Explaining Verification Errors via Faceted Symbolic Execution

THE AUTHORS AND THEIR AFFILIATIONS ARE OMITTED FOR ANONYMITY

Modular verifiers allow programmers to specify correctness properties of their code using function contracts or pre- and post- conditions or refinement types and to automatically verify their implementations using SMT solvers. Unfortunately, when verification *fails*, the hapless programmer is given no feedback about *why* their code was rejected (let alone *how* they can fix it.) We introduce Faceted Symbolic Execution, a new approach that produces *counterexample executions* that demonstrate why modular static verification fails. First, we formalize a core symbolic execution framework for the Haskell. Second, we show how to find counterexamples for refinement type errors by *reducing* refinement types into assumes (for input types), asserts (for output types) and (recursive) traversals over inductive data types. Third, we formalize the notion of Faceted Symbolic Execution by showing how to speculatively trigger *multiple* symbolic searches for each function application, to localize whether the *code* or the *specification* of the function is to blame. Finally, we implement and evaluate our approach on a large corpus of 143 errors gathered from users of the LiquidHaskell refinement type system. We show that Faceted Symbolic Execution is able to quickly find counterexamples for 81% of the errors, thereby providing effective explanations of why verification fails.

ACM Reference Format:
the authors and their affiliations are omitted for anonymity. 2018. Explaining Verification Errors via Faceted Symbolic Execution. 1, 1 (March 2018), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modular verifiers allow programmers to specify correctness properties of their code using function contracts or 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]). These verifiers can compose the specifications and code to obtain logical *verification conditions*(VCs) that can be automatically checked using SMT solvers, to verify, at compile-time, that the implementations meet critical, programmer specified safety requirements ranging from safe array accesses[Kent et al. 2016], to data structure invariants[Kawaguchi et al. 2009], to the correctness of distributed systems[Hawblitzel et al. 2015] and fully abstract compilation[Fournet et al. 2013]. Unfortunately, modular verifiers have remained within the ivory tower, as they are extremely 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.)

While modularity has many virtues, it greatly complicates the task of explaining why verification fails. To see why, let us consider the different ways in which verification can fail when checking if a function f satisfies a contract given by a *pre-condition* P and a *post-condition* Q. (1) First, the code may not satisfy the contract. The precondition P may be too *weak*: maybe the postcondition only holds on a smaller set of inputs than those described by the precondition, or, the postcondition Q may be too *strong*: maybe the function's code is incorrect and establishes a weaker property than

Author's address: the authors and their affiliations are omitted for anonymity.

XXXX-XXXX/2018/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

stipulated by the postcondition. (2) Second, most perniciously, the code of f may in reality satisfy the contract, but verification may still fail as the contracts for one or more functions called within the body of f are not *strong enough* to yield a valid VC. This could be, e.g. because an invariant is not inductive, or because the post-condition for some called function g does not capture enough information about the values returned by that function.

We introduce Faceted Symbolic Execution, a new approach that produces *counterexample executions* that demonstrate why modular static verification fails. In case (1) there is typically a short execution trace that witnesses the failure. That is, via symbolic execution, we can find a set of inputs that satisfy the precondition but which produce an output which fails the postcondition. Crucially, Faceted Symbolic Execution addresses case (2) via the insight that we can determine whether there is some g whose specification is *too weak* by comparing the result of *two* symbolic executions: one where we use the *concrete implementation* of g, and one where we use the *modular specification*. If the former fails to produce a counterexample and the latter does, then in fact, the specification of g is too weak and we can precisely pinpoint the kinds of spurious behaviors that must be removed by a strengthened modular specification.

In this paper, we develop, implement and evaluate Faceted Symbolic Execution via the following concrete contributions.

- First, we formalize a core symbolic execution framework for *Haskell*. As is standard in symbolic execution, in our framework, contracts are expressed simply via *assume* and *assert* terms. However, Haskell's semantics is fundamentally different from other (strict, call-by-value) languages that symbolic execution has been developed for. In this paper, we show how to account for features such as partial function application on functional terms, lazy graph reduction based semantics, algebraic data types and pattern matching as means of path branching, to obtain the first symbolic execution framework for Haskell, which can automatically synthesize inputs and executions that trigger particular behaviors on programs represented as GHC Core (§ 4).
- Second, we show how to find counterexamples for refinement type errors by *reducing* refinement types into assumes (for input types), asserts (for output types) and (recursive) traversals over inductive data types. Our translation shows for the first time how to synthesize inputs that yield executions that concretely witness how refinement subtyping fails (§ 5.1)
- Third, we formalize the notion of Faceted Symbolic Execution by showing how to speculatively trigger two symbolic searches for each function application: one using the callee's (concrete) implementation, and the other using the callee's (abstract) specification. If a counterexample is found by the concrete search, then we can directly report that as an execution that violates the contract being checked. If instead, a counterexample is found with the abstract search (but not the concrete one), then we know that the abstract specification for the callee is to *blame* (§ 5.2).
- Symbolic execution is heuristic procedure that may fail to find a counterexample with either the concrete or abstract searches. Our final contribution is an implementation and empirical evaluation of our approach over a large dataset of 143 errors, from 115 files, gathered from users of the LiquidHaskell refinement type system [Vazou et al. 2014]. We show that Faceted Symbolic Execution is able to quickly – in roughly 5 to 15 seconds per error – find concrete counterexamples for 86 of the errors. Furthermore, Faceted Symbolic Execution is able to find abstract counterexamples that pinpoint the library function whose specification is too weak for 30 of the errors. In all, Faceted Symbolic Execution is able to synthesize inputs and executions that explain 81% of verification failures, and hence, could point the way towards making modular verification accessible.

2 BACKGROUND

In the next section, we present symbolic execution for type error localization in function pre and post-conditions, specified via LiquidHaskell. For new users in particular, error messages from LiquidHaskell may be enigmatic, as they often involve first order logic over complex type structures. To assist these users, we demonstrate G2 as a means to generate counterexamples that violate function specifications. With illustrative counterexamples, we can help users better address edge cases they may not have thought of otherwise.

LiquidHaskell [Vazou et al. 2014] is a Haskell program verification framework that augments Haskell's type system through *refinement types*. A refinement type endows Haskell types with additional constraints through predicates in decidable first-order logics. We illustrate some basic functionality of LiquidHaskell through examples. Consider a type of the natural numbers `Nat`, which denotes non-negative integers. To express this, one writes:

```
type Nat = {v:Int | v >= 0}
```

In LiquidHaskell, refinement type annotations appear where regular type declarations in Haskell would, wrapped in curly braces. For example, a refinement type for the `abs` function, which ensures that it only returns `Nat`'s, can be written as follows:

```
{-@ abs :: Int -> Nat @-}
abs n = if 0 <= n then n else 0 - n
```

A restricted class of Haskell like functions may be lifted into the refinement types through the `measure` annotation. They are used to specify properties of algebraic data structures. For example, the following measure returns the size of a custom list:

```
data List a = Emp
            | (:+:) a (list a)
            deriving (Eq, Ord, Show, Generic)

{-@ measure 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:[Int] | size v > 0} -> Int @-}
head (x:_) = x
head _     = die "head: called with empty list!"

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

The second pattern match of `head` is never reachable due to the refinement of the input parameter requiring a non-empty list. The unreachability is specified by providing the refinement type on `die` as false. In practice that means that LiquidHaskell should never verify code that can call `die`, since false is unsatisfiable. Therefore, LiquidHaskell will only type check `head` if `die` cannot be called from within `head`'s body.

3 MOTIVATING EXAMPLES

3.1 Symbolic Execution

As a symbolic execution engine, G2 allows running Haskell functions with symbolic values in place of concrete values. One of the possible scenarios is running G2 on a Haskell program that does not contain any program annotations. A symbolic execution engine should be able to generate function inputs that explore the paths through program, or that satisfy specific constraints. To illustrate this basic functionality of G2, consider the following Haskell functions, drawn from the Haskell Prelude (the Haskell standard library) [Jones 2003]:

```
intersect :: (Eq a) => [a] -> [a] -> [a]
intersect = intersectBy (==)

intersectBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy _  [] _  = []
intersectBy _  _  [] = []
intersectBy eq xs ys =  [x | x <- xs, any (eq x) ys]

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

`intersect` returns the intersections of two lists, by calling the more general `intersectBy`. `intersectBy`, in turn, calls `any`. By analyzing the specification, we realize that `intersect` is a bit different that standard intersection. A specification for `intersectBy` (available in List.hs) states "If the first list contains duplicates, so will the result."

In practice, programs rely on several calls to different functions, so it might be harder to understand their behavior without any annotations. If provided examples, representing well the behavior of the program, the user can better understand what the program is about, without manually analyzing underlying calls to other functions. By running G2 on `intersect`, we output a sequence of representative examples that illustrate the function's behavior:

```
intersect ([])        ([])     = []
intersect ([0])       ([])     = []
intersect ([1])       ([0])    = []
...
intersect ([0, 0, 1]) ([0, 1]) = [0, 0, 1]
intersect ([0, 0, 0]) ([0])    = [0, 0, 0]
```

In addition to generating examples, a symbolic execution engines should also be able to verify certain properties of the given code. If the user would like to check if `intersect` is commutative, they first need to define a predicate describing that property:

```
prop_commutative :: (Eq a) => [a] -> [a] -> Bool
prop_commutative xs ys = xs `intersect` ys == ys `intersect` xs
```

We can check this by running G2 on `prop_commutative` with the flag `--returns-true`. This flag tell G2 that you expect the proposition to hold, and asks it to report any counterexamples. In fact, `prop_commutative` is not true, and G2 will report several counterexamples, including:

```
prop_commutative ([0, 1]) ([1, 1, 0]) = False
```

Indeed, it is the case that `intersect ([0, 1]) ([1, 1, 0])` returns `[0,1]`, while `intersect ([1, 1, 0])([0, 1])` returns `[1, 1, 0]`.

Notably, G2 does not have to execute `intersect` on arbitrary lists to find this counterexample. Rather, it internally represents the lists as formulas, and then tries to solve the formulas to violate the proposition.

3.2 Refinement Type Error Localization

One of the main features of the G2 symbolic execution engine is automated generation of concrete and symbolic counterexamples. As a use case for empirically testing this feature, we use programs written by the students of a course taught at **omitted for anonymity** during **anonymity**. The course taught students to better understand the concepts of pre- and post-conditions, and as a part of the course, students were asked to write LiquidHaskell type annotations and verify a given program. Whenever students submitted their code to LiquidHaskell for verification, the outputs results and files were logged.

In this section, we execute G2 on a couple of those programs. Throughout the examples we use the same `List` datatype and `size` measure as in Section 2.

All these programs were written and annotated by real world users. If their programs do not verify, having a counterexample can help them either to better understand how to improve their specification, or to identify an error in their code. One of the envisioned applications of the G2 faceted symbolic execution is using it for educational purposes, when built on the top of a verification tool.

3.2.1 Finding the correct refinement type for zipWith. Students were considering the `zipWith` function defined below. 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 students were asked to verify the function by providing type refinements describing the preconditions of the function.

One student submitted the following incorrect specification for verification:

```
{-@ zipWith :: (a -> b -> c) -> v1 : List a
            -> {v2 : List b | size v1 > 0 => size v2 > 0} -> List c @-}
zipWith _ Emp         Emp         = Emp
zipWith _ (x :+: xs) (y :+: ys) = f x y :+: zipWith f xs ys
zipWith f _           _           = die "Bad call to zipWith"
```

LiquidHaskell fails to verify this type, with multiple errors, one of which we replicate below:

```
 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}
   In Context
     xs : (List a)
     ys : (List a)
```

To a beginner, 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 G2 on `zipWith` produces and outputs the following *concrete counterexample*:

```
zipWith asTypeOf Emp (0 :+: Emp) = error
makes a call to
die ("Bad call to zipWith") = error
violating die's refinement type
```

The counterexample illustrates which inputs (the empty list and the list containing only 0 as an element) will cause `zipWith` to invoke the `die` function, making LiquidHaskell fail to type check. The counterexample should be understood as follows: making a call to `zipWith` with the function `asTypeOf` and two `List`s of unequal size causes `zipWith` to yield an `error`. With this information in hand, the student can easily see how to improve the refinement type.

3.2.2 Introducing the abstract counterexamples for the `concat` function. In another project the students were asked to write a function definition for `concat`, then write a refinement type to prove the returned lists length was the sum of the input's sublist lengths. One student approached the problem by writing:

```
{-@ measure 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 :: List (List a) -> List a
concat Emp                    = Emp
concat (xs :+: Emp)           = xs
concat (xs :+: (ys :+: xss)) = concat ((concat' xs ys) :+: xss)

{-@ concat' :: List a -> List a -> List a @-}
concat' :: List a -> List a -> List a
concat' Emp        Emp = Emp
concat' xs         Emp = xs
concat' Emp        ys  = ys
concat' (x :+: xs) ys  = x :+: concat' xs ys
```

In fact, this `concat` function is correct, but LiquidHaskell cannot verify it. This is because at function call sites, LiquidHaskell is only aware of the information that is in the refinement type of the called function, and does not examine the definition of called functions. Thus, when trying to verify `concat`, LiquidHaskell knows nothing about the data values returned by `concat'`. While it is clear to us that the size of a list returned by `concat'` is the sum of the sizes of the input lists, LiquidHaskell does not provide mechanisms for concluding that. Running a symbolic execution engine can help to pinpoint functions that need stronger refinement types.

In the case above, we would still like to provide to users some sort of helpful counterexamples. Clearly, it is impossible to provide a concrete counterexample, because the refinement type of the function is correct. Instead we introduce the concept of *abstract counterexamples*. In an abstract counterexample, we derive and provide alternative partial definitions for called functions (for example, `concat'`). These new definitions still obey the functions' refinement types, but they lead to a violation

of the refinement type of the calling function's (`concat`, in the above example). More precisely, running G2 on `concat` reports the following abstract counterexamples:

```
concat (Emp :+: (Emp :+: Emp)) = 0 :+: (1 :+: Emp)
makes a call to
concat ((concat' Emp Emp) :+: Emp) = 0 :+: (1 :+: Emp)
violating concat's refinement type
when
concat' Emp Emp = 0 :+: (1 :+: Emp)
Strengthen the refinement type of concat' to eliminate this possibility
```

To explain in more detail, when invoking `concat`, due to insufficient type refinements, LiquidHaskell cannot prove that the expression `concat' Emp Emp` does not return `0 :+: (1 :+: Emp)`. If we assume that it was true, and indeed this call to `concat'` yields such values, this would, actually, violate the refinement types of `concat`. Inspired by this counterexample, a student may improve the type refinement on `concat'` to:

```
{-@ concat' :: x : List a  -> y : List a
            -> {z : List a | size x + size y = size z} @-}
```

With this strengthened refinement type, LiquidHaskell can now verify `concat`.

We call G2 *Faceted Symbolic Execution*, for its ability to apply and combine modular reasoning in the symbolic execution engine. As illustrated, this faceted symbolic execution is particularly important for generating counterexamples.

4 THE G2 SYMBOLIC EXECUTION ENGINE

In this section, we describe the principles on which is G2 based. Our tool translates a Haskell program into an internal representation. We first define the internal language of G2 and document the modifications employed to facilitate the symbolic execution process. Then, using this language, we present evaluation semantics for G2, which augments those of Haskell's semantics in order to account for symbolic expressions.

4.1 Internal Language Representation for G2

The Glasgow Haskell Compiler (GHC) [Jones et al. 1993] compiles Haskell by reducing Haskell source code to an intermediate lambda calculus-like representation, known as Core Haskell. Core Haskell is adapted from System $F^{\uparrow}_{C}$, and has proven valuable for code optimizations and transformation in GHC [Jones 1996].

In G2, we opt to translate Core Haskell into our own Core language. This language is similar to the existing Core language, but features some changes and additions to aid in symbolic execution. In particular, we add *assume* and *assert* statements, which can be used to represent LiquidHaskell predicates. The grammar of this language is given in Figure 1.

Haskell programs are represented by expressions in the G2 language. Every expression in G2 has a type. We write $e : \tau$ to denote that e has type τ. Type deduction rules in G2 resemble those used in System $F^{\uparrow}_{C}$ [Jones 1996], and so we omit them.

We now provide some intuition to the meaning of each expression. In Sec. 4.2, we will give a formal description of execution semantics.

Variables		:=	$x : \tau,\ \ f : \tau,\ \ s : \tau$	
Literals	l	:=	i	int, double, char, string
Expressions	e	:=	$x \mid l$	
			$\oplus$	primitive operation
			D	data constructor
			$e_1\ e_2$	function application
			$\lambda x\ .\ e$	lambda abstraction
			`let` $x = e$ `in` e	let binding
			`case` e `as` x `of` $\{alt_1, \ldots, alt_n\}$	case expression
			`type` τ	type expression
			`assume` $e_1\ e_2$	assumption of e_1 about e_2
			`assert` $b\ e_1\ e_2$	assertion of e_1 about e_2
			`cast` $e : \tau_1 \sim \tau_2$	type cast τ_1 to τ_2
Alternatives	alt	:=	$D\ \ x_1 \ldots x_n \rightarrow e$	data constructor alternative
			$l \rightarrow e$	literal alternative
			`default` $\rightarrow$ e	default alternative
Type Constructors	T	:=	$T\ \ \tau_1 \ldots \tau_m\ \ D_1 \ldots D_n$	
Data Constructors	D	:=	$K\ \ \tau_1 \ldots \tau_m\ \ \tau_1 \ldots \tau_n$	
Tracker	b	:=	$(x, \overrightarrow{x}, x) \mid \emptyset$	
Types	τ	:=	α	type variable
			`Int` $\mid$ `Double` $\mid$ `Char` $\mid \ldots$	literal types
			$\tau_1 \rightarrow \tau_2$	type application
			$T\ \tau_1 \ldots \tau_n$	type constructor application
			$\forall \alpha\ .\ \tau$	for-all type
			`TYPE`	polymorphic type
			$\bot$	bottom type

Fig. 1. A grammar of the languge for internal representation of Haskell programs in G2

Variables. Variables represent values, and can be either *concrete* or *symbolic*. A concrete variable has a fixed value, whereas a symbolic variable does not, and depends on the path constraint formulae in the execution state (cf. Sec 4.2). Variables are also endowed with a type.

Literals. Literals represent primitive literal constants that cannot be defined in Haskell. We note that datatypes like **`Int`**, **`Double`**, and **`Char`** are algebraic datatypes that wrap these primitive literal constants, which have types that end in a hash `#`, such as **`Int#`**, **`Double#`**, and **`Char#`**. Furthermore, although `Bool` is an algebraic datatype in the Haskell Prelude, we treat it as a primitive literal constant for simplified representation and constant folding.

Primitive Operators. Primitive operators are functions over primitive literals that also cannot be defined in Haskell. We special case such functions during translation from Core Haskell.

Data Constructor. A data constructor is used to build a value for an algebraic datatype. For example, the constructors of a list are `(:)` and `[]`. Data constructors may be thought of as functions that output a value of their corresponding algebraic datatype.

Function Application. Function application applies an argument to a function. All functions in Haskell, and likewise G2, are curried, meaning that they take a single argument. As is in standard

lambda calculus notation, function application is left-binding, so we interpret the expression $f\ x\ y$ to mean y applied to the result of $f\ x$.

Lambda Abstraction. We write $\lambda x\ .\ e$ to denote a lambda abstraction of x over the body expression e. To use function application as an example, the expression $\lambda x\ .\ e_1\ e_2$ then denotes the application of e_2 to the lambda abstraction, binding e_2 to x in the body of e_1. Every time binding occurs, we use fresh variable names to avoid naming conflicts.

Let Binding. An expression of form `let` $x = e_2$ `in` e_1 is understood as binding e_2 to x in the body of e_1. We treat recursive let bindings and non-recursive let bindings the same way because all variables receive fresh names upon binding. This resolves scoping issues that may occur otherwise.

Case. Case expressions of form `case` e `as` x `of` $alt_1, \ldots, alt_n$ allow for conditional branching. The *scrutinee* expression e is first evaluated to *Symbolic Weak head Normal Form* (SWHNF, cf Sec 4.2, Sec 1). We then bind the resulting SWHNF to the *case binder* variable x, which may be used to refer to the value of the scrutinee e in the body of any of the *alt* branches that possible alternatives of execution. Following this, we then select one *alt* in the list $alt_1, \ldots, alt_n$ to branch on based on pattern matching. There are three different types of *alt* pattern matching available:

(1) Data *alt* matches are for constructors, where constructor arguments are bound to *alt* parameters.
(2) Literal *alt* matches match against literals.
(3) The `default` alternative is matched when all other *alt* branches fail to match.

The list of alternatives is always exhaustive: a `default` alternative is added if one is not already written, and will raises an *non-exhaustive pattern match* error upon execution.

Type. A type expression `type` τ allows a type to be used as expressions. Type expressions are artifacts of Core Haskell, and appear only as arguments to polymorphic functions in order to keep track of type specialization.

Cast. Cast expressions and type coercions are used to record type equalities and type level operations. GHC uses such expressions to support features such as generalized algebraic datatypes, kind polymorphism, and newtype declarations [Yorgey et al. 2012]. Cast expressions may be eliminated during program execution without affecting type safety. However, G2 retains these expressions in order to output well-typed expressions according to type cast annotations. We currently support a subset of cast expression features, including the ability to encode newtype declarations. Casts may never be applied directly to a primitive literal.

Assume and Assert. The `assume` $e_1\ e_2$ and `assert` $e_1\ e_2$ expressions denote logical predicates that e_1 states about e_2. In addition, the assertion expression has a *tracker* variable b, which watches for assertion violation. In G2, each assertion is always associated with some specific function. We set the tracker to watch $(f, a_1, \ldots, a_n, r)$, where f is the name of the associated function with formal parameters $a_1, \ldots, a_n$, and some return value r. This allows us to track and localize the source of assertion violations.

4.2 The Execution Model for G2

We describe a symbolic execution as a sequence of rewrite steps. We introduce the concept of the *execution state*, which is a data structure that captures a snapshot of program properties such as environment mappings, stack frames, and path constraints.

State	S	:=	$(\mathcal{E}, \mathcal{C}, \mathcal{PC}, Stk, b)$	
Path Constraints	$\mathcal{PC}$	:=	$\bigwedge_i pred_i$	
Predicate	$pred$	:=	$D\ x_1 \ldots x_n = x$	Constructor Binding
			$l = e$	Literal Binding
			`isCons` $D\ x$	Constructor Presence
			e : Bool	
			$\neg pred$	
Current Expression	$\mathcal{C}$	:=	e	
Frames	Fr	:=	`CaseFrame` x `of` $\{alt_1 \ldots alt_n\}$	
			`ApplyFrame` e	
			`UpdateFrame` x	
			`CastFrame` $\tau_1 \sim \tau_2$	
			`AssumeFrame` e	
			`AssertFrame` $b\ e$	

Fig. 2. Definition of a State, used by G2 as a snapshot of program execution.

A pure functional program can be seen as a graph with some *current expression* $\mathcal{C}$ of interest, such as the main function [Jones 1992]. In this graph, expressions form the vertices, while each variable within an expression draws a directed edge towards another expression, which may be a functional term like a lambda abstraction. This graph is implicitly encoded in the expression environment $\mathcal{E}$, which is a table that maps variables to expressions.

We denote the binding (insertion) of a variable x to an expression e in the expression environment $\mathcal{E}$ by writing $\mathcal{E}\{x = e\}$. Likewise, we define the $\texttt{lookup}(\mathcal{E}, x)$ function to return the appropriate expression for which x is bound to in $\mathcal{E}$, and error if not found. In addition, we define a $\texttt{deepLookup}(\mathcal{E}, e)$ function to perform recursive lookup on expressions, which may contain variables:

$$\texttt{deepLookup}(\mathcal{E}, e) = \begin{cases} \texttt{deepLookup}(\mathcal{E}, e') & \text{if } e = x \text{ is a variable, and } \texttt{lookup}(\mathcal{E}, x) = e' \\ e & \text{otherwise} \end{cases}$$

This recursive lookup strategy is convenient for "pointer chasing" in $\mathcal{E}$. We also define the predicate $\texttt{isSymbolicVar}(\mathcal{E}, x)$, which checks if a variable is symbolic or concrete:

$$\texttt{isSymbolicVar}(\mathcal{E}, x) = \begin{cases} \text{True} & x \text{ is a symbolic variable} \\ \text{False} & \text{otherwise} \end{cases}$$

To perform program execution on the current expression $\mathcal{C}$ with respect to the graph induced by the expression environment $\mathcal{E}$, we run repeated applications of reduction rules on **RED**ucible sub-**EX**pressions (redex) within $\mathcal{C}$ until some normal form is reached. The order in which we pick the redex of the current expression delineates different evaluation semantics, such as *strict evaluation* or *lazy evaluation*. Haskell targets the left-most outer-most redex within the current expression for reduction [Marlow and Jones 2004], which requires a stack *Stk* of frames. In regular Haskell execution, this stack is used to store and delay the evaluation of subexpressions within the current expression that are not the active redex on which we reduce, and is therefore different from a conventional call stack common in other languages. In symbolic execution, we also use this stack to track logical predicates that we assume or assert about the expression under evaluation.

G2 also tracks the path constraints $\mathcal{PC}$ that needed to be satisfied so far in some state of graph reduction execution, as well as an assertion violation flag b for error localization. We initialize $b = \emptyset$, and when the associated function f with parameters $a_1, \ldots, a_n$ and return result r is violated, we set $b = (f, a_1, \ldots, a_n, r)$. Together, we write $S := (\mathcal{E}, \mathcal{C}, \mathcal{PC}, \mathit{Stk}, b)$ to represent the execution state.

Symbolic execution also differs from regular Haskell execution in that we attempt to reduce programs to *Symbolic Weak Head Normal Form* (SWHNF) instead of *Weak Head Normal Form* (WHNF) [Jones 1996]. We define the predicate $\texttt{isSWHNF}(\mathcal{E}, e)$ as follows:

$$\texttt{isSWHNF}(\mathcal{E}, e) = \begin{cases} \text{True} & \text{if } e \text{ is a literal, constructor, or lambda} \\ \texttt{isSymbolicVar}(\mathcal{E}, x) & \text{if } e \text{ is a variable } x \\ \texttt{isSWHNF}(\mathcal{E}, e_2) & \text{if } e \text{ is a } \texttt{cast}\ e_2 : \tau_1 \sim \tau_2 \\ \vee_{i=1}^{n} \texttt{isSWHNF}(\mathcal{E}, e_i) & \text{if } e \text{ is a primitive operation} \oplus e_1, \ldots, e_n \\ \text{False} & \text{otherwise} \end{cases} \tag{1}$$

SWHNF is similar to WHNF. In fact, the first three conditions listed are those for WHNF. SWHNF also includes the notion of symbolic variables, type casting, and primitive operations where at least one argument is symbolic. We further introduce the function $\texttt{uniqArgs}(x, n)$, which generates a list of n unique variables given some existing variable x and amount n. ¡¡¡¡¡¡¡ HEAD Name generation is deterministic with respect to the variable x and n.

======= Name generation is deterministic with respect to the variable x and n. ¿¿¿¿¿¿¿ 121aff74e18e60d145b7
In addition, we track typing information during symbolic execution, rather than disregarding them as regular Haskell execution does. This information is necessary for solving path constraints. We also define the $\texttt{retype}(\tau_1, \tau_2, e)$ function, which aids us in performing type deduction on polymorphic variables during runtime. $\texttt{retype}(\tau_1, \tau_2, e)$ walks over the types of all subexpression within the expression e, and replaces any instance of τ_1 with τ_2.

4.3 Execution Rules

Evaluation is performed according to the rules below. These rules resemble those used by the Spineless Tagless G-machine, as presented in [Marlow and Jones 2004]. However, there are several changes to allow for symbolic execution, the most notable of which are the addition of symbolic variables, the maintenance of type information, and the addition of rules to handle casts. Normally, Haskell execution reduces expressions to WHNF. Because of the addition of symbolic values, these rules reduce an expression to SWHNF.

VAR-IS-VAL and **VAR-NON-VAL** describe variable evaluation. The variable is looked up in the expression environment, and if it is concrete its definition is set as the current expression. If the variable is symbolic, it will not have a definition, and so we are either done executing, or a frame will be popped from the stack.

If a concrete variable's definition is not in SWHNF, we push an UpdateFrame containing the variable to the stack. This allows us to update the definition of the variable later, in rule **UPDATE-FRAME-INSERT**. In the case that the variable is used more than once, this allows only having to evaluate it once. If the expression is in SHWNF, we do not push an UpdateFrame, as the expression has already been evaluated as much as possible.

If our expression is in SWHNF, and there is nothing in our stack, then we are done evaluating.

Function application, rule **APP**, pushes the argument onto the stack in an ApplyFrame. ApplyFrames can be removed from the stack by one of 3 possible rules. **APPLY-FRAME-LAM** performs lambda bindings on non type variables, by binding the expression in the frame to a fresh variable, and renaming the lambdas body to this bound variable.

VAR-NON-VAL
$(\mathcal{E}, x, \mathcal{PC}, Stk, b) \quad \Longrightarrow \quad \{(\mathcal{E}, e, \mathcal{PC}, \texttt{UpdateFrame}\ x : Stk, b)$
if $\neg\texttt{isSWHNF}(\mathcal{E}, e)$ when $\texttt{lookup}(\mathcal{E}, x) = e$

VAR-IS-VAL
$(\mathcal{E}, x, \mathcal{F}, Stk, b) \quad \Longrightarrow \quad \{(\mathcal{E}, e, \mathcal{F}, Stk, b)\}$
if $\texttt{isSWHNF}(\mathcal{E}, e)$ when $\texttt{lookup}(\mathcal{E}, x) = e$

UPDATE-FRAME-INSERT
$(\mathcal{E}, e, \mathcal{PC}, \texttt{UpdateFrame}\ x, Stk, b) \quad \Longrightarrow \quad \{(\mathcal{E}\{x = e\}, e, \mathcal{PC}, Stk, b)\}$
if $\texttt{isSWHNF}(\mathcal{E}, e)$

Fig. 3. Variable lookup and update rules

LET
$(\mathcal{E}, \texttt{let}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e_b, \mathcal{PC}, Stk, b) \quad \Longrightarrow \quad \{(\mathcal{E}', e'_b, \mathcal{PC}, Stk, b)\}$
where $\mathcal{E}' = \mathcal{E}\{x'_1 = e'_1, \ldots, x'_n = e'_n\}$
$e'_b = e_b[x'_1/x_1, \ldots, x'_n/x_n]$
$e'_i = e_i[x'_1/x_1, \ldots, x'_n/x_n]$ for all $1 \leq i \leq n$
$x'_1, \ldots, x'_n$ are fresh variables

Fig. 4. Let expression rules

APPLY-TYPE-FRAME is for lambda type bindings. A type binding, α is either a type expression, or some variable that can be looked up in the expression environment to find a type expression. Once we have a type expression, $\texttt{type}\ \tau$, we use retype to substitute α with τ in the lambda's expression. We also ensure that future function calls will receive the correct type biding, by inserting $\texttt{type}\ \tau$ in the expression environment, with a fresh name x', and replace any occurrences of $\texttt{type}\ \alpha$ with this binding.

Finally, when applying to a Data Constructor, in rule **APPLY-FRAME-DATA**, we simply add the expression in the ApplyFrame as an argument to the data constructor.

Primitives represent functions on literal types, with no Haskell definition. Typically, literal types are in "boxes"- that is, an algebraic data type wrapper [Jones 1992]. For example, the value 7 is represented as $I\#7\#$, where $I\#$ is an algebraic data constructor of the Int type, and $7\#$ is a literal (unboxed) Int. For this reason, calls to primitive operators are surrounded by case statements, to unbox the literal types. The primitive operator application is also in a case statement, so that the resulting primitive can be put into a data constructor. For example, the function to add two Ints is:

$$\lambda x.\lambda y.\texttt{case}\ x\ \texttt{as}\ x_b\ \texttt{of}\ \{I\#\ x_u \rightarrow \texttt{case}\ y\ \texttt{as}\ y_b\ \texttt{of}\ \{I\#\ y_u \rightarrow \texttt{case}\ (x_u +\# y_u)\ \texttt{as}\ z_b\ \texttt{of}\ I\#\ z_b\}\}$$

where $+\#$ is addition of literal integers.

Pattern matching on concrete literals or data constructors is done by **CASE-LIT**, **CASE-LIT-DEF**, **CASE-DATA**, and **CASE-DATA-DEF**. **CASE-LIT** and **CASE-DATA** correspond to the scrutinee matching one of the alt patterns, whereas **CASE-LIT-DEF** and **CASE-DATA-DEF** corresponding

APP

$(\mathcal{E}, e_1\ e_2, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}, e_1, \mathcal{PC}, Stk', b)\}$

where $Stk' = \texttt{ApplyFrame}\ e_2 : Stk$

e_1 is not a data constructor or primitive

APPLY-FRAME-LAM

$(\mathcal{E}, \lambda x\ .\ e, \mathcal{PC}, \texttt{ApplyFrame}\ e_f : Stk, b) \implies \{(\mathcal{E}', e', \mathcal{PC}, Stk, b)\}$

where $\mathcal{E}' = \mathcal{E}\{x' = e_f\}$

$e' = e[x'/x]$

x' fresh

APPLY-FRAME-DATA

$(\mathcal{E}, D\ e_1, \ldots, e_k, \mathcal{PC}, \texttt{ApplyFrame}\ e_f : Stk, b) \implies \{(\mathcal{E}, e', \mathcal{PC}, Stk, b)\}$

where $e' = D\ e_1, \ldots, e_k, e_f$

APPLY-TYPE-FRAME

$(\mathcal{E}, \lambda\alpha\ .\ e, \mathcal{PC}, \texttt{ApplyFrame}\ e_f : Stk, b) \implies \{(\mathcal{E}', e', \mathcal{PC}, Stk, b)\}$

where $e' = \texttt{retype}(e[x'/\texttt{type}\ \alpha], \alpha, \tau)$

$\mathcal{E}' = \mathcal{E}\{x' = \texttt{type}\ \tau\}$

$\texttt{type}\ \tau = \texttt{deepLookup}(\mathcal{E}, e_f)$

x' a fresh variable

Fig. 5. Function application rules

PRIM

$(\mathcal{E}, \oplus e_1, \ldots, e_n, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}, \oplus e'_1, \ldots, e'_n, \mathcal{PC}, Stk, b)\}$

if there exist some $1 \leq i \leq n$ such that $\neg\texttt{isSWHNF}(\mathcal{E}, e_i)$

where $e'_i = \texttt{deepLookup}(\mathcal{E}, e_i)$ for every $1 \leq i \leq n$

Fig. 6. Primitive operation rules

to hitting a default case. In all the rules, we bind the scrutinee to a fresh name, and replace the case binder with that name in the matched alt expression. In **CASE-DATA** we also bind fresh variables to all the arguments of the data constructor, and replace those names in the matched alt expression.

When the scrutinee of a case is not in SWHNF, we put the case binding and alts from the case on the stack, and set the scrutinee as the current expression. The scrutinee is evaluated until it is in SWHNF. Then, we pop the case frame from the stack, and continue evaluating the case.

The rules **CASE-LIT-SYM** and **CASE-DATA-SYM** perform case splitting on symbolic variables. We generate one state for each alt, and add path constraints to each state to record the branch we took. States from a data or lit alt gain a single path constraint, which binds the case binder to a specific literal or data constructor. In the case of a data alt, it also binds the data alts arguments to variables, which are given by `uniqArgs`(this is useful for the optimization described in Section 4.4.) Default

CASE-LIT

$(\mathcal{E}, \texttt{case}\ l\ \texttt{as}\ x\ \texttt{of}\ alts, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}', e_i', \mathcal{PC}, Stk, b)\}$

if there is an $(l_i \to e_i) \in alts$ such that $l_i = l$
where $\mathcal{E}' = \mathcal{E}\{x' = l\}$
$e_i' = e_i[x'/x]$
e_i is the expression of the alt that matches l
x' is a fresh variable

CASE-LIT-DEF

$(\mathcal{E}, \texttt{case}\ l\ \texttt{as}\ x\ \texttt{of}\ alts, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}', e_d', \mathcal{PC}, Stk, b)\}$

where $\mathcal{E}' = \mathcal{E}\{x' = l\}$
$e_d' = e_d[x'/x]$
x' is a fresh variable

Fig. 7. Concrete literal case branching rules

CASE-DATA

$(\mathcal{E}, \texttt{case}\ D\ e_1, \ldots, e_n\ \texttt{as}\ x\ \texttt{of}\ alts, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}', e_i', \mathcal{PC}, Stk, b)\}$

if there is an $(D_i\ x_i^1, \ldots, x_i^n \to e_i) \in alts$ such that $D_i = D$
where $\mathcal{E}' = \mathcal{E}\{x_i^{1'} = e_1, \ldots, x_i^{n'} = e_n, x' = D\ e_1, \ldots, e_n\}$
$e_i' = e_i[x'/x, x_i^{1'}/x_i^1, \ldots, x_i^{n'}/x_i^n]$
$x', x_i^{1'}, \ldots, x_i^{n'}$ are fresh variables

CASE-DATA-DEF

$(\mathcal{E}, \texttt{case}\ D\ e_1, \ldots, e_n\ \texttt{as}\ x\ \texttt{of}\ alts, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}', e_d', \mathcal{PC}, Stk, b)\}$

if no alts except $\texttt{default} \to e_d$ matches D
where $\mathcal{E}' = \mathcal{E}\{x' = D\ e_1, \ldots, e_n\}$
$e_d' = e_d[x'/x]$
x' is a fresh variables

Fig. 8. Concrete data constructor case branching rules

cases generate a path constraint from every other possible alt, blocking the case binder from taking on that literal or data constructor.

In **CAST**, we push casts that are not over function types onto the stack, and continue evaluating the expression being cast. In rule **CAST-FRAME**, we pop a cast frame off the stack, and reapply it to an SWHNF expression. In **CAST-SPLIT**, we reduce casts on function types to casts on non function types. This is important to ensure CastFrames and ApplyFrames interact correctly. Consider the expression: $(\texttt{cast}\ (\lambda x\ .\ e) : \tau_1 \to \tau_2 \sim \tau_3 \to \tau_4)\ y : \tau_3$.

Without rule **CAST-SPLIT**, we would push $\texttt{ApplyFrame}\ y$ onto the stack, followed by $\texttt{CastFrame}\ \tau_1 \to \tau_2 \sim \tau_3 \to \tau_4$. The lambda expression needs y, from the ApplyFrame, but it cannot pop y, because of the cast frame.

CASE-NON-VAL
$(\mathcal{E}, \texttt{case}\ e\ \texttt{as}\ x\ \texttt{of}\ alts, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}, e, \mathcal{PC}, Stk', b)\}$
if $\neg\texttt{isSWHNF}(\mathcal{E}, e)$
where $Stk' = \texttt{CaseFrame}\ x\ \texttt{of}\ alts : Stk$

CASE-FRAME
$(\mathcal{E}, e, \mathcal{PC}, \texttt{CaseFrame}\ x\ \texttt{of}\ alts : Stk, b) \implies \{(\mathcal{E}, e', \mathcal{PC}, Stk, b)\}$
if $\texttt{isSWHNF}(\mathcal{E}, e)$
where $e' = \texttt{case}\ e\ \texttt{as}\ x\ \texttt{of}\ alts$

Fig. 9. Non-SWHNF case rules

CASE-LIT-SYM
$(\mathcal{E}, \texttt{case}\ x_1\ \texttt{as}\ x_2\ \texttt{of}\ alts, \mathcal{PC}, Stk, b) \implies S_d \cup (S_1 \cup \ldots \cup S_N)$
where $l_i \to e_i$ is the ith literal alternative
$S_i = (\mathcal{E}_i, e_i', \mathcal{PC}_i, Stk, b)$ for all $1 \leq i \leq N$ non-`default` alternatives
$\mathcal{E}_i = \mathcal{E}\{x_2' = x_1\}$
$e_i' = e_i[x_2'/x_2]$
$\mathcal{PC}_i = \mathcal{PC} \wedge x_2' = l_i$
x_2' is a fresh variable
$S_d = (\mathcal{E}_d, e_d', \mathcal{PC}_d, Stk, b)$ for the `default` alternative
$\mathcal{E}' = \mathcal{E}\{x_2' = x_1\}$
$e_d' = e_d[x_2'/x_2]$
$\mathcal{PC}_d = \mathcal{PC} \wedge \left(\wedge_{i=1}^{N} x_1 \neq l_i\right)$ for all N non-`default`
x_2' is a fresh variable

Fig. 10. Symbolic case literal branching rules

With rule **CAST-SPLIT**, we avoid this unfortunate situation. As before, we push `ApplyFrame` y. Then, rule **CAST-SPLIT** converts

$$\texttt{cast}\ (\lambda x\ .\ e) : \tau_1 \to \tau_2 \sim \tau_3 \to \tau_4$$

to

$$\lambda x_2\ .\ \texttt{cast}\ ((\lambda x\ .\ e)\ (\texttt{cast}\ x_2 : \tau_1 \sim \tau_3)) : \tau_2 \sim \tau_4$$

Now, we pop off the apply frame, and appropriately bind y *before* we push the cast frame onto the stack. Of course, τ_2 and τ_4 could also be function types, but if they are, rule **CAST-SPLIT** will just be applied repeatedly.

ASSUME and **ASSERT** handle assume and assert expressions, respectively. The assumption or assertion is set as the current expression, and the program expression is pushed onto the stack in an AssumeFrame or an AssertFrame.

When we hit an AssumeFrame or an AssertFrame in the stack, we know our current expression has been reduced to a boolean expression. In the case of an AssumeFrame, we pop it, add the current

CASE-DATA-SYM

$(\mathcal{E}, \texttt{case}\ x_1\ \texttt{as}\ x_2\ \texttt{of}\ \mathit{alts}, \mathcal{PC}, \mathit{Stk}, b) \quad \Longrightarrow \quad S_d \cup (S_1 \cup \ldots \cup S_N)$

where $D_i\ x_i^1, \ldots, x_i^n \rightarrow e_i$ is the ith data alternative

$S_i = (\mathcal{E}_i, e_i', \mathcal{PC}_i, \mathit{Stk}, b)$ for all $1 \leq i \leq N$ non-`default` alternatives

$\mathcal{E}_i = \mathcal{E}\{x_i^{1'} = sym_i^1, \ldots, x_i^{n'} = sym_i^n, x_2' = x_1\}$

$e_i' = e_i[x_2'/x_2, x_i^{1'}/x_i^1, \ldots, x_i^{n'}/x_i^n]$

$\mathcal{PC}_i = \mathcal{PC} \wedge x_1 = D_i x_i^{1'}, \ldots, x_i^{n'}$

$sym_i^1, \ldots, sym_i^n$ are fresh symbolic values

x_2' is a fresh variable

$\{x_i^{1'}, \ldots, x_i^{n'}\} = \texttt{uniqArgs}(x_1, n)$ are fresh variables on x_1

$S_d = (\mathcal{E}_d, e_d', \mathcal{PC}_d, \mathit{Stk}, b)$ for the `default` alternative

$\mathcal{E}_d = \mathcal{E}\{x_2' = x_1\}$

$e_d' = e_d[x_2'/x_2]$

$\mathcal{PC}_d = \mathcal{PC} \wedge \left(\wedge_{i=1}^{N} \neg \texttt{isCons}\ D_i\ x_1\right)$ for all N non-`default`

`default` $\rightarrow e_d$ is the ith data alternative

x_2' is a fresh variable

Fig. 11. Symbolic case data constructor branching rules

CAST

$(\mathcal{E}, \texttt{cast}\ e : \tau_1 \sim \tau_2, \mathcal{PC}, \mathit{Stk}, b) \quad \Longrightarrow \quad \{(\mathcal{E}, e, \mathcal{PC}, \mathit{Stk}', b)\}$

if $\nexists \tau_1, \tau_2$ such that $\tau_1 = \tau_i \rightarrow \tau_j$ or $\tau_2 = \tau_i \rightarrow \tau_j$, and $\neg \texttt{isSWHNF}(\mathcal{E}, e)$

where $\mathit{Stk}' = \texttt{CastFrame}\ \tau_1 \sim \tau_2 : \mathit{Stk}$

CAST-FRAME

$(\mathcal{E}, e, \mathcal{PC}, \texttt{CastFrame}\ \tau_1 \sim \tau_2 : \mathit{Stk}, \mathcal{PC}, b) \quad \Longrightarrow \quad \{(\mathcal{E}, e', \mathcal{PC}, \mathit{Stk}, b)\}$

if $\texttt{isSWHNF}(\mathcal{E}, e)$

where $e' = \texttt{cast}\ e : \tau_1 \sim \tau_2$

CAST-SPLIT

$(\mathcal{E}, \texttt{cast}\ e : \tau_1 \rightarrow \tau_2 \sim \tau_3 \rightarrow \tau_4, \mathcal{PC}, \mathit{Stk}, b) \quad \Longrightarrow \quad \{(\mathcal{E}, e', \mathcal{PC}, \mathit{Stk}, b)\}$

where $e' = (\lambda x\ .\ e\ x') : \tau_2 \sim \tau_4$

$x' = \texttt{cast}\ x : \tau_1 \sim \tau_3$

x is a fresh lambda binder variable

Fig. 12. Type casting rules

expression to the list of constraints, and continue our execution based on the expression contained in the frame.

When we hit an AssertFrame in the stack, we split into two states. In one of the states, we negate the current expression, and add it to the list of path constraints. This represents the set of states where the assertion is violated. We set the tracker in this State to the the tracker from the assert, to track

ASSUME

$(\mathcal{E}, \texttt{assume}\ e_1\ e_2, \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}, e_1, \mathcal{PC}, Stk', b)\}$

where $Stk' = \texttt{AssumeFrame}\ e_2 : Stk$

ASSERT

$(\mathcal{E}, \texttt{assert}\ e_1\ e_2\ , \mathcal{PC}, Stk, b) \implies \{(\mathcal{E}, e_1, \mathcal{PC}, Stk', b)\}$

where $Stk' = \texttt{AssertFrame}\ e_2\ : Stk$

ASSUME-FRAME

$(\mathcal{E}, e_p, \mathcal{PC}, \texttt{AssumeFrame}\ e_f : Stk, b) \implies \{(\mathcal{E}, e_f, \mathcal{PC} \wedge e_p, Stk, b)\}$

if $\texttt{isSWHNF}(\mathcal{E}, e_p)$

ASSERT-FRAME

$(\mathcal{E}, e_p, \mathcal{PC}, \texttt{AssertFrame}\ e\ _f : Stk, b) \implies \{(\mathcal{E}, e_f, \mathcal{PC}_T, Stk, b), (\mathcal{E}, e_f, \mathcal{PC}_F, Stk, b)\}$

if $\texttt{isSWHNF}(\mathcal{E}, e_p)$

where $\mathcal{PC}_T = \mathcal{PC} \wedge e_p$ and $\mathcal{PC}_F = \mathcal{PC} \wedge \neg e_p$

Fig. 13. Assume and assertion rules

which assertion was violated. When we are done evaluating, we can filter out states in which b is $\emptyset$, to find only states with assertion violations. The other State simply has it's current state set to the expression from the frame, with no constraints added. This allows us to represent states in which the assertion passed. While we could explicitly add the true assertion to our path constraint, it would serve little purpose. Either $b = \emptyset$ at the end of execution, and so this state will be filtered out, or some other assertion will be violated, and the truth of this assertion will not matter.

4.4 Constraint Solving

As described in Section 4.2, G2 generates constraints over numeric literals and algebraic datatypes. To reduce state explosion, we check the satisfiability of the path constraints every time we add a new path constraint, and discard any states with unsatisfiable constraints. We solve numeric constraints with an off-the-shelf constraint solver, and constraints on Algebraic Datatypes as described below.

Algebraic Datatype Constraints. Here, we present a new optimization to solve Algebraic Datatype constraints that are generated by G2. We adapt the constraint independence optimization introduced by EXE [Cadar et al. 2008b]. EXE views the path constraints as nodes of an undirected graph, with two nodes connected by an edge if and only if they share some variable. Then, to check the satisfiability of a new constraint, we need only consider those path constraints in it's strongly connected component. However, without some adjustment to this optimization, the pervasiveness of Algebraic Datatypes in functional languages can easily lead to very large strongly connected components, counteracting the constraint independence scheme. Consider the function **average**:

```
average :: (Real a, Fractional a) => [a] -> a
average xs = sum xs / genericLength xs

sumN :: Num a => [a] -> a
```

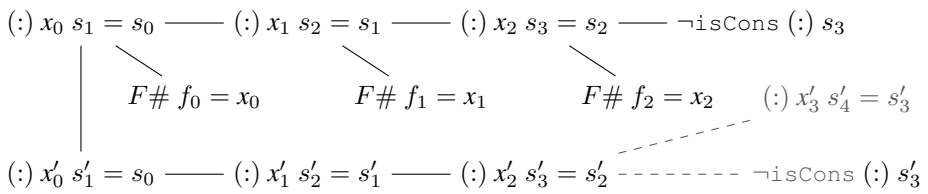

Fig. 14. Using EXE's method, the graph of the path constraints generated from sumN and lengthN, with two new possible constraints on the constructor of s'_3.

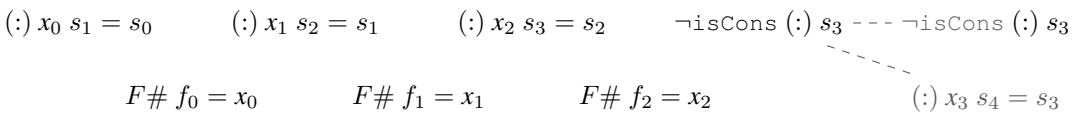

Fig. 15. Using G2's method, the graph of the path constraints generated from sumN and lengthN, with two new possible constraints on the constructor of s_3.

```
sumN _       = 0
sumN (x:xs) = x + sumN xs

lengthN :: Num b => [a] -> b
lengthN _       = 0
lengthN (_:xs) = 1 + lengthN xs
```

average must walk over the list twice- once to compute the length, once to compute the sum. Suppose we generated fresh names for data constructor arguments in Rule **CASE-DATA-SYM**. There will be a state where **sumN** generates the path constraints $\mathcal{PC}_{\text{sumN}}$:

$$(:)\ x_0\ s_1 = s_0,\quad (:)\ x_1\ s_2 = s_1,\quad (:)\ x_2\ s_3 = s_2,\quad \neg\texttt{isCons}\ (:)\ s_3,\quad F\#\ f_0 = x_0,\quad F\#\ f_1 = x_1,\quad F\#\ f_2 = x_2$$

These constraints are satisfiable, modeled by a list of three floats $F\#\ f_0, F\#\ f_1, F\#\ f_2$. Now consider the following constraints that will be generated by **lengthN**:

$$(:)\ x'_0\ s'_1 = s_0 \qquad (:)\ x'_1\ s'_2 = s'_1 \qquad (:)\ x'_2\ s'_3 = s'_2$$

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

$$\begin{array}{lllll} \mathcal{PC}_{\text{length1}} & (:)\ x'_0\ s'_1 = s_0 & (:)\ x'_1\ s'_2 = s'_1 & (:)\ x'_2\ s'_3 = s'_2 & \neg\texttt{isCons}\ (:)\ s'_3 \\ \mathcal{PC}_{\text{length2}} & (:)\ x'_0\ s'_1 = s_0 & (:)\ x'_1\ s'_2 = s'_1 & (:)\ x'_2\ s'_3 = s'_2 & (:)\ x'_3\ s'_4 = s'_3 \end{array}$$

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

To reduce this interconnectedness, we use `uniqArgs`(x, n), and change our construction of the path constraints graph. Recall from Section 4.3 that `uniqArgs` returns the same unique n names, for each x. When using it, we get the same path constraints from sumN. Considering the same two possible sets of constraints from **`lengthN`**, we will get:

$$\begin{array}{lllll} \mathcal{PC}_{\text{length1}} & (:)\ x_0\ s_1 = s_0 & (:)\ x_1\ s_2 = s_1 & (:)\ x_2\ s_3 = s_2 & \neg\texttt{isCons}\ (:)\ s_3 \\ \mathcal{PC}_{\text{length2}} & (:)\ x_0\ s_1 = s_0 & (:)\ x_1\ s_2 = s_1 & (:)\ x_2\ s_3 = s_2 & (:)\ x_3\ s_4 = s_3 \end{array}$$

Already, this has made our graph smaller, as of course we do not need to keep multiple copies of the same path constraint. However, we can reduce the size of the strongly connected components even more. We adjust our graph construction to only connect DataCon nodes with edges based on the scrutinee, not the arguments. Then, checking the satisfiability of $\mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length1}}$ requires only checking one constraint: $\neg\texttt{isCons}\ (:)\ s_3$ Similarly, determining that $\mathcal{PC}_{\text{sumN}} \wedge \mathcal{PC}_{\text{length2}}$ is unsatisfiable requires only two constraints: $\neg\texttt{isCons}\ (:)\ s_3 \wedge (:)\ x_3\ s_4 = s_3$.

This optimizations relies on all constraints on ADTs being of the form $D\ x_1 \ldots x_n = x$ or $\neg\texttt{isCons}\ D\ x$. Fortunately this is the case, as, given a symbolic variable x that is a value of some algebraic datatype, constraints on x will only ever be generated by rule **CASE-DATA-SYM**. To see why, consider the other rules that generate constraints: rule **CASE-LIT-SYM**, rule **ASSUME-FRAME**, and rule **ASSERT-FRAME**. Since x is not a literal, constraints on x will also clearly not be generated by **CASE-LIT-SYM**.

Now, we consider constraints from the rules **ASSUME-FRAME** or **ASSERT-FRAME**. To help with this, we define an expression in *symbolic weak head normal literal form* (SWHNLF) as an expression in SWHNF, which is one of a literal, a literal variable, or a primitive operator whose arguments are in SWHNF. The expression may not be a cast, since you cannot cast to or from a literal. Recall that, by definition, if a primitive operator is in SWHNF, its arguments are also in SWHNF. Since primitive operators only work with literals as input and output, this implies that if a primitive operator is in SWHNF, it's arguments are in SWHNLF.

We now return to the constraint from the rules **ASSUME-FRAME** or **ASSERT-FRAME**. Since the constraint must be of type Bool, and has been evaluated to SWHNF, there are three possibilities: The constraint is a Boolean literal, the constraint is a symbolic Boolean variable, or the constraint is a primitive operation. The first two options clearly do not impose any constraints on x, so we consider only primitive operators. Since the primitive operator is in SWHNF, we know the arguments are in SWHNLF. Therefore, they also do not impose any constraints on x.

We conclude that constraints on x will come only from the rule **CASE-DATA-SYM**, and will only be in two forms: $D\ x_1 \ldots x_n = x$ and $\neg\texttt{isCons}\ D\ x$.

Finally, given all the constraints on x, we determine which constructors x may use as follows. Construct a set of all constructors for the x's type. For a constraint of the form $D\ x_1 \ldots x_n = x$, remove all constructors except D from the set. For a constraint of the form $\neg\texttt{isCons}\ D\ x$, remove D from the set. When you finish, if the set is nonempty, then x may use any constructor in the set. If the set is empty, then the constraints are unsatisfiable.

5 FACETED SYMBOLIC EXECUTION FOR LIQUIDHASKELL

Now that we have described the design of our general symbolic execution engine, we turn our attention to how it can be adapted to generate counterexamples to incorrect LiquidHaskell refinement types. We currently support a subset of the LiquidHaskell refinement language, including operations on numeric types, measures, and refinements on polymorphic arguments.

5.1 Refinement Type Translation

We translate LiquidHaskell refinement types into predicates in our Core language. Given an arity n function f of type $\tau_1 \to \ldots \to \tau_n \to \tau$, and the refinement of f, f_r, we translate f_r into a predicate, f_p of arity $n+1$, with type $\tau_1 \to \ldots \to \tau_n \to \tau \to \texttt{Bool}$. f_p is true if and only if it's arguments satisfy the refinement f_r. Given $\texttt{lookup}(\mathcal{E}, f) = e$, we replace the existing definition of f with:

$$\lambda x_1 ... \lambda x_n . \texttt{let } x = e\ x_1 \ldots x_n \texttt{ in assert } (f, (x_1 \ldots x_n), x)\ (f_p\ x_1 \ldots x_n)\ x$$

We introduce new lambda bindings, to match arguments passed to f. Then we evaluate f in a let expression, storing the result in x. Finally, we assert that the refinement type holds, and return x. We use a let expression to evaluate f to ensure this evaluation only happens once. Otherwise, it might be evaluated by both f_p, and the caller of f.

Refinement types can be applied to an argument, α of a polymorphic datatype $T\ \alpha$. To handle this in our Core Language, we introduce a new, recursive function, of type. $(\alpha \to \texttt{Bool}) \to T\ \alpha \to Bool$. This function recursively walks over each constructor of $T\ \alpha$, and applies $(\alpha \to \texttt{Bool})$ to each α. The results of these calls are anded, and returned. Then, we use this function to assume or assert that the required refinement holds for all α's.

5.2 Counter Example Generation

As described in Section 3.2.2, when proving a function correct, LiquidHaskell does not look at the definition of any called functions. Rather, it just looks at those functions refinement types. This can lead to situations in which a function definition is correct, but LiquidHaskell cannot verify the function. In order to give useful feedback to the user, we extend G2 to find *abstract counterexamples*. An abstract counterexample for a function f is an input that conforms to f's refinement type, and a new (partial) definition for some function g that f calls, which satisfies the refinement of g, but causes f's refinement to fail. Then, the user knows they need to strengthen the refinement on g, so that LiquidHaskell knows everything about g that it needs to, to verify f.

To find abstract counterexamples, we replace **VAR-NON-VAL** and **VAR-IS-VAL** with new rules. When we evaluate a function, these new rules split into two states. One state works exactly like the original rules, and allows us to find concrete counterexamples, if they exist. In the other state, we replace the result of the function with a symbolic variable, and assume the refinement type holds on this symbolic output.

We create a list of top level functions and constants in the Haskell program, B. We only replace the top level definitions with symbolic variables, to properly model the behavior of LiquidHaskell. Then we define a function to switch out a Haskell function definition with a symbolic variable.

$$\texttt{replaceWithSym}(e, x) = \begin{cases} \lambda x_2\ .\ \texttt{replaceWithSym}(e_2, x) & \text{if } e = \lambda x_2 . e_2 \\ \texttt{let } x_2 = x \texttt{ in assume } e_1\ x_2 & \text{if } e = \texttt{let } x_2 = e_x \texttt{ in } e_1\ x_2 \\ x & \text{otherwise} \end{cases}$$

Notice that we assume the function refinement, and return only the symbolic variable.

To track the functions that we are replacing with symbolic variables, we add a new field, a list $\overrightarrow{b}$ to our state:

$$(\mathcal{E}, e, \mathcal{PC}, \texttt{UpdateFrame } x : Stk, b, \overrightarrow{b})$$

All existing rules from Section 4.3 are updated to copy the $\overrightarrow{b}$ from the initial state to all output states.

We define a function $\texttt{createTracker}(f, e, r)$, to create a tracker from a function name, f, an expression e, and a variable r:

$$\texttt{createTracker}(f, e, r) = (f, \texttt{lamVars}(e), r)$$

$$\texttt{lamVars}(e) = \begin{cases} x : \texttt{lamVars}(e_2) & \text{if } e = \lambda x \,.\, e_2 \\ [\,] & \text{otherwise} \end{cases}$$

Finally, we replace **VAR-NON-VAL** and **VAR-IS-VAL** with four new rules:

LH-VAR-NON-VAL
$(\mathcal{E}, x, \mathcal{PC}, \mathit{Stk}, b, \vec{b}) \implies \{(\mathcal{E}, e, \mathcal{PC}, \mathit{Stk}', b, \vec{b})\}$
if $x \notin B$ and $\neg\texttt{isSWHNF}(\mathcal{E}, e)$ where $e = \texttt{lookup}(\mathcal{E}, e)$

LH-VAR-VAL
$(\mathcal{E}, x, \mathcal{PC}, \mathit{Stk}, b, \vec{b}) \implies \{(\mathcal{E}, e, \mathcal{PC}, \mathit{Stk}, b, \vec{b})\}$
if $x \notin B$ and $\texttt{isSWHNF}(\mathcal{E}, e)$ where $e = \texttt{lookup}(\mathcal{E}, e)$

Fig. 16. LiquidHaskell evaluation rules for variables not in B

The first two rules evaluate a variable that is not in B. In this case, we do not generate abstract counterexamples, and simply continue evaluation as normal.

LH-VAR-NON-VAL
$(\mathcal{E}, x, \mathcal{PC}, \mathit{Stk}, b, \vec{b}) \implies \{S_1, S_2\}$
if $x \in B$ and $\neg\texttt{isSWHNF}(\mathcal{E}, e)$
where $S_1 = (\mathcal{E}, e, \mathcal{PC}, \texttt{UpdateFrame}\ x : \mathit{Stk}, b, \vec{b})$
$S_2 = (\mathcal{E}, \texttt{replaceWithSym}(e, x_2), \mathit{Stk}, b, \texttt{createTracker}(x, e, x_2) : \vec{b})$
$e = \texttt{lookup}(\mathcal{E}, e)$
x_2 is a fresh variable

LH-VAR-VAL
$(\mathcal{E}, x, \mathcal{PC}, \mathit{Stk}, b, \vec{b}) \implies \{S_1, S_2\}$
if $x \in B$ and $\texttt{isSWHNF}(\mathcal{E}, e)$
where $S_1 = (\mathcal{E}, e, \mathcal{PC}, \texttt{UpdateFrame}\ x : \mathit{Stk}, b, \vec{b})$
$S_2 = (\mathcal{E}, \texttt{replaceWithSym}(e, x_2), \mathit{Stk}, b, \texttt{createTracker}(x, e, x_2) : \vec{b})$
$e = \texttt{lookup}(\mathcal{E}, e)$

Fig. 17. LiquidHaskell evaluation rules for variables in B

These two rules split into two states. When we introduce a symbolic variable, we do not push an update frame. If we pushed an update frame, and x was a constant, we would replace the value of the constant in $\mathcal{E}$ with the symbolic variable when executing **UPDATE-FRAME-INSERT**.

When displaying abstract counterexamples, we only show those with a minimal number of abstracted function definitions. From a users point of view, counterexamples with fewer abstracted functions are likely easier to understand. Furthermore, this allows us to cut down on state explosion. If we have already found a counterexample that requires n abstracted functions, we stop evaluating any state with $n + 1$ or more abstracted functions.

5.2.1 Correctness of the algorithm. It should be noted that, occasionally, this algorithm generates false positives. That is, it generates examples that LiquidHaskell actually can determine do not occur, due to LiquidHaskell's refinement type inference rules. This is because those inference rules are not yet fully implemented in G2, and so G2 typically uses just the base function refinement types. In practice, we have found that G2 often generates just correct, helpful counterexamples, and even when there are incorrect counterexamples, correct counterexamples are also present. Nonetheless, in the future, we aim to improve G2's ability to reason about refinement type inference.

6 IMPLEMENTATION AND EVALUATION

We have implemented G2 in Haskell. To parse Haskell program sources, we leverage GHC's API. For constraint solving, we support both Z3 and CVC4 as SMT backends. G2 is open source, and available at **omitted for anonymity**. We also offer a custom version of Haskell's Base library and Prelude [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 official Base and Prelude. We plan on expanding the range of supported features in the future.

Because symbolic execution is a bounded model checking approach to software verification, measures such as timeouts or counters are often implemented within the engine to prevent indefinite execution. We accept a step counter parameter, which controls the number of reduction rules (see Section 4.3) down each branching path generated during symbolic execution. We also allow setting a timeout, and stopping the search after a specified number of counterexamples are found.

We present evaluations using G2 to perform test generation and error localization of Haskell programs, and analyze the program coverage and performance achieved.

6.1 Unverifiable Refinement Type Error Localization

We demonstrate G2's ability to perform error localization by running it on homeworks submitted by students in the course discussed in section 3. By running G2 over the codebase, we found refinement type violations and generated counterexamples that would induce these errors. We examine some of these different errors that G2 caught in the case studies below.

6.1.1 List Concatenation. We consider an assignment where students were tasked with writing a `concat` function, and giving it an appropriate refinement type. One student submitted the following:

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

Here, the student claims that the result of `List` concatenation yields a `List` of length at least the number of input lists in the input. However, this fails to account for the case of empty `List`s in the input. G2 returns a concrete counterexample on which the refinement type is violated:

```
The call
```

```
concat (Emp :+: Emp) = Emp
violates concat's refinement type
```

Indeed, this violates `concat`'s refinement type: The input list `Emp :+: Emp` has size 2, but the output list `Emp`, has size 0. Running G2 for longer, we also detected a case when the target function `concat` made a (recursive) call to `concat`, which violated the callee `concat`'s post-condition:

```
concat (Emp :+: (Emp :+: Emp)) = Emp
makes a call to
concat (Emp :+: Emp) = Emp
violating concat's refinement type
```

Because of the `concat` post-condition in the refinements on `concat`, it is possible that the callee `concat` will also yield a null `List`. This also happens to be a violation on the post-condition of the calling target `concat`.

6.1.2 Map Entry Aggregation. We now examine a related assignment where students were tasked with writing annotations for functions over a `Map` data structure, which is akin to a key-value based binary tree. In this part of the assignment, students were to provide refinement types for the `collapse` function, which goes through all the values mapped to in the `Map`, and aggregates them into a single value. One student submitted the following program:

```
-- module MapReduce
import List
import qualified Data.Map as M

{-@ collapse :: (v -> v -> v) -> M.Map k {l : (List v) | true}
                      -> M.Map k v @-}
collapse f = M.map (foldr1 f)

-- foldr and foldr1 imported from List
{-@ foldr1 :: (a -> a -> a) -> {v : List a | size v > 0} -> a @-}
foldr1 op (x :+: xs) = foldr op x xs
foldr1 op Emp        = die "Cannot call foldr1 with empty list"

foldr :: (a -> b -> b) -> b -> List a -> b
foldr _  b Emp         = b
foldr op b (x :+: xs) = x `op` (foldr op b xs)
```

The student does not account for the fact that `foldr1` requires a non-null `List` as input, an error which G2 catches and reports.

```
collapse asTypeOf (fromList ([(0, Emp)])) = error
makes a call to
foldr1 asTypeOf Emp = error
violating foldr1's refinement type
```

We note that `asTypeOf` is exported from the Haskell Prelude, and is a function that satisfies the first parameter type `a -> a -> a` of `foldr1`. Because G2 also parses through custom versions of

Prelude and libraries such as Containers (which contains `Map`), we have these functions in scope, and generated as concrete example input for symbolic execution.

6.1.3 Data Clustering. In this assignment, students implemented the k-means clustering algorithms. Here we consider an erroneous `kmeans1`, which updates the center average based on new inputs:

```
-- module KMeans
type Point   = List Double
type Cluster = (Int, Point)
type Center  = Int
type Centering = M.Map Center Point

{-@ kmeans1 :: k : Nat -> n : Nat -> {x : List (PointN n) | size x = n}
                       -> CenteringKN k n -> CenteringKN 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

-- mapReduce and group imported from MapReduce
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
    kvsm = group kvs
    kvm  = collapse fr kvsm

{-@ group :: (Ord k) => List (k, v)
                 -> M.Map k {l:(List v) | size l > 0} @-}
group = foldr addKV M.empty
```

Here the error is subtle: it is possible to call `kmeans1` with empty `List`s as input. This would then propagate to the `newClusters` function, which calls `mapReduce`. The `mapReduce` function makes an additional call to `group`, which states that the output `List` should be non-null. However, this may not be the case given the student's implementation of `kmeans1`. G2 reports this error.

```
kmeans1 1 0 Emp (fromList []) = error
violating kmeans1's refinement type
when
foldr addKV (empty) (expand fm Emp) = fromList ([(1, Emp)])
Strengthen the refinement type of foldr to eliminate this possibility
```

Here G2 performs abstract counterexample generation using type information. With the current type annotations present, LiquidHaskell cannot prove that users *does not* call `kmeans1` with erroneous

G2 Performance Analysis				
				`List.lhs` (50 Files)
Function	LH Errors	Concrete	Abstract	Runtime (s)
`concat`	30	19	6	14.7
`zipWith`	7	0	0	5.609
`length`	6	2	0	5.7
`replicate`	4	0	0	5.6
`foldr1`	3	2	0	5.6
`foldr`	2	2	0	5.4
				`MapReduce.lhs` (50 Files)
Function	LH Errors	Concrete	Abstract	Runtime (s)
`expand`	29	12	17	5.5
`mapReduce`	24	16	5	5.8
`collapse`	21	20	0	5.6
`group`	2	0	2	6.8
				`KMeans.lhs` (15 Files)
Function	LH Errors	Concrete	Abstract	Runtime (s)
`nearest`	9	9	0	6.8
`distance`	6	4	0	6.8

Table 1. Counterexamples generated from student's LiquidHaskell problem set submissions.

input. Although the definition of `fm` guarantees the construction of a non-null **`List`**, LiquidHaskell cannot detect this. By providing this counterexample that trigger localized type errors, G2 yields feedback on how, and where, to strengthen refinement types to assist LiquidHaskell.

6.2 Performance

To measure performance, we used the codebase of erroneous files submitted by students to evaluate the speed and coverage G2's counterexample generation. Each time a student submitted a file for LiquidHaskell type checking, any errors generated were logged, and files were sorted into "safe" and "unsafe" categories with their logs. A good metric for performance was therefore to see how many of these errors could be reproduced from the "unsafe" files' logs.

However, a large number of submissions in this codebase were incomplete, with nubbed implementations using catch-alls designed to cause LiquidHaskell to report a type check error. Many of the "unsafe" files in the codebase were therefore just unfinished assignments. Because these types of submissions inflate the number of errors present, we opted to sample a number of these files by hand for evaluation. A total of three different categories of files that students submitted were available: `List.lhs`, `MapReduce.lhs`, and `KMeans.lhs`. This was the same set of files that we presented case studies on in Section 6.1 above. We randomly picked 50 "unsafe" files from `List.lhs` and `MapReduce.lhs`, and 15 from `KMeans.lhs`. Because some submissions were non-compiling Haskell programs, and others also contained the catch-all functions, we discarded the non-functioning ones in the batch to pick new ones at random over a few iterations. We have less files from `KMeans.lhs`, because the types of errors students were making were more homogenous.

For each file in our sample, we logged whether or not G2 was able to generate counterexamples for each instance of an error. Furthermore, we measured the performance of enabling and disabling the

G2 Optimization Analysis		
		List.lhs
Function	Opt (s)	No Opt (s)
zipWith	6.3	7.3
length	6.5	8.8
replicate	6.0	6.2
foldr1	6.0	6.5
foldr	5.9	6.0
		RedBlack.hs
Function	Opt (s)	No Opt (s)
insert	8.9	560.7

Table 2. Comparisons of runtimes with and without the ADT Constraint Optimization.

optimizations described in Sec. 4.4. The times shown are the average for all attempts, both successful and unsuccessful, to find counterexamples for instances of the function. On the List and MapReduce files, G2 was run for 1000 steps of reduction rule applications with a 300 second timeout per query. On the KMeans files, due to the complexity of the functions, we ran G2 for 2000 steps with a 300 second timeout per query. We set the max counterexamples to find to 3. Each execution of a function in a file is treated as a fresh execution of G2 without reuse of previous execution information on the same file. Our results are shown in Table 1.

6.3 ADT Constraint Optimization

In Section 4.4, we describe an optimization to improve the speed of ADT constraint solving. With this optimization, we construct a more efficient path constraint graph, and avoid calling any external solvers to reason about ADTs. To test the effect of this optimization, we implemented a conversion of ADT constraints into SMT formulas, which is sufficient to handle the non-`concat` functions from List.lhs. Results are shown in Table 2.

While the difference in runtimes for the LiquidHaskell tests are small, other tests make the potential improvements from the optimization clearer. In particular, it improves performance on functions with a heavy reliance on complex algebraic datatypes. For example, we ran unassisted symbolic execution, with 500 steps, on the `insert` function of Okasaki's Red Black Trees [Okasaki 1999], both with and without the ADT optimization. With the optimization, G2 generated 162 inputs/output pairs in 8.9 seconds. Without the optimization, G2 took 9 minutes and 30 seconds.

7 RELATED WORK

7.1 Haskell 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.

7.2 Symbolic Execution

Functional. CutEr [Giantsios et al. 2015] is a symbolic execution engine for Erlang programs. Unlike Haskell, Erlang is strict, non-currying, and emphasizes code hot swapping, and so CutEr has a different execution model than G2.

[Tobin-Hochstadt and Van Horn 2012] uses symbolic execution to verify higher-order contracts in Racket. Similarly, Xu's work on static contract checking [Xu 2006] and Halo [Vytiniotis et al. 2013] both verify properties of Haskell programs, using symbolic execution like techniques. This is a different focus than G2, and as such requires different mthods. G2 attempts to find inputs that will lead to fully exploring the program or to refinement type contradictions, where these tools attempt to abstract away details from the code, so they can verify specifications.

Imperative. Most existing work on symbolic execution applies to imperative programming languages or architectures, including 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 it's sequel Klee [Cadar et al. 2008a] for LLVM. The execution semantics of imperative are quite different from ours, but techniques related to path search strategy or constraint solving are often still applicable to Haskell symbolic execution.

8 FUTURE DIRECTIONS

Currently, G2's support for polymorphic types and symbolic higher order functions simply tries instantiating with arbitrary types and correctly-typed functions. Unfortunately, this can easily miss paths, if it does not try the right functions. Reasoning about higher order functions is, in fact, a common difficulty for many functional program analysis tools, including Catch [Mitchell and Runciman 2008], Reach [Naylor and Runciman 2007], and CutEr [Giantsios et al. 2015]. In future work, we hope to address this, possibly through the use of uninterpreted functions in the SMT solver. Depending on the application, uninterpreted functions could be shown directly to the user, or help guide a search to actual Haskell functions.

9 CONCLUSION

In this paper we highlight the importance and difficulty of functional program reachability analysis, and present the G2 symbolic execution engine to meet this challenge. We have implemented G2 to target Haskell, and demonstrated how the approach and optimizations taken may be generalized to a wider class of functional languages. In addition, we showcase the power of G2 in practice by using it as a backend for LiquidHaskell to find and prove errors in real-life user studies.

REFERENCES

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.

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

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

C. Fournet, N. Swamy, J. Chen, P-É. Dagand, P-Y. Strub, and B. Livshits. 2013. Fully abstract compilation to JavaScript. In *POPL*.

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.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 1–17.

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.

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

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

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.

M. Kawaguchi, P. Rondon, and R. Jhala. 2009. Type-based Data Structure Verification. In *PLDI*.

A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. 2016. Occurrence typing modulo theories. In *PLDI*.

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

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.

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

Chris Okasaki. 1999. Red-black trees in a functional setting. *Journal of functional programming* 9, 4 (1999), 471–477.

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.

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.

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*.

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

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.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. ACM, 53–66.