion that ensures that the function produces values that satisfy its postcondition. In the *right* state, we introduce a new symbolic variable, $s$ that is constrained, via the assume, to satisfy the function's postcondition (as defined by the contract $c$), but which otherwise makes no use of $f$'s definition. Therefore, $s$ can take on any value that $f$ would be allowed to return by its postcondition. This allows us to find abstract counterexamples when $f$'s implementation is correct, but its contract does not describe its behavior precisely enough to verify a caller. When a function has been replaced by a symbolic variable, we refer to it as *abstracted*.

***Searching for Counterexamples*** We can find (abstract) counterexamples for a function $f$ of arity $n$, with contract $c$ by performing symbolic execution starting from an initial state defined as follows:

$$\texttt{assume}\ (\mathsf{pre}(c)\ \overline{x})\ \texttt{in}\ (f_{det}\ \overline{x})$$

Here, $\overline{x}$ are symbolic inputs that ensure that any counterexample we find will use inputs that satisfy $f$'s precondition. We create another special copy of $f$, called $f_{det}$. The function $f_{det}$ does not itself have the counterfactual branch, but has each occurence of callee functions $g$ replaced by $\hat{g}$. The above definition of $f_{det}$ matches how modular verifiers use the implementation of their client functions, by using the definition of $f$, but only the specifications for library functions when verifying that $f$ meets its specification.

***Concrete vs Abstract Counterexamples*** When an assertion violation concretely exists, we will find it by taking only *left* paths. When verification only fails because of imprecision in one or more contracts, we find this by taking *right* paths in those functions. We present

| | | | |
|---|---|---|---|
| $\tau$ | ::= | | **Types** |
| | \| | $\{v\colon b\,\tau \mid r\}$ | refinement |
| | \| | $x\colon \tau \to \tau$ | function |
| $b$ | ::= | | **Basic Types** |
| | \| | `Int` | integer |
| | \| | `Bool` | boolean |
| | \| | $A$ | algebraic data type |
| $r$ | ::= | | **Refinements** |
| | \| | $r = r$ | equality |
| | \| | $r < r$ | inequality |
| | \| | $r \wedge r$ | conjunction |
| | \| | $\neg r$ | negation |
| | \| | $x$ | variable |
| | \| | $m\,\overline{r}$ | measure application |
| | \| | $n$ | integer value |
| | \| | $r \oplus r$ | integer operation |
| | \| | `true` | true |
| | \| | `false` | false |
| $m$ | ::= | $m$ | **Measures** |

Fig. 8. $\lambda_D$ types

only those states which require the fewest *right* paths, as these most resemble a concrete execution, and are therefore the most understandable to a user.

In practice, presenting only states with a minimal number of abstracted functions also allows us to perform pruning as we perform symbolic execution. If we have found a state which violates an assertion, with $n$ abstracted functions, we can immediately drop any state in which we abstracted $n + 1$ or more functions.

## 5 REFINEMENT TYPE COUNTEREXAMPLES

Now that we have described a general technique for counterfactual symbolic execution, we turn our attention to leveraging it to generate refinement typing counterexamples.

***Refinement Types*** We support the language of refinement types shown in Figure 8. This subset includes operations on numeric types, measures (*e.g.* `size` from (§ 2.5)), and refinements on polymorphic arguments. In the refinement language, $\{v\colon b\,\tau_1 \ldots \tau_k \mid r\}$ represents the base type, $b$, refined by the predicate $r$. The $\tau_1 \ldots \tau_k$ are type arguments to the base type, which may themselves be further refined. The $v$ is an *inner bound name*, allowing reference to the value of the type in $r$ and $\tau_1 \ldots \tau_k$. The $x\colon \tau_1 \to \tau_2$ is a function of type $\tau_1$ to $\tau_2$. The $x$ is a *outer bound name* to refer to the value of $\tau_1$, allowing it to be referenced in refinements in $\tau_2$.

$$\mathsf{pre}(\tau) = \begin{cases} x_1\colon \tau_1 \to \mathsf{pre}(x_2\colon \tau_2 \to \tau_3) & \tau = x_1\colon \tau_1 \to x_2\colon \tau_2 \to \tau_3 \\ \tau_1 & \tau = x\colon \tau_1 \to \tau_2 \end{cases}$$

Fig. 9. $\lambda_D$ precondition

$$\mathsf{post}(\tau) = \begin{cases} x_1 \colon \mathsf{bType}(\tau_1) \to \mathsf{post}(\tau_2) & \tau = x \colon \tau_1 \to \tau_2 \\ \{v \colon b\ \tau_1 \ldots \tau_k \mid r\} & \tau = \{v \colon b\ \tau_1 \ldots \tau_k \mid r\} \end{cases}$$

$$\mathsf{bType}(\tau) = \begin{cases} \{v \colon b\ \mathsf{bType}(\tau_1) \ldots \mathsf{bType}(\tau_k) \mid true\} & \tau = \{v \colon b\ \tau_1 \ldots \tau_k \mid r\} \\ \tau & \text{otherwise} \end{cases}$$

Fig. 10. $\lambda_D$ postcondition

$$\mathsf{toExpr}(\tau) = \mathsf{toExpr}_\lambda(\tau, \tau)$$

$$\mathsf{toExpr}_\lambda(\tau, \tau_a) = \begin{cases} \lambda x\ .\ \mathsf{toExpr}_\lambda(\tau_2, \tau_a) & \tau = x \colon \tau_1 \to \tau_2 \\ \lambda x^F\ .\ \mathsf{toExpr}_\tau(x^F, \tau_a) \text{ for fresh } x^F & \tau = \{v_1 \colon b \ldots \mid r\} \end{cases}$$

$$\mathsf{toExpr}_\tau(x^F, \tau) = \begin{cases} \mathsf{toExpr}_b(x, \tau_1) \wedge \mathsf{toExpr}_\tau(x^F, \tau_2) & \tau = x \colon \tau_1 \to \tau_2 \\ \mathsf{toExpr}_b(x^F, \tau) & \tau = \{v \colon b\ \tau_1 \ldots \tau_k \mid r\} \end{cases}$$

$$\mathsf{toExpr}_b(x, \tau) = \begin{cases} \lambda v\ .\ p_b\ (v, \mathsf{toExpr}_\tau(x_1, \tau_1) \ldots \mathsf{toExpr}_\tau(x_k, \tau_k)) & \\ \quad \wedge\ \mathsf{toExpr}_r(r)\ x \text{ for fresh } x_1 \ldots x_k & \tau = \{v \colon b\ \tau_1 \ldots \tau_k \mid r\} \\ \text{True} & \tau = x \colon \tau_1 \to \tau_2 \end{cases}$$

$$\mathsf{toExpr}_r(r_1 = r_2) = \mathsf{toExpr}_r(r_1) = \mathsf{toExpr}_r(r_2)$$
$$\ldots = \ldots$$

Fig. 11. $\lambda_D$ to $\lambda_G$ translation

To use counterfactual symbolic execution for refinement types, we need only convert refinement type specifications to assume and assert expressions needed by $\lambda_G$. That is, we need only implement the three functions, pre, post, and toExpr, described in (§ 4), that describe the contracts of each function.

***pre and post*** The implementations of both pre and post, are shown in Figures 9 and 10. pre walks over the outermost functions and drops the return type. post keeps the bindings from the function, but sets all refinements, except the return type's refinement, to `True`. Keeping the bindings themselves is important, as they may be used in the return type's refinement.

***Converting Refinements to Contracts*** Refinement types are converted to contracts, *i.e.* asserts and assumes, on the inputs and output of a function. toExpr, shown in Figure 11, translates LiquidHaskell refinement types into predicates in $\lambda_G$. This function has many subparts:

- $\mathsf{toExpr}_\lambda$ Since refinements are written over the inputs and output of the function, we need names bound to each input and the output. $\mathsf{toExpr}_\lambda$ creates lambda bindings, to give us these names.
- $\mathsf{toExpr}_b$ **and** $\mathsf{toExpr}_r$ Each individual refinement on a type needs to be translated into a $\lambda_G$ *predicate* on a value. This translation is performed by $\mathsf{toExpr}_b$ and $\mathsf{toExpr}_r$.
- $\mathsf{toExpr}_\tau$ walks over the spine of a LiquidHaskell function type, conjoining the results of applying $\mathsf{toExpr}_b$ to each function.

***Polymorphic Data Types*** LiquidHaskell allows checking refinements on polymorphic type variables. For example, consider the polymorphic list type `[a]`. We may refine a list to contain only positive integers, by writing $\{x : \texttt{Int} \mid 0 < x\}$. Thus, we require a way to translate LiquidHaskell polymorphic type refinements, into predicates on expressions in $\lambda_G$. To do this for a type constructor $\tau$, with type variable $a$, a higher order function $p_\tau$ is automatically created. The function takes an expression of type $\tau\ a$, and a predicate of function type $a \to \text{Bool}$. It walks over the structure of the type, conjoining the application of the predicate to each occurence of $a$. We can then apply $p_\tau$ to a predicate expression and an expression of type $\tau$, to assume or assert that those predicate expressions hold on all type variables in $p$. For example, on a list, we have:

$$\begin{array}{llll} p_{\text{List}} & p & \texttt{[]} & = & \texttt{True} \\ p_{\text{List}} & p & (x : xs) & = & (p\ x)\ \wedge\ (p_{\text{List}}\ p\ xs) \end{array}$$

and so translate the type $\{x : \texttt{Int} \mid 0 < x\}$ to the $\lambda_G$ predicate $p_{\text{List}}\ \lambda x\ .\ x > 0$.

## 6 IMPLEMENTATION AND EVALUATION

Next, we describe our implementation of lazy, counterfactual symbolic execution, and present an evaluation that demonstrates the effectiveness of our method for localizing refinement type errors.

### 6.1 Implementation

We have implemented G2 in Haskell. We use the GHC API to parse Haskell programs, and use Z3 [De Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011] as SMT solving backends. G2 is open source, and available at **omitted for anonymity**. Our implementation supports a large Haskell98-like subset of the code compiled by GHC, additionally including features (such as case statements on literals, and default alternative cases) not shown in our formalism. Furthermore, G2 is also equipped with a custom version of Haskell's Base library and Prelude [Peyton Jones 2003]. For a select range of modules, functions, and datatypes, G2 can use this custom standard library to symbolically execute programs written with the standard Base and Prelude.

***Depth Bound*** Symbolic execution is a *bounded* approach to verification. G2 has a tunable *depth* bound parameter that limits the maximum length of a symbolic execution trace, *i.e.* the maximum number of symbolic reduction rules down each path generated during symbolic execution.

### 6.2 Quantitative Evaluation

***The goal*** of our evaluation is twofold. ***Q1*** Does symbolic execution find concrete counterexamples that explain refinement type errors? ***Q2*** Does counterfactual execution find abstract counterexamples that accurately pinpoint the functions whose specifications are too weak to permit type checking? We answer these questions by using G2 to generate counterexamples for refinement type errors on a corpus of programs written by students using LiquidHaskell for a homework assignment at (omitted for anonymity, collected under IRB #2014009900).

***Corpus*** In this particular assignment, students were to submit three different files that cumulatively implemented a $K$-means clustering algorithm based on `MapReduce`. On average, those three files were around 300 lines of code all together. To collect the dataset, we logged every file submitted to LiquidHaskell as the students worked on their assignment. Consequently, the data set comprises traces of files, giving us access to the code at different

stages of progression — both the incorrect programs and the correct one that finally type checked. For each student then, and for each file, we use a pair of versions: "good" : the latest version that is correct (*i.e.* passes the type checker), and "bad" : a version prior to "good" which parses, is complete – *i.e.* does not have `undefined` or other stubs – but *fails* refinement typing. For students who did not succeed in submitting error-free final versions of a particular file for their homework, we disregarded their submissions for that file type, because we would not be able to easily compare their earlier erroneous submission to a correct "ground truth" reference. Since it was possible for each file to contain multiple errors related to different functions, we recorded triples of (top-level function, bad-file, good-file). As a result, we obtained a corpus of about 80 bad-good pairs, spanning the three different file-types, that form the basis of our evaluation.

***Correctness Criteria*** For each such triple, we ran G2 on the "bad" file to symbolically execute the corresponding function, and recorded whether we succeeded in finding a counterexample, and if so, whether it was concrete or abstract. Every concrete counterexample that we found is a "smoking gun" – in that it clearly demonstrates an execution that witnesses why refinement typing fails, as the code simply does not satisfy the given specification. For abstract counterexamples, where G2 also provided suggestions for error localization (*e.g.* pinpointed *which* function's type should be strengthened), we compute a diff with the "good" file to see whether that function's specification was, in fact changed in order to facilitate typing. If so, we deem the abstract counterexample to be *genuine*, and if not, *i.e.* if the changes are (only) in some other part of the code, we deem the counterexample to be *spurious*.

***Experimental Setup*** An automated test script targeting the 78 triples was run to assist with evaluation. The machine used had a non-overclocked Intel Core i7-5600U, with 8 GB of RAM (DDR3L, 1600Mhz) and 16 GB of Swap (SSD model SAMSUNG_MZ7LN256 with swappiness of 40), running Linux Mint 18. We first gave a budget of 1000 reduction steps to `List.lhs` files, 3000 steps of reduction to `MapReduce.lhs` files, and 3000 steps of reduction to `KMeans.lhs` files. Additionally, all files were run on a timeout of 300 seconds with a restriction to yield at most 10 counterexamples. Z3 was used as the SMT backend. For functions and files on which G2 timed out or failed to generate counterexamples, we modified the reduction step budget to see if we could improve performance.

***Results*** Figure 12 summarizes the results of our evaluation.

- First, we observe that G2 finds counterexamples for 70 out of the 78 triples.
- Second, 28 counterexamples are *concrete*, indicating that nearly 36% of refinement type errors (at least in our dataset) occur when the code simply does not conform to the specification.
- Surprisingly, 32 counterexamples are *abstract* and *genuine*, meaning that counterfactual symbolic execution correctly pinpoints the function with the weak specification.
- Finally, 8 counterexamples are abstract but spurious, meaning that G2 pinpoints the wrong location due to reasons we discuss below.
- All together, we find that symbolic execution – and counterexamples – are very promising for explaining type errors. This is surprising because, unlike, for example, model checkers, type systems do not work by enumerating executions through the code. Nevertheless, this result shows that – as with plain non-refined type errors [Seidel et al. 2018] – most (all but 10 out of 78 in our corpus) statically ill-typed programs do admit dynamic witnesses.

| Function | # Error | # Concrete | # Abstract | # Spurious | Time(s) |
|---|---|---|---|---|---|
| List.lhs | | | | | |
| concat | 22 | 14 | 4 | 0 | 24.82 |
| prop_concat | 7 | 1 | 6 | 0 | 7.03 |
| zipWith | 1 | 1 | 0 | 0 | 6.94 |
| prop_zipWith | 1 | 0 | 1 | 0 | 8.17 |
| concat_2_lists | 1 | 1 | 0 | 0 | 5.50 |
| concatHelper | 1 | 1 | 0 | 0 | 5.99 |
| prop_map | 1 | 0 | 1 | 0 | 8.42 |
| foldr | 1 | 0 | 0 | 0 | NA |
| MapReduce.lhs | | | | | |
| mapReduce | 27 | 4 | 20 | 0 | 24.70 |
| collapse | 4 | 3 | 0 | 0 | 7.79 |
| expand | 3 | 3 | 0 | 0 | 4.87 |
| KMeans.lhs | | | | | |
| kmeans1 | 8 | 0 | 0 | 8 | 10.52 |
| mergeCluster | 1 | 0 | 0 | 0 | NA |
| Total | 78 | 28 | 32 | 8 | |

Fig. 12. G2 evaluation results for errors reported by LiquidHaskell on student homeworks. **# Error** is the number of benchmarks (bad-good pairs) with a refinement type error on this function; **# Concrete** is the number of concrete counterexamples reported by G2; **# Abstract** is the number of abstract counterexamples reported by G2 that *correctly* locate the function whose specification is too weak; **# Spurious** is the number of spurious counterexamples *i.e.* abstract counterexamples that pinpoint the wrong function; and **Time(s)** is the average number of seconds spent by G2 to find the counterexamples via symbolic execution.

***Spurious and Non-Reports*** G2 reports a spurious location for 8 benchmarks, and fails to produce any counterexample for 10. Both of these are, due to the limitations of symbolic evaluation, inherent in the very nature of the approach and specific to our current implementation. Recall that counterfactual execution reports an error at a function $f$ if the abstracted version of $f$ admits a trace ending with a `CRASH` but the concrete does not. All the spurious examples arose because, G2 was unable to find concrete counterexamples as they only occured at depths larger than the given parameter. We believe this problem can be addressed by applying the heuristics proposed to mitigate path-explosion in the symbolic execution literature [Cadar and Sen 2013]. Finally, we are still investigating the 10 cases where we found no counterexamples. We conjecture this is partly due to insufficient search depth, and partly because there are no counterexample traces as such: the errors arise due to the over-approximations made by syntax-directed refinement typing.

### 6.3 Qualitative Evaluation

We qualitatively evaluate our approach by describing some of the programs and the errors reported by G2.

***Example: Overly Strong Types*** One class of errors is when helper functions have overly and unnecessarily strong type signatures. For example, some students wrote the following code:

```
mapReduce :: (Ord k) => (a -> List (k, v)) -> (v -> v -> v)
          -> List a -> M.Map k v
mapReduce fm fr xs = kvm
  where
    kvs  = expand       fm xs -- step 1
    kvsm = group        kvs   -- step 2
    kvm  = collapse fr kvsm   -- step 3

expand :: (a -> List (k, v))
       -> {xs : List a | size xs > 0}
       -> {r : List (k, v) | size r > 0}
expand f xs = concat (map f xs)
```

The signature for `expand` states that it requires a non-empty `List` as input. However, the type signature of `mapReduce` does not guarantee this, and hence the code is rejected by LiquidHaskell with an error at the call to `expand` in the body of `mapReduce`. Running G2 on `mapReduce` yields the following concrete witness that clarifies the issue:

```
Concrete counterexample:
  mapReduce _ _ Emp = fromList []
makes a call to
  expand _ Emp = Emp
violating the refinement type of `expand`
```

***Example: Pinpointing a Weak Type Specification*** The following example illustrates a subtle example of how abstract counterexamples can pinpoint functions whose type specifications are too weak. One student attempted to define list concatenation in terms of a `foldr` function:

```
concat :: l1: List (List a) -> {l2: List a | size l2 = nestSize l1}
concat Emp        = Emp
concat (h :+: t) = foldr (\x a -> add x a) (concat t) h
```

The student attempts to verify that the result of `concat` yields a `List` whose size equals the sum of the sizes of the lists within its input. While the code indeed yields output lists that enjoy this property, LiquidHaskell is unable to verify it as we cannot even *express* the fact that `foldr` is a catamorphism using refinement types! On this program, G2 reports

```
Abstract counterexample:
  concat (Emp :+: Emp) = 0 :+: Emp
violating concat's refinement type when
  foldr ds (concat Emp) Emp = 0 :+: Emp
Please strengthen the refinement type of `foldr` to eliminate ...
```

This correctly points out that the problem is in the type of `foldr`. Indeed, we find that the student eventually came to the same conclusion, and that they could not find a suitable type for `foldr`. Thus, they worked around the problem by writing a recursive function that *inlines* `foldr`, yielding code that does permit the verification of `concat`:

```
concat :: l1: List (List a) -> {l2: List a | nestSize l1 = size l2}
```

```
concat Emp        = Emp
concat (h :+: t) = addList h (concat t)


addList :: l1: List a -> l2: List a
        -> {l3: List a | size l3 = size l1 + size l2}
addList Emp        l2 = l2
addList (h :+: t) l2 = h :+: addList t l2
```

***Example: Spurious Error*** All of our spurious reports arise due to submissions involving a particular function `kmeans1` in the file `KMeans.lhs`. (The intent of the `KMeans.lhs` assignment was for students write and annotate functions *other* than `kmeans1` itself. Hence, although students could modify `kmeans1`, all of the submissions for `kmeans1` were very similar.) When run on the ill-typed versions of `kmeans1` shown below:

```
kmeans1
  :: k: Nat -> n: Nat -> List (Point n) -> Cluster k n -> Cluster k n
kmeans1 k n ps cs = normalize newClusters
  where
    normalize   :: M.Map a Cluster -> M.Map a Point
    normalize   = M.map (\ (sz, p) -> centroid n p sz)
    newClusters = mapReduce fm fr ps
    fm p        = singleton (nearest k n cs p, (1 :: Int, p))
    fr wp1 wp2  = mergeCluster n wp1 wp2
```

G2 reported that the type of `mapReduce` — whose code is shown above — should be refined. However, this is a spurious report, as the problem was not with the specification of `mapReduce`. Instead, the problem here is that the `mergeCluster` function passed as a parameter to (the higher-order) `mapReduce` was ascribed the wrong type. This is causing the wrong type to be inferred for the *polymorphic instantiation* of `mapReduce`, leading to the refinement type error. Indeed in all the good versions of the above file, the student fixed the problem by writing the correct type for `mergeCluster`.

In a sense, counterexamples pointing to `mapReduce` are correct – the inferred refinement type of `mapReduce` does need to be strengthened. In principle, though, we would have wanted counterfactual execution to pinpoint `mergeCluster`, by finding a counterexample involving the concrete implementation of `mapReduce` but an abstract version for `mergeCluster`. However, in practice, perhaps due to the large search depth, G2 was unable to find such a counterexample, and instead pointed the blame at `mapReduce` (where it could find such a counterfactual counterexample.) In future work, we hope to address this problem by augmenting G2 with some of the heuristics proposed to tackle path-explosion during symbolic execution[Cadar and Sen 2013].

## 7 RELATED WORK

***Haskell Libraires, Program Analysis, and Testing*** Catch [Mitchell and Runciman 2008] and Reach [Naylor and Runciman 2007] are static analysis tools for Haskell. Catch finds pattern matching errors, while Reach aims to execute some marked part of a program. Both tools support a smaller subset of Haskell and aim to solve specific problems, rather than implement

general symbolic execution. To this end, they abstract away information we preserve, such as precise numeric values.

QuickCheck [Claessen and Hughes 2011] is a tool to test Haskell programs on random inputs. Similarly, SmallCheck [Runciman et al. 2008] is a tool to test Haskell programs on inputs below a certain size. For certain constraints, symbolic execution may be more successful at fully exploring a program than QuickCheck or SmallCheck. Furthermore, symbolic execution allows for different kinds of analysis then either of these tools. For example, while QuickCheck or SmallCheck could potentially be used to generate concrete counterexamples from LiquidHaskell types, neither could generate abstract counterexamples.

Type Targeted Testing [Seidel et al. 2015] presents a technique to test Haskell functions using refinement types. Inputs matching the refinement type's preconditions are generated using an SMT solver, and concretely passed to the function. Then, the values returned by the function is checked against the refinement type's postcondition. This is a very different technique than symbolic execution: it is more akin to the methodology of QuickCheck and SmallCheck Furthermore, unlike G2, Type Targeted Testing aims to be a test programs based on refinement types, rather than an aid to debug refinement types. As such, it offers no technique similar to G2's abstract counterexamples.

Smten [Uhler and Dave 2014] is a library to ease interaction between Haskell and SMT solvers. However, Smten seeks to leverage Haskell as way to formulate search problems and construct SMT queries, rather than direct analysis of Haskell programs themselves. Consequently, the design and goals of Smten and G2 are largely orthogonal.

***Haskell Verification*** Xu's work on static contract checking [Xu 2006], Halo [Vytiniotis et al. 2013], and LiquidHaskell [Vazou et al. 2014] all verify properties of Haskell programs. These techniques do not make use of symbolic execution to run the program – rather they rely on abstracting properties of the Haskell programs, to translate them into first-order logic.

***Symbolic Execution for Functional Languages*** CutEr [Giantsios et al. 2015] is a symbolic execution engine for Erlang programs. Unlike Haskell, Erlang is strict, non-currying, dynamically typed, and emphasizes code hot swapping as a feature.

[Tobin-Hochstadt and Van Horn 2012] and [Nguyễn and Van Horn 2015] uses symbolic execution to verify higher-order contracts, an orthoganal problem to us. ROSETTE [Torlak and Bodík 2014] is a solver aided language, capable of symbolic execution. These works are all targeted towards Racket, a dynamic language employing strict semantics, rather than a statically typed lazy languages. As demonstrated in (§ 2), non-strict semantics require a different approach to symbolic execution, and thus our work has different and new semantics, compared to these previous works.

Furthermore, our work introduces counterfactual counterexample generation for verifiers, a problem not considered by any existing work on symbolic execution.

***Symbolic Execution for Imperative Languages*** Most existing work on symbolic execution applies to imperative programming languages. Recent examples include Dart [Godefroid et al. 2005] for C, Symbolic Pathfinder [Păsăreanu et al. 2013] for Java, Sage [Godefroid et al. 2012] for x86 Windows applications, and EXE [Cadar et al. 2008b] and its sequel Klee [Cadar et al. 2008a] for LLVM. The execution semantics of imperative programs are quite different from ours, but techniques related to path search strategy or constraint solving are likely to be applicable to Haskell symbolic execution.

## 8 CONCLUSION

In this paper we highlight the importance and difficulty of lazy functional program analysis that eager analysis tools struggle with, and present the G2 symbolic execution engine augmented with novel symbolic lazy evaluation semantics to meet this challenge. We have implemented G2 to target Haskell, and demonstrated its effectiveness on producing both concrete and conuterfactual counterexamples to provide error localization and reachability analysis. Our work extends the theory and tooling of program analysis techniques to offer programmers novel methods for debugging software in lazy functional languages.

## REFERENCES


Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Advances in Computers* 58 (2003), 117–148. https://doi.org/10.1016/S0065-2458(03)58003-2

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008a. 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)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008b. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 10.

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*. 48–59. https://doi.org/10.1145/581478.581484

C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. 2002. Extended static checking for Java. In *PLDI*.

Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2015. Concolic Testing for Functional Languages. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*. ACM, New York, NY, USA, 137–148. https://doi.org/10.1145/2790449.2790519

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. ACM, 213–223.

Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2012. Constraints as control. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 151–164. https://doi.org/10.1145/2103656.2103675

K. R. M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*.

Neil Mitchell and Colin Runciman. 2008. 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)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/1411286.1411293

M. Naylor and C. Runciman. 2007. Finding Inputs that Reach a Target Expression. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 133–142. https://doi.org/10.1109/SCAM.2007.30

Phúc C Nguyễn and David Van Horn. 2015. Relatively complete counterexamples for higher-order programs. *ACM SIGPLAN Notices* 50, 6 (2015), 446–456.

Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (2013), 391–425.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Simon L Peyton Jones. 1996. Compiling Haskell by program transformation: A report from the trenches. In *European Symposium on Programming*. Springer, 18–44.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, Vol. 44. ACM, 37–48.

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2018. Dynamic witnesses for static type errors (or, Ill-Typed Programs Usually Go Wrong). *J. Funct. Program.* 28 (2018), e13. https://doi.org/10.1017/S0956796818000126

Eric L Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type targeted testing. In *European Symposium on Programming Languages and Systems*. Springer, 812–836.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Y. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *POPL*.

Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*. 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order symbolic execution via contracts. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 537–554.

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 530–541. https://doi.org/10.1145/2594291.2594340

Richard Uhler and Nirav Dave. 2014. Smten with Satisfiability-based Search. *SIGPLAN Not.* 49, 10 (Oct. 2014), 157–176. https://doi.org/10.1145/2714064.2660208

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 269–282.

Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: Haskell to Logic Through Denotational Semantics. *SIGPLAN Not.* 48, 1 (Jan. 2013), 431–442. https://doi.org/10.1145/2480359.2429121

H. Xi and F. Pfenning. 1999. Dependent Types in Practical Programming. In *POPL*.

Dana N Xu. 2006. Extended static checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. ACM, 48–59.