Lazy Counterfactual Symbolic Execution

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

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

Maxwell Troy Bland
Computer Science & Engg.
University of Califonia, San Diego
USA
mbland@eng.ucsd.edu

Ranjit Jhala
Computer Science & Engg.
University of Califonia, San Diego
USA
jhala@cs.ucsd.edu

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

Abstract

We present counterfactual symbolic execution, a new approach that produces counterexamples that localize the causes of failure of static verification. First, we develop a notion of *symbolic weak head normal form* and use it to define lazy symbolic execution reduction rules for non-strict languages like Haskell. Second, we introduce *counterfactual branching*, a new method to identify places where verification fails due to imprecise specifications (as opposed to incorrect code). Third, we show how to use counterfactual symbolic execution to *localize* refinement type errors, by translating refinement types into assertions. We implement our approach in a new Haskell symbolic execution engine, G2, and evaluate it on a corpus of 7550 errors gathered from users of the LiquidHaskell refinement type system. We show that for **97.7%** of these errors, G2 is able to quickly find counterexamples that show how the code or specifications must be fixed to enable verification.

CCS Concepts • **Theory of computation → Abstraction**; • **Computing methodologies → Symbolic and algebraic manipulation**.

Keywords symbolic execution, Haskell, lazy, counterfactual, counterexamples

ACM Reference Format:
William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy Counterfactual Symbolic Execution.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314618

In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314618

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 [9], Dafny [20]), or refinement types (*e.g.* DML [42], F* [33]). 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* and the postcondition may only hold 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 [20] code shown in Figure 1. The Dafny 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 the contract for `incr` is *too weak*: the post-condition that it returns a non-negative value is not enough to prove the post-condition that `main` returns `x + 2`.

One might be tempted to use bounded model checking [2] or symbolic execution [15] 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

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

method main(x: int) returns (r: int)
requires 0 ≤ x ensures r = x + 2
{ var tmp := incr(x); r := incr(tmp); }
```

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

but which produce an output which violates the postcondition [5]. However, this approach will be fruitless in the case where the code actually satisfies the contract but verification fails due to *imprecise specifications* for callee functions.

We introduce the novel concept of *abstract counterexamples* to help programmers debug errors due to imprecise specifications. An abstract counterexample for a function f and its callee g is a partial definition of g that satisfies g's contract, but creates a violation of f's contract. For the code in Fig. 1 we aim to find an abstract counterexample:

```
  main(0) = 0
violating the contract of 'main' if
  incr(0) = 0
Strengthen contract of 'incr'
to eliminate this possibility
```

The counterexample is a *partial* definition of the callee `incr` where `incr(0) = 0`. This definition satisfies `incr`'s contract but causes a violation of the caller `main`'s contract. The user can use the above to strengthen `incr`'s contract to `r == x + 1` to verify `main`.

In this paper, we develop and evaluate *lazy counterfactual symbolic execution*, a new technique to generate concrete and abstract counterexamples that localize the causes of failure of static modular verification for non-strict languages like Haskell. We do so via the following concrete contributions.

1. Lazy Symbolic Evaluation Our first contribution is a lazy symbolic execution engine (§ 3) for a language with non-strict semantics. Existing work on symbolic execution [3, 14, 24] uses laziness as an implementation technique to improve the efficiency of symbolic execution for languages with strict semantics. On the other hand, our work describes symbolic execution of Haskell, a language with *non-strict semantics*. Consequently, as we show in § 2.2, existing symbolic execution engines can fail to find simple counterexamples that arise with lazy evaluation. Similarly, they can return spurious counterexamples that are avoided by lazy evaluation.

We solve this problem by augmenting classical lazy graph reduction semantics [21, 28, 29] with symbolic variables to reduce terms into *Symbolic Weak Head Normal Form*, that only computes values as needed, thereby obtaining the first symbolic execution framework for a non-strict language.

2. Counterfactual Branching Our second contribution is the notion of *counterfactual branching* that allows us to simultaneously conduct a symbolic search for both concrete and abstract counterexamples (§ 4). A counterfactual branch denotes a choice between two alternative implementations of some function, *e.g.* the function's concrete implementation or an abstract one derived from the function's specification. Our key insight is that we can find abstract counterexamples by finding a counterfactual branch from which *all concrete* executions are safe, but from which *some abstract* execution leads to an error.

3. Refinement Types as Contracts Our third contribution is to show how to use counterfactual symbolic execution to localize the cause of *refinement type* errors (§ 5). We show how to translate refinement types into value-level assertions and where refinement type specifications for functions are translated into the abstract implementations to be used at counterfactual branches.

4. Implementation and Evaluation Our last contribution is an implementation of our approach as a tool, G2. We evaluate G2 on a corpus of 7550 refinement type errors from users of LiquidHaskell, a verification tool that has been used to verify various properties of the Haskell standard libraries [39] (§ 6). G2 is able to quickly find counterexamples **97.7%** of the time. **57.6%** of the time, G2 finds concrete counterexamples showing how the code fails the specification, and **40.1%** of the time it finds abstract counterexamples caused by an imprecise specification. By comparing the "error"-ing programs with their "fixed" versions we find that the abstract counterexamples *correctly* pinpoint the library function whose specification was too weak in **96.1%** of the cases, demonstrating the importance, effectiveness and practicality of counterfactual symbolic execution in making modular verification more usable.

2 Overview

We start with an overview of our goals and 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 xs ys = [x|x <- xs, any (x ==) ys]

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

```
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 = ?
in assert (repl i ! k == i)
```

Figure 2. Program with assertion over an infinite list that strict analyzers would struggle with.

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 engine then symbolically evaluates all executions of the above program (up to some given number of reduction steps) to find a counterexample, *i.e.* values for `xs` and `ys` under which the asserted predicate is `False`:

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

2.2 Challenge: Lazy Evaluation

While there are several symbolic execution engines that can produce the above result [5], including those for functional languages like $F^{\#}$ [34], Scala [16], and Racket [35, 37], all of these tools assume *strict* or *call-by-value* semantics. This is problematic for a non-strict language (like Haskell.) Strict evaluation can both *miss* assertion failures, and report *spurious* failures that cannot occur under lazy evaluation.

Strictness Reports Spurious Failures Consider:

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

Under strict evaluation, `g 0` would be computed first, violating the assertion. However, under non-strict semantics, `f` is evaluated first, and immediately returns `10` without evaluating its argument. Thus, as `g 0` is never reduced, the assertion is never 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`. The code asserts that the `k`-th element of `repl i` should be `i`. Strict symbolic execution will keep unfolding the infinite list corresponding to the term `repl i`, and thus, will *miss* that the assertion can be violated 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 (SWHNF), 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.

Symbolic States Symbolic execution evaluates a *State*, which is a triple (E, H, P) comprised of an expression E being evaluated, a heap H, mapping variables to other 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 symbolic executing the code in Figure 2. Each node is a symbolic state, and has children corresponding to the states that the parent node can transition to.

- *Initial State:* The initial symbolic state S_0 is comprised of E_0, the source program expression, H_0, the initial empty heap, and P_0, the trivial path constraint (`True`).
- *Variable Binding:* $S_0 \hookrightarrow S_1$ accounts for the let-bindings, which are *not* evaluated, but are, instead, *bound* on the heap as shown in S_1. The symbolic variables k and i correspond to the (unknown) input values.
- *Variable Lookup and Application:* $S_1 \hookrightarrow S_2$ looks up and *applies* the definition of the list index operator `!` to `repl i` and `k`. Due to laziness, we create fresh bindings on the heap, rather than evaluate the arguments.
- *Lazy Evalution to Symbolic WHNF:* $S_2 \hookrightarrow S_3$ looks up xs_2 – namely `repl i` – and lazily evaluates it to SWHNF, *i.e. precisely* enough to determine which of the patterns to branch on. Under strict semantics, the list would have to be completely evaluated *before* picking a case alternative, but since `repl` generates an infinite list, this evaluation would never terminate.
- *Pattern Matching:* $S_3 \hookrightarrow S_4$ matches the non-empty list against the cons-pattern by introducing fresh binders h_4 and t_4, and binding them to the the respective terms on the heap H_4.
- *Symbolic Branching:* At S_4, the scrutinized expression is $j_2 = 0$ which, after looking up j_2 in the heap, is $k = 0$. This contains a symbolic value k and hence, is in SWHNF, so it could evaluate to `True` or `False`. Therefore, there are two possible transitions, to S_5 and S_7. We strengthen the path constraints P_5 and P_7 with $k = 0$ and $\neg k = 0$ respectively, to record the condition under which the transition occured. $S_5 \hookrightarrow S_6$ looks up h_4 to reduce the asserted predicate to a tautology $i = i$, meaning the assertion holds.

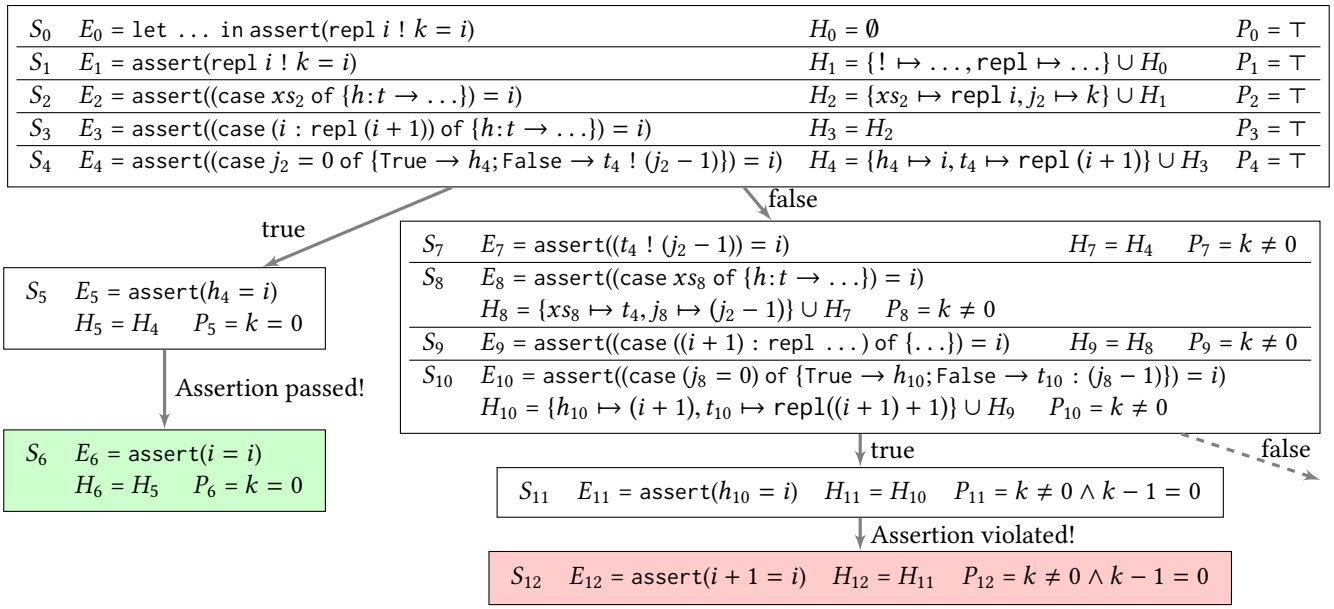

Figure 3. Symbolic Execution Tree for Example from Figure 2.

- *Recursive Unfolding:* The symbolic execution continues to explore the other branch, $S_4 \hookrightarrow S_7$. Again, the binders are lazily looked up on the heap. Via a sequence of transitions we arrive at S_{10}, where the head of the list is bound to the value $h_{10} = i + 1$.
- *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 $h_{10} = i$. $S11 \hookrightarrow S12$ looks up h_{10} in the heap to find that the asserted predicate, $i + 1 = i$, is not `True`. Thus, our symbolic execution reports a counterexample to the assertion in Figure 2.

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 that lead to the failure. This allows us to determine concrete values that violate the assertion. 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 refinement types to specify properties of algebraic data types. For example, the following function computes the `size` of a list:

```
size [] = 0
size (x:xs) = 1 + size xs
```

Using `size`, one can write a safe `head` function as:

```
head :: {xs:[a] | size xs > 0} -> a
head (x:xs) = x
head [] = die "Bad call to head"
```

The input refinement type of `head` states that it is only called with positively-sized lists. 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. Consider `zip`, defined below:

```
zip :: xs:[a] -> {ys:[b]
  | size xs > 0 => size ys > 0} -> [(a, b)]
zip [] [] = []
zip (x:xs) (y:ys) = (x, y):zip xs ys
zip _ _ = die  "Bad call to zip"
```

The function iterates over two lists and produces a new list of corresponding pairs. It is rejected by the refinement type checker LiquidHaskell [40] with the vexing error:

```
zip (x:xs) (y:ys) = (x, y):zip xs ys
                            ^^^^^^^^^

Inferred type
 VV : {v : [a] | size v >= 0 && len v >= 0
                 && v == ys}
not a subtype of Required type
 VV : {VV : [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:

```
zip [] [0] = error
makes a call to
die "Bad call to zip" = error
violating die's refinement type
```

The counterexample (`[] [0]`) illustrates an input that satisfies `zip`'s precondition, but causes `zip` to invoke the `die` function. 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.)

2.5 Localizing Imprecise Refinement Types

Next, consider `concat`, which 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 sizes of the lists in the input:

```
sumsize [] = 0
sumsize (x:xs) = size x + sumsize xs

concat :: x:[[a]] -> {v:[a]
                      | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) =
  concat ((append xs ys):xss)

append xs [] = xs
append [] 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 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 confusing, situation where the verifier rejects a program, not because the property being checked does not hold (as in `zip`), but because the specifications for called functions are too *weak*. Worse, as the code is correct, we cannot report counterexamples, since they do not exist.

Abstract Counterexamples In this situation, ideally we would point the user to the 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, an abstract counterexample for `concat` is:

```
  concat [[0], []] = [0, 0]
violating its refinement type, if
  append [0] [] = [0, 0]
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 [0] []` to return `[0, 0]`, causing the evaluation of `concat [[0], []]` to return a value that violates its specification.

Crucially, the abstract counterexample points the user to the fact that the error 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:[a]  -> y:[a]
  -> {z:[a] | size x + size y = size z}
```

which then lets LiquidHaskell verify `concat`.

Counterfactual Symbolic Execution We can find both concrete and abstract counterexamples with a new technique called counterfactual symbolic execution. We introduce a *counterfactual* branching operator, essentially a non-deterministic choice operator that can evaluate either of its two arguments. Each function definition is replaced with a counterfactual branch that non-deterministically chooses either the concrete implementation, or an abstract version derived solely from the function's refinement type.

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 this case, as illustrated above, we can also report exactly how the abstract implementation leads to a property violation.

3 Lazy Symbolic Execution

Here, we describe a core language λ_G (§ 3.1), which draws inspiration from GHC's Core language [27]. We formalize *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 errors (§ 5).

3.1 Syntax

Figure 4 summarizes the syntax of our core language λ_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)

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
	|	$? : \tau$	symbolic generator
	|	$e \,\square\, e$	counterfactual branch
	|	$\texttt{assume}\ e\ \texttt{in}\ e$	assumption
	|	$\texttt{assert}\ e\ \texttt{in}\ e$	assertion
	|	CRASH	assertion failure
a	::=	$D\,\overline{x} \rightarrow e$	**Alternatives**

Figure 4. λ_G grammar

pattern variables $\overline{x}$ – to the expression that 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 value. We assume that all symbolic binders are to *first order* values: higher-order values are orthogonal and can be handled via the approach of [35].
- ***Symbolic generator*** expressions $? : \tau$ are used to introduce new symbolic variables of type τ.
- ***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.
- ***Assert*** expressions $\texttt{assert}\ e_1\ \texttt{in}\ e_2$ *check* that e_1 evaluates to `True` and cause evaluation to CRASH otherwise.
- ***Counterfactual*** branch expressions $e_1 \,\square\, e_2$ nondeterministically evaluates to either e_1 or e_2.

Types Every expression has a type. We write $e : \tau$ to denote that e has type τ. Type checking λ_G is standard for polymorphic functional languages, *e.g.* the rules used in System $\mathrm{F}^{\uparrow}_{C}$ [29], and is omitted for brevity. $\texttt{assume}\ e_1\ \texttt{in}\ e_2$ and $\texttt{assert}\ e_1\ \texttt{in}\ e_2$ require that e_1 have type `Bool`. In a 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 λ_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 (*i.e.* unevaluated expressions) until the

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

Figure 5. Symbolic States

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 P to capture the conditions under which evaluation proceeds along different case-branches.

Well-formedness Only symbolic variables may occur free in a state, all other variables are bound, either on the heap, or by a lambda, let, or case expressions. We denote the binding 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 such binding, $\texttt{lookup}(H, x)$ is not defined.

Symbolic Variables and Primitive Applications $\mathrm{Sym}(e)$ checks if an expression is a symbolic variable, or is a primitive application that cannot be concretely reduced:

$$\mathrm{Sym}(e) = \begin{cases} \text{True} & e = s \\ \text{True} & e = e_1 \oplus e_2 \wedge (\mathrm{Sym}(e_1) \wedge \mathrm{Sym}(e_2)) \\ \text{True} & e = e_1 \oplus l \wedge \mathrm{Sym}(e_1) \\ \text{True} & e = l \oplus e_2 \wedge \mathrm{Sym}(e_2) \\ \text{False} & \text{otherwise} \end{cases}$$

Symbolic Weak Head Normal Form The essence of non-strict semantics, *e.g.* in Haskell, is to reduce expressions to Weak Head Normal Form (WHNF) [29], *i.e.* a literal, lambda abstraction, or data constructor application. Consequently, the heart of our *lazy* symbolic execution is a notion of *Symbolic Weak Head Normal Form* (SWHNF), that generalizes WHNF to account for (unknown) symbolic values. Formally, an expression e is in SWHNF if the predicate $\mathrm{SWHNF}(e)$ holds:

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

3.3 Symbolic Execution 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 in Figures 6 and 7. For some states, more than one rule

applies, or there is 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 program runs up to some bounded number of transitions.

3.3.1 Lazy Transitions

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

Bindings and ***variables*** are implemented via lazy evaluation facilitated by the heap. Var and Var-Red lookup a concrete variable, x, in the heap, to find the expression it is mapped to, e. If e is already in SWHNF, it is simply returned by Var. Otherwise, Var-Red reduces e to an expression, e', in SWHNF, before both returning e', and remapping x to e' in the heap. Typically, Var-Red is simply an optimization in case x is reevaluated: since Haskell is pure, evaluating e repeatedly would be semantically correct, but inefficient [21]. However, in Section 3.3.2, we will see that during symbolic execution with symbolic generators or counterfactual branching, this rule takes on a new importance.

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 arguments are evaluated to SWHNF by Pr-L and Pr-R. If both of the arguments of a primitive are concrete literals, Pr evaluates the primitive concretely.

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

3.3.2 Symbolic Transitions

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

Counterfactual branches proceed nondeterministically by either Ch-L or Ch-R, allowing reduction on either e_1 or e_2.

Symbolic generators are evaluated using Sym-Gen, which introduces a fresh symbolic value s.

Assume expressions are evaluated by first reducing the predicate e_p to SWHNF using Assume-Ev. Then, the rule Assume adds the predicate to the path constraint, thereby recording 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.

3.3.3 Impurity of Symbolic Transition Rules

Unlike GHC's Core Haskell, λ_G is impure, due to Symbolic Generators and Counterfactual Branching. For instance, consider the λ_G program in Figure 8a. This program is reducible to four different values in SWHNF: `4`, `2 * s`, `s * 2`, or `s * s'` (where `s` and `s'` are symbolic variables.) The evaluation of `f 2` may vary, even in a single reduction.

When symbolically executing the program, we explore each of these 4 branches separately. However, it is often desirable to require two values to match within an individual state. For example, we might wish to ensure that both calls to `f 2` either result in 2, or result in the same symbolic value.

We can achieve this with the program shown in Figure 8b. In a strict, call by value language, it would be clear why this program achieved the desired result: `y` would be computed only once, during the let binding, and before the evaluation of the multiplication. In a lazy setting, `f 2` is stored as a thunk, and only computed when forced by the multiplication. A natural question then arises: why does this program work in our lazy setting?

This is a result of us taking advantage of the Var-Red rule. During normal execution, this rule is just an optimization, but during symbolic execution, it allows us to control nonpurity. In the modified program, in Figure 8b, it means that, even though the reduction is performed only when needed, the reduction of `y` (and thus the reduction of `f 2`) is still performed only once. Thus, there are only 2 possible values in SWHNF: `4`, and `s * s`.

3.3.4 Completeness of Symbolic Execution

We write $\hookrightarrow_c$ for the *concrete transition relation* obtained by replacing the rule Sym-Gen with Conc-Gen, shown below, which replaces a symbolic generator with *some* total expression of the suitable type:

$$\frac{H \vdash e' : \tau}{(\,? : \tau\,, H, P) \hookrightarrow_c (e', H, P)}\,\textsc{Conc-Gen}$$

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

$$\frac{e = \texttt{lookup}(H, x) \quad \text{SWHNF}(e)}{(x, H, P) \hookrightarrow (e, H, P)}\textsc{Var} \qquad \frac{e = \texttt{lookup}(H, x) \quad \neg\text{SWHNF}(e) \quad (e, H, P) \hookrightarrow (e', H', P')}{(x, H, P) \hookrightarrow (e', H'\{x = e'\}, P')}\textsc{Var-Red}$$

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

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

$$\frac{(e_1, H, P) \hookrightarrow (e_1', H', P')}{(e_1 \oplus e_2, H, P) \hookrightarrow (e_1' \oplus e_2, H', P')}\textsc{Pr-L} \qquad \frac{\text{SWHNF}(e_1) \quad (e_2, H, P) \hookrightarrow (e_2', H', P')}{(e_1 \oplus e_2, H, P) \hookrightarrow (e_1 \oplus e_2', H', P')}\textsc{Pr-R}$$

$$\frac{l_1 \oplus l_2 = l}{(l_1 \oplus l_2, H, P) \hookrightarrow (l, H, P)}\textsc{Pr} \qquad \frac{\neg\text{SWHNF}(e) \quad (e, H, P) \hookrightarrow (e', H', P')}{(\texttt{case } e \texttt{ of } \{\overline{a}\}, H, P) \hookrightarrow (\texttt{case } e' \texttt{ of } \{\overline{a}\}, H', P')}\textsc{Case-Ev}$$

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

Figure 6. Lazy Transition Rules

$$\frac{}{(e_1 \,\square\, e_2, H, P) \hookrightarrow (e_1, H, P)}\textsc{Ch-L} \qquad \frac{}{(e_1 \,\square\, e_2, H, P) \hookrightarrow (e_2, H, P)}\textsc{Ch-R} \qquad \frac{s \text{ fresh}}{(\ ?:\tau\ , H, P) \hookrightarrow (s, H, P)}\textsc{Sym-Gen}$$

$$\frac{(e_p, H, P) \hookrightarrow (e_p', H', P')}{(\texttt{assume } e_p \texttt{ in } e_b, H, P) \hookrightarrow (\texttt{assume } e_p' \texttt{ in } e_b, H', P')}\textsc{Assume-Ev} \qquad \frac{\text{SWHNF}(e_p)}{(\texttt{assume } e_p \texttt{ in } e_b, H, P) \hookrightarrow (e_b, H, e_p \wedge P)}\textsc{Assume}$$

$$\frac{(e_p, H, P) \hookrightarrow (e_p', H', P')}{(\texttt{assert } e_p \texttt{ in } e_b, H, P) \hookrightarrow (\texttt{assert } e_p' \texttt{ in } e_b, H', P')}\textsc{Assert-Ev} \qquad \frac{\text{SWHNF}(e_p)}{(\texttt{assert } e_p \texttt{ in } e_b, H, P) \hookrightarrow (e_b, H, e_p \wedge P)}\textsc{Assert}$$

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

Figure 7. Symbolic Transition Rules

```
let f = λx . x □ ? in f 2 * f 2
```

(a) A program which evaluates `f 2` twice.

```
let f = λx . x □ ?; y = f 2 in y * y
```

(b) A program which evaluates `f 2` once.

Figure 8. Two impure λ_G programs.

4 Counterfactual Symbolic Execution

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

As discussed in § 2.4 and § 2.5 we use symbolic execution to find two types of counterexamples. Ideally, we find *concrete* counterexamples, *i.e.* actual function inputs that violates a contract. However, we also introduce *abstract counterexamples*, found via *counterfactual* symbolic execution, to help debug spurious errors. 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 not verified.

Our goal, then, is to find a *minimally abstract* or *least abstracted* counterexample- either a concrete counterexample, or a counterexample with a minimal number of abstracted functions. Such states are likely to be the most understandable to a user, as they most closely resembly an actual execution of the program.

Contracts To this end, we introduce three functions that we require on the original contracts: pre returns just the preconditions, post returns just the postconditions, and toExp converts a contract to a λ_G expression. Here, we assume these functions can be implemented for some arbitrary set of contracts. In § 5, we show these functions over LiquidHaskell refinement types.

Counterfactual Function Definitions To find abstract counterexamples, we create *assertion functions* and *counterfactual functions*. Given a function $f \equiv \lambda\overline{x}.\ e$ with a contract c, we define its assertion function as:

$$f^a \equiv \lambda\overline{x}.\texttt{let}\ r = f\ \overline{x}\ \texttt{in assert}\ (\text{toExp}(c)\ \overline{x}\ r)\ \texttt{in}\ r$$

We define the counterfactual function of f as:

$$\widehat{f} \equiv \lambda\overline{x}.\ f^a\ \overline{x}\ \square$$
$$(\texttt{let}\ s\ = ?:\tau\ \texttt{in assume}\ (\text{toExp}(\text{post}(c))\ \overline{x}\ s)\ \texttt{in}\ s)$$

When symbolic execution reduces $\widehat{f}$, 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, corresponding to the left and right of the counterfactual choice. The *left* state corresponds to normal execution, with an assertion that both ensures that the function's preconditions are met, and that the function returns values that satisfy its postcondition. In the *right* state, we introduce a new symbolic variable, s, that is assumed 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. The *right* state does not check that its arguments satisfy its preconditions, because if there is a violation of a precondition, it will also occur in the *left* case.

We can find (abstract) counterexamples for a function f of arity n, with contract c. To do so, we define another special copy of f, called f_{det}. The function f_{det} is f, but with each occurence of a callee function g replaced by $\widehat{g}$. This 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.

Then we perform symbolic execution starting from an initial state defined as follows:

$$\texttt{assume}\ (\text{toExp}(\text{pre}(c)\ \overline{s}))\ \texttt{in}\ ((f_{det})^a\ \overline{s})$$

$\overline{s}$ are symbolic inputs that ensure that any counterexample we find use inputs satisfying f's precondition.

In order to find minimally abstract counterexamples, we maintain a counter of the number of *right* paths selected for each states. Then, we filter the found states, and present only those which require the fewest abstracted functions.

4.1 Search Strategy

Symbolic execution, as described in this paper, is an unbounded and therefore incomplete search technique. When searching for counterexamples, we aim to minimize the number of abstracted functions, but we can almost never actually prove we succeeded (and an incompletely minimized counterexample may still be useful to a user.) Here, we describe two strategies we employ to try to minimize time spent searching, while still finding useful, and close to minimal, counterexamples.

Abstract Counterexample Filtering Presenting only minimally abstract counterexamples allows us to prune during symbolic execution. If we find an assertion violation with n abstracted functions, we can drop any state- including states which have not finished execution- in which we abstracted $n+1$ or more functions.

Search Deepening The reductions rules in § 3.3 implicity create a (often infinite) tree of states. The order we search the branches of this tree, and how deep we search, is an important consideration to find counterexamples efficiently.

We search in a depth first manner, to some maximal depth. If we have explored all branches, and not found a counterexample, we increase the maximal depth and continue searching. Every time we find a counterexample that is better (has less abstracted functions) than our current best counterexample, but that is deeper in the tree, we also increase the maximal depth.

Thus, this strategy allows searching to continue if better counterexamples are being found by searching deeper in the tree. However, we avoid fruitlessly searching too many states, if they are not producing promising results.

The gradual increase in the maximal depth ensures we are evaluating a variety of states and branches, preventing us from spending too much time on branches that will not yield a counterexample. It often enables us to find a close-to-minimal counterexample fairly quickly, allowing us to prune all states with a greater number of abstracted 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 counterexamples to refinement types, as shown in § 2.4 and § 2.5.

Refinement Types We support the language of refinement types shown in Figure 9. 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 τ_1 to τ_2. The x is a *outer bound name* to refer to the value of τ_1, allowing it to be referenced in refinements in τ_2.

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

τ	::=		**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**

Figure 9. λ_D types

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

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

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

Figure 10. λ_D precondition and postcondition

Pre and Post Figure 10 shows pre and post. pre walks over the function and drops the return type. post keeps the argument bindings, but sets all refinements, except the return type's refinement, to `True`. Keeping the bindings 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. toExp, shown in Figure 11, translates LiquidHaskell refinement types into predicates in λ_G. This function has many subparts:

- toExp_λ creates lambda bindings, giving us names to refer to both the inputs and outputs of the funciton.
- toExp_b and toExp_r translate each individual refinement on a type into a λ_G *predicate* on a value.
- toExp_τ walks over the spine of a LiquidHaskell function type, to apply toExp_b to each argument.

Polymorphic Data Types LiquidHaskell allows checking refinements on polymorphic type variables. For example, we may refine a polymorphic list `[a]` 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 λ_G. To do this for a type constructor τ, with type variable a, a higher order function p_τ 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_τ to a predicate expression and an expression of type τ, to assume or assert that those predicate expressions hold on all type variables in p. For example, on a list, we have:

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

and we translate $[\{x : \texttt{Int} \mid 0 < x\}]$ to $p_{\text{List}}\ (\lambda x\ .\ 0 < x)$.

6 Implementation and Evaluation

We next describe the 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 lazy symbolic execution for the Haskell language in a tool named G2. It is open source, and available at https://github.com/BillHallahan/G2. We use the GHC API to parse Haskell programs, and Z3 [8] and CVC4 [1] as SMT solving backends. G2 supports a large Haskell98-like subset of the code compiled by GHC, which also includes features not detailed in § 3.1 such as polymorphism. G2 uses a custom version of Haskell's Base library and Prelude [26]. For a range of modules, functions, and datatypes, G2 can use this custom standard library to symbolically execute programs written with the standard Base and Prelude.

6.2 Quantitative Evaluation

The goal of our evaluation is twofold. ***Q1*** Does symbolic execution find counterexamples that explain refinement type errors? ***Q2*** Do the abstract counterexamples accurately pinpoint the functions whose specifications are too weak to permit type checking?

Our empirical evaluation answers these questions positively. We use G2 to generate counterexamples for refinement type errors on a corpus of programs written by students using LiquidHaskell for a homework assignment in CSE 230, a graduate level programming languages class, at the University of California, San Diego (IRB #140608). The assignment contained a variety of exercises. For some, the students had to write code that implemented a function, and matched a given refinement type. For others, the students were asked to write refinement types for prewritten functions. In total,

$$\text{toExp}(\tau) = \text{toExp}_\lambda(\tau, \tau)$$

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

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

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

$$\text{toExp}_r(r) = \begin{cases} \text{toExp}_r(r_1) == \text{toExp}_r(r_2) & r = r_1 == r_2 \\ \text{toExp}_r(r_1) < \text{toExp}_r(r_2) & r = r_1 < r_2 \\ \text{toExp}_r(r_1) \wedge \text{toExp}_r(r_2) & r = r_1 \wedge r_2 \\ \ldots & \ldots \end{cases}$$

Figure 11. λ_D to λ_G translation

each students assignment was roughly 150 to 200 lines of code.

Corpus The corpus contains, in total, 10,349 incorrect refinement types. The data was collected by logging the student's work every time a student typechecked their code with LiquidHaskell. 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.

Preprocessing The corpus was collected from a class run in 2015. LiquidHaskell's syntax has changed since then, rendering some of the files non-parsable. Altogether, on the student written data set G2 can be applied to 93.6% of the files. From those, we excluded 2136 functions because they were only stubs, which immediately called error. Finding counterexamples for these functions is trivial, because *any* input would be a counterexample. This left us with a total of 7550 functions to evaluate G2 on.

Search Strategy Our search deepening strategy (§ 4.1) takes two parameters: an amount s to increase the search depth, if no counterexample is found, and an amount c to increase the search depth, when a better counterexample is found. Based on our experience with G2 we selected $s = 300$ and $c = 500$ as values that appeared to give reasonable results. G2 was given a maximum of 2 minutes to find counterexamples for each function.

Results Figure 12 summarizes the results of our evaluation on the 7550 functions, drawn from actual code written by students. It demonstrates that G2 finds counterexamples for the vast majority of the LiquidHaskell errors. In total, we found counterexamples for 7379, or 97.7%, of the errors. We found concrete counterexamples for 4354, or 57.6%, of the errors, and found abstract counterexamples for 3025, or 40.1%, of the errors. While G2 has an average runtime of only 17.6 seconds, the median running time is even lower – 7.9 seconds. This shows that G2 is a practical and efficient tool to help debug LiquidHaskell refinement type errors, giving a very positive answer to ***Q1***.

G2 failed to find a counterexample only 2.3% of the time. 1.5% of our failures come from timeouts, while the remaining 0.7% is accounted for by errors in G2, which mostly relate to unimplemented edge cases in LiquidHaskell specifications.

Correctness of Abstract Counterexamples Our benchmarks come from traces of programmers iteratively invoking LiquidHaskell to verify some properties. Thus, we determine whether G2's abstract counterexamples *correctly localize* the imprecise specification by comparing each "bad" file – that was *rejected* by LiquidHaskell, for which G2 found an abstract counterexample – with the first "fixed" file along the user's trace that was *accepted* by LiquidHaskell. We say that an abstract counterexample *correctly localizes the error* if the counterexample blames a call to some function f such that in the "fixed" version (a) the user *specifies* a different type for f, or (b) the user *replaces* f with a different function with a stronger type, or (c) LiquidHaskell *infers* a different type for f *e.g.* because it is used differently in the code. We say an abstract counterexample is *spurious* otherwise.

Evaluating Correctness Of the 3025 counterexamples, after discarding 1041 "bad" files that had no "fixed" version (as some students did not finish the assignments) we were left with 1984 abstract counterexamples. We categorized these counterexamples via a combination of scripts and manual inspection as one of (a), (b), (c) or spurious. We find that in 1747 (88.1%) cases the user ends up specifying a different type (a), in 9 (0.4%) cases the user ends up replacing the function (b), and in 151 (7.6%) cases the user ends up changing other code to allow LiquidHaskell to infer the right type needed

Function	Con.	Abs.	Time out	Error	Avg. Time (s)
prop_map	13	523	2	2	8.9
foldr1	607	0	0	2	10.1
kmeans1	0	875	22	0	120.5
prop_concat	877	217	0	5	5.7
replicate	8	7	2	2	17.9
mergeCluster	329	2	0	0	7.2
collapse	250	0	0	5	8.3
prop_zipWith	18	708	13	2	10.5
concat	181	80	53	8	58.3
nearest	80	0	0	6	10.5
prop_replicate	335	125	0	2	5.7
expand	49	74	0	0	8.3
length	16	0	7	0	40.8
zipWith	760	3	4	2	9.2
kmeans	0	18	2	0	120.5
centroid	594	0	0	0	6.4
prop_size	0	224	2	3	6.7
mapReduce	93	137	8	8	12.2
add	3	0	0	2	4.4
concat2	5	1	0	0	10.0
prop_concat2	36	2	0	0	5.9
distance	84	3	0	0	9.5
prop_concat_1	0	6	0	0	6.0
prop_join	0	6	0	0	5.6
Other	16	14	0	7	14.5
Total	**4354**	**3025**	**115**	**56**	**17.6**

Figure 12. Evaluation results for errors reported by LiquidHaskell on student homeworks. Con. is the number of reported concrete counterexamples. Abs. is the number of abstract counterexamples reported by G2. Timeout is the number of times G2 timed out before returning counterexamples. Error is the errors encountered in G2 when generating counterexamples. Avg. Time is the average amount of time taken by all runs of G2 reported in the table.

for verification (c). Thus, we conclude that in 96.1% of the cases, G2's abstract counterexamples correctly identified the function whose specification was too weak.

Replicate Function One particularly interesting abstract counterexample stood out to us. This counterexample was actually counted as spurious, as it does not fit any of our classifiers for correct abstract counterexamples, but nonetheless shows something interesting about the code. Consider:

```
replicate :: n:Int -> a ->
  { xs:[a] | size xs == n }
replicate 0 x = []
replicate n x = x:replicate n x
```

`replicate` is supposed to return a list of the given length, but due to a mistake in the implementation (the counter is never decreased) instead returns an infinite list. However, classical symbolic execution would fail to find a concrete counterexample, because the computation of `size xs == n` would never terminate. However, G2 finds an abstract counterexample for `replicate`:

```
    replicate 1 0 = [0, 0]
  violating its refinement type, if
    replicate 1 0 = [0]
```

If the first recursive call to `replicate 1 0` returns `[0]`, the the outer call to `replicate 1 0` returns `[0, 0]`, violating the refinement type.

Our primary motivation to develop abstract counterexamples was to aid in cases where the specification was insufficient. Therefore, it was a surprising discovery that it can also provide output in cases of non-termination.

7 Related Work

Verification Techniques and Debugging An IDE for Dafny that helps debug genuine and spurious failed verification conditions is described in [6]. Like our work, it uses a symbolic execution based approach to find concrete counterexamples. However, for spurious errors, it simply displays the SMT model, which, unlike abstract counterexamples, does not pinpoint any specific function whose specification needs strengthening for verification to succeed.

CORRAL [17–19] is a reachability solver based on generating verification conditions. CORRAL introduces the stratified inlining technique, which inlines functions on demand if verification fails when just using the function contracts. As opposed to counterfactual counterexamples, stratified inlining aims to improve the underlying verification, rather than improve explainability of verification errors. As such, stratified inlining can be seen as an orthogonal technique to G2's counterfactual counterexamples. Stratified inlining aims to minimize the number of inlined functions, whereas counterfactual counterxample generation aims to minimize the amount of abstraction.

Haskell Libraries, Program Analysis, and Testing Catch [22] and Reach [23] are static analyses for Haskell that look for specific kinds of errors as opposed to our general symbolic execution. QuickCheck [7] and SmallCheck [30] and Target [31] test properties by running Haskell code on large numbers of random or SMT-generated, concrete inputs. While either could potentially be used to generate concrete counterexamples from LiquidHaskell types, neither produces abstract (counterfactual) counterexamples.

Haskell Verification Xu's work on static contract checking [43, 44], relies on a symbolic simplifier, parts of which resemble our reduction rules (§ 3.3.) Similarly, Halo [41] and LiquidHaskell [40] aim to verify properties of Haskell

programs. However, in contrast to G2, these tools aim for verification, as opposed to refutation which is the goal of our lazy reduction-based symbolic execution. None of them produce abstract counterexamples when verification fails.

Solver-aided Programming Solver-aided programming is a paradigm that makes it easier to write code that uses a constraint solver. As opposed to G2, which focuses on finding assertion violations, solver-aided programming allows directly manipulating symbolic values in code.

ROSETTE [36, 37] is a general purpose framework that enables solver-aided programming in Racket. This makes it easy to use Racket to formulate search problems over a symbolic domain. Programs can be written with traditional Racket code, but ROSETTE introduces symbolic integer and Boolean values. Unlike G2, ROSETTE does not attempt to find assertion violations in code. Rather, it gives programmers a higher level interface to constraint solvers, simplifying the writing of tools that manipulate symbolic values.

SmtEn [38] is a plugin for GHC that allows for high level constraint solving. A user can call SmtEn's API to manipulate symbolic values. Similarly to ROSETTE, SmtEn gives a higher level interface to the capabilities of constraint solvers. It is therefore better suited than G2 for programmers who want to make use of constraint solving in their programs. However, SmtEn does not symbolically execute general purpose Haskell code, which makes G2 more usable as an off-the-shelf debugging aid.

Symbolic Execution for Functional Languages CutEr [10, 11] is a symbolic execution engine for Erlang programs. SCV [24, 35] is a static contract verifier for Racket based on symbolic execution. Racket and Erlang are strict languages, and thus neither of the above tools considers lazy evaluation, which requires a different approach as demonstrated in (§ 2). Further, to our knowledge, neither of the above scales to check inductive properties (*e.g.* size, height) of recursive datatypes (*e.g.* lists, trees). Finally, none of them produces abstract counterexamples to pinpoint weak specifications.

Symbolic Execution for Imperative Languages There are many symbolic execution engines for imperative languages, including Dart [12] and Cute [32] for C, Symbolic Pathfinder [25] for Java, Pex [34] for .NET, Sage [13] for x86 Windows applications, and EXE [4] and its sequel Klee [3] for LLVM. The execution semantics of imperative programs are quite different from ours, but other techniques (such as path search strategies) are likely to be applicable to Haskell symbolic execution.

8 Conclusion

We presented counterfactual symbolic execution for non-strict languages, and used it to find counterexamples that illustrate concretely or abstractly why a modular checker fails to verify a program. Our evaluation on a large corpus of 7550 verification errors from users of LiquidHaskell demonstrates that we can find counterexamples to **97.7%** of errors. For **57.6%** of the errors we find concrete counterexamples, and for an additional **40.1%** of the errors we find abstract counterexamples, which **96.1%** of the time correctly pinpoint the imprecision that precludes verification. Thus, our results show that by generalizing the notion of counterexamples via counterfactual execution, we can quickly, automatically, and accurately guide the puzzled developer to the part of their code or specification that they need to fix.

Acknowledgments

We thank the anonymous referees and our shepherd Michael Greenberg for their feedback on earlier versions of this paper. This work was supported by the the National Science Foundation under Grant Numbers CCF-1553168, CCF-1302230, CCF-1422471, CCF-1223850, CCF-1218344, CCF-1763814, and a generous gift from Microsoft Research.

References

[1] 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.

[2] 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

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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

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

[5] 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

[6] Maria Christakis, K Rustan M Leino, Peter Müller, and Valentin Wüstholz. 2016. Integrated environment for diagnosing verification errors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 424–441.

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

[8] 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

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

[10] 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

[11] Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2017. Concolic testing for functional languages. *Science of Computer Programming* 147 (2017), 109–134.

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

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

[14] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 553–568.

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

[16] 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

[17] Akash Lal and Shaz Qadeer. 2014. A program transformation for faster goal-directed search. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, 147–154.

[18] Akash Lal and Shaz Qadeer. 2015. DAG inlining: a decision procedure for reachability-modulo-theories in hierarchical programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 280–290.

[19] Akash Lal, Shaz Qadeer, and Shuvendu K Lahiri. 2012. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification*. Springer, 427–443.

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

[21] Simon Marlow and Simon Peyton Jones. 2004. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 4–15.

[22] 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

[23] 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

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

[25] 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.

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

[27] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, Vol. 93.

[28] Simon L Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of functional programming* 2, 2 (1992), 127–202.

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

[30] 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.

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

[32] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[33] 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*.

[34] 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

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

[36] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 135–152.

[37] 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

[38] 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

[39] Niki Vazou, Eric L Seidel, and Ranjit Jhala. 2014. Liquidhaskell: Experience with refinement types in the real world. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 39–51.

[40] 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.

[41] 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

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

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

[44] Dana N Xu, Simon Peyton Jones, and Koen Claessen. 2009. *Static contract checking for Haskell*. Vol. 44. ACM.