G2Q: Haskell Constraint Solving

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

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

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

Abstract

Constraint solvers give programmers a useful interface to solve challenging constraints at runtime. In particular, SMT solvers have been used for a vast variety of different, useful applications, ranging from strengthening Haskell's type system to verifying network protocols.

Unfortunately, interacting with constraint solvers directly from Haskell requires a great deal of manual effort. Data must be represented in and translated between two forms: that understood by Haskell, and that understood by the SMT solver. Such a translation is often done via printing and parsing text, meaning that any notion of type safety is lost. Furthermore, direct translations are rarely sufficient, as it typically takes many iterations on a design in order to get optimal – or even acceptable – performance from a SMT solver on large scale problems. This need for iteration complicates the translation issue: it is easy to introduce a runtime bug and frustrating to fix said bug.

To address these problems, we introduce a new constraint solving library, G2Q. G2Q includes a quasiquoter that allows solving constraints written in Haskell itself, thus unifying data representation, ensuring correct typing, and simplifying development iteration. We describe the API to our library and its backend. Rather than a direct translation to SMT formulas, G2Q makes use of the G2 symbolic execution engine. This allows G2Q to solve problems that are out of scope when directly encoded as SMT formulas. Finally, we demonstrate the usability of G2Q via four example programs.

CCS Concepts • **Computing methodologies → Symbolic and algebraic manipulation**; • **Software and its engineering → Software libraries and repositories**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell '19, August 22–23, 2019, Berlin, Germany
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342590

Keywords constraint solving, Haskell, constraint programming, symbolic execution

ACM Reference Format:
William T. Hallahan, Anton Xue, and Ruzica Piskac. 2019. G2Q: Haskell Constraint Solving. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3331545.3342590

1 Introduction

The advancements in constraint solvers, such as integer linear programming and SMT solvers, have enabled a range of new programming language tools. Such solvers have brought tackling previously intractable NP-hard problems into the realm of practicality. In particular, SMT solvers have been applied to a wide variety of challenges, including tools that strengthen the Haskell type system [13, 33, 42–44], test Haskell functions [37], verify DSL programs [17], and synthesize code [35].

Unfortunately, using Haskell to interact with a SMT solver requires a significant amount of engineering effort. When applying SMT solvers on some formula, it is frequently the case that variables, predicates and functions appearing in that formula require two representations: one in a traditional programming language, and one in the language of SMT solvers via the SMT-LIB format [8]. Additionally, one also needs to develop a parser which translates from each representation to the other. Furthermore, one of the most ubiquitous means of communication is via textual representation, which offers no type safety. These issues are compounded by the fact that, often, a direct translation of a problem is not enough. SMT solvers are sensitive to the encoding scheme [2], and it often requires several iterations to arrive at the best translation of a problem to a formula or formulas. In the process of this iteration it is – naturally – easy to introduce bugs and mistranslations [1, 32].

To make this more concrete, consider the following scenario: our goal is to write a function, `sumToN`, which takes as input two variables: `n`, an `Int`, and `xs`, a list of `Int`s. The function `sumToN` needs to return a non-empty list of `Int`s `ys` such that the sum of all elements of `ys` is `n`, and every element in `ys` also appears in `xs`.

One way to approach this problem would be to make use of a SMT solver. Figure 1 contains an encoding of this problem in the SMT-LIB format. The encoding can be seen

```
(declare-datatypes (T)
  ((list nil (cons (head T) (tail list)))))

(define-fun-rec sum ((zs (list Int))) Int
  (ite (is-nil zs)
    0 (+ (head zs) (sum (tail zs)))))

(define-fun-rec length ((zs (list Int))) Int
  (ite (is-nil zs)
    0 (+ 1 (length (tail zs)))))

(define-fun-rec elem
  ((z Int) (zs (list Int))) Bool
  (ite (is-nil zs) false
    (ite (= z (head zs)) true
      (elem z (tail zs)))))

(declare-const xs (list Int))
(declare-const ys (list Int))
(declare-const n Int)

(assert (= (sum ys) n))
(assert (forall ((y Int))
    (implies (elem y ys) (elem y xs))))
(assert (>= (length ys) 1))
(assert (= xs XS))
(assert (= n N))

(check-sat)
(get-model)
```

Out: Unknown

Figure 1. Finding the solution to sumToN using a direct encoding to a SMT solver.

as a template describing the above: we first represent a list datatype in the SMT-LIB format, next we define functions to sum the elements of the list, we then check if an element is in a list, and finally we calculate the length of the list (to check if the list is non-empty). Finally, a SMT solver is invoked at runtime when the values of n and xs are known. We therefore have two more assertions, containing the variables XS and N, which are instantiated with concrete values at runtime.

Analyzing the code in Figure 1 we see that we had to duplicate much of what already exists in Haskell: the list datatype, and three functions to manipulate and examine it. Furthermore, in order to call this code with lists from a Haskell program, we still need to write code to translate a Haskell list to a SMT list. After completing all these tasks, unfortunately our efforts were fruitless: even for very simple values of xs and n (for example xs = [-5, 5] and n = 0), when the formula is passed to a state-of-the-art SMT solver, Z3 [12], Unknown is returned, meaning it could neither find a value for ys nor determine if such a value exists. Our experimental evaluations found that Z3 cannot find a solution as soon as xs has two or more elements due to difficulties that SMT

```
sumToN :: Int -> [Int] -> IO (Maybe [Int])
sumToN = [g2| \(n :: Int) (xs :: [Int]) ->
              ?(ys :: [Int])
            |  sum ys == n
            && all (\e -> e `elem` xs) ys
            && length ys >= 1 |]

main :: IO ()
main = do
    print =<< sumToN 0 [-5, 10, -15, 20, 25]
```

Out: Just [20,-15,-5]

Figure 2. Encoding the sumToN problem in the g2 quasiquoter.

solvers have with handling quantified formulas and recursive definitions [22].

To address all these problems, we introduce G2Q, a new library that defines the g2 quasiquoter to simultaneously empower and simplify the solving of complex constraints. A quasiquoter [31] is a way of using metaprogramming to embed a domain specific language (DSL) into Haskell. At compile time, the code encapsulated in the quasiquoter is automatically translated into traditional Haskell code using Template Haskell metaprogramming [38].

The g2 quasiquoter, [g2|...|], allows Haskell programmers to write constraints in a flexible, type-safe language: Haskell itself. Programmers do not need to concern themselves at all with the low level details of external constraint solvers. Rather, the library's quasiquoter allows a programmer to write a predicate using traditional Haskell syntax and Haskell functions while making use of concrete (runtime determined) variables, and *symbolic* (unknown) variables. The quasiquoter generates code that will at runtime accept the concrete arguments and return either (1) Nothing if no values for the symbolic variables that will satisfy the predicate are found or (2) Just values for the symbolic variables.

Figure 2 contains the aforementioned sumToN problem, written using our quasiquoter. The quasiquoter takes two concrete arguments, xs and n, and returns an IO (Maybe [Int]) – a value for ys, if one exists. The function returns in the IO monad because G2Q's constraint solving may be nondeterministic.

Obviously, the quasiquoter code is significantly shorter and also allows us to reuse the existing Haskell datatypes and functions. There is another key advantage to using our library: G2Q is not simply bindings to a SMT solver. Rather, under the hood, G2Q makes use of an existing Haskell symbolic execution engine, G2 [20]. Symbolic execution is a technique that allows running a program with symbolic inputs. By using symbolic execution, we can reduce the Haskell predicates to constraints over the symbolic inputs, and then solve those constraints with a SMT solver. The key to G2Q's increased effectiveness, versus the effectiveness of directly calling a SMT

solver, is that a large number of optimizations have been applied to G2's execution and constraint solving. In particular, techniques such as function unrolling and constraint independence optimizations [9] allow G2Q to solve predicates making use of recursive functions and datatypes. As a result, the `g2` quasiquoter can actually solve many problems that are not solvable with more direct SMT encodings. For instance, G2Q is capable of handling inputs to `sumToN` that challenge SMT solvers. Running `sumToN 0 [-5, 10, -15, 20, 25]` outputs a valid solution: `Just [20,-15,-5]`.

The G2 symbolic execution engine used by G2Q is specially designed to support Haskell's non-strict semantics. G2 implements lazy reduction rules on a typed, functional, intermediate representation, which resembles GHC Core Haskell [24]. At compile time, the `g2` quasiquoter converts constraints from Haskell code into G2's intermediate representation. Furthermore, it instruments the code with functions to translate input-output between their actual values and value representations in G2's intermediate language. As functions to perform these conversions are defined in a derivable typeclass, the details of this translation is hidden from users of the G2Q library.

To evaluate G2Q, we wrote four programs using it. These programs demonstrate a range of use cases ranging from an n-queens problem solver to a program analyzer. They also demonstrate a spectrum of complexity, suggesting possible ways G2 could be improved and optimized in the future.

In short, we make the following contributions:

1. We describe our library, which provides a quasiquoter to give programmers access to the capabilites of constraint solving via writing Haskell predicates. In addition, we describe the quasiquoter's *strictness* and *fairness* properties, which govern how the quasiquoter handles infinitely large data structures, and searches over infinitely large sets of values.
2. Behind the scenes, the quasiquoter is using a Haskell symbolic execution engine, G2, to reduce the user-written Haskell code to constraints that are solvable by SMT solvers. We describe how we compile a quasiquoter to a form that is runnable in G2.
3. We show code for a number of case studies, demonstrating a variety of use cases of our library. In the next section, we will describe a technique to easily convert a program executor to a program analyzer. In Section 5, we will present three additional use cases.

2 G2Q for Program Analysis

Haskell is frequently used to implement programming languages and DSLs [5, 16, 17]. Here, we consider a simple imperative language with support for basic arithmetics as shown in Figure 3.

The language supports *assertion* statements, a common technique for performing sanity checks and error detection during software development. The `evalStmts` function is responsible for running a program. It accepts an `Env` – which maps variables to values, as an input – and also allows the caller to specify initial values. `evalStmts` and its subfunctions either return `Just` some type if they succeed, or `Nothing` if an assertion is violated.

Although this language and its interpreter are rather small, it suffices to represent many large imperative programs. Without specialized tooling and engineering overhead, it can be difficult to tell *if* and *how* an assertion will fail.

```haskell
type Ident = String
type Env = [(Ident, Int)]
type Stmts = [Stmt]

data AExpr = I Int | Var Ident
    | Add AExpr AExpr | Mul AExpr AExpr
  deriving (Show, Eq, Data)
$(derivingG2Rep ''AExpr)

data BExpr = Not BExpr
    | And BExpr BExpr | Or BExpr BExpr
    | Lt AExpr AExpr | Eq AExpr AExpr
  deriving (Show, Eq, Data)
$(derivingG2Rep ''BExpr)

data Stmt = Assign Ident AExpr | Assert BExpr
    | If BExpr Stmts Stmts | While BExpr Stmts
  deriving (Show, Eq, Data)
$(derivingG2Rep ''Stmt)

evalA :: Env -> AExpr -> Int
evalA = ...

evalB :: Env -> BExpr -> Bool
evalB = ...

evalStmt :: Env -> Stmt -> Maybe Env
evalStmt e (Assign ident aexpr) =
  Just $ (ident, evalA e aexpr) : e
evalStmt e (If bexpr lhs rhs) =
  if evalB e bexpr
    then evalStmts e lhs
    else evalStmts e rhs
evalStmt e (While bexpr loop) =
  if evalB e bexpr
    then evalStmts e (loop ++ [While bexpr loop])
    else Just e
evalStmt e (Assert bexpr) =
  if evalB e bexpr then Just e else Nothing

evalStmts :: Env -> Stmts -> Maybe Env
evalStmts = foldM evalStmt
```

Figure 3. Simple arithmetics language

```
prog :: Stmts
prog =
  [ Assign "k" (I 1),
    Assign "i" (I 0),
    Assign "n" (I 5),
    While (Or (Lt (Var "i") (Var "n"))
              (Eq (Var "i") (Var "n")))
        [ Assign "i" (Add (Var "i") (I 1))],
    Assign "z" (Add (Var "k")
               (Add (Var "i") (Var "j"))),
    Assert (Lt (Mul (Var "n") (I 2)) (Var "z"))
  ]
```

Figure 4. A program inspired from [14] written with the simple arithmetic language shown in Figure 3. It accepts a variable `"j"` as input, and checks an assertion at its end.

Consider, for instance, the program in Figure 4. This program contains an assertion that claims that upon completing execution, the assertion `(Lt (Mul (Var "n") (I 2)) (Var "z")` will hold. From just manually examining the program, it is not immediately clear whether inputs exists that violate this assertion. Of course, one could rely on testing, but such approaches still require picking the correct values to violate an assertion.

G2Q provides an easy way for the language developer to find assertion violations through a *symbolic search* over the space of inputs by leveraging the existing `evalStmts` function. The developer can simply write the following function:

```
badEnvSearch :: Stmts -> IO (Maybe Env)
badEnvSearch
  = [g2|\(stmts :: Stmts) -> ?(env :: Env) |
          evalStmts env stmts == Nothing|]
```

`badEnvSearch` takes a concrete `Stmts` as an arguments, and searches for an `Env` that causes `evalStmts` to evaluate to `Nothing`, thereby indicating an assertion violation.

We can run this on the program, to see if it can find an assertion violation. The call:

```
env <- badEnvSearch prog
putStrLn $ show env
```

returns `Just [("j",-18)]`, revealing that an assignment of $j = -18$ will lead to an assertion violation. Notably, no random testing occurred to land on the value -18. Rather, constraints were generated from `evalStmts` and solved to determine that $j = -18$ would violate an assertion.

3 Solver-Aided Interface

In this section, we present the exposed API of G2Q, which enables Haskell solver-aided programming. We begin with a description of the core of G2Q: the `g2` quasiquoter. Using this quasiquoter, programmers can write Haskell predicates over symbolic (unknown) variables and automatically find concrete values that satisfy the predicate. We then briefly discuss the `G2Rep` typeclass, which is required to lift values to and from the quasiquoter. Finally, we discuss the strictness and fairness guarantees offered by our library.

$$
\begin{array}{lll}
QQ & ::= \lambda\ x_1\ \ldots\ x_m\ \rightarrow\ s_1\ \ldots\ s_n \mid \mathrm{e} & \\
x & ::= (\mathrm{y}\ ::\ \tau) & \text{concrete argument} \\
s & ::= ?(\mathrm{y}\ ::\ \tau) & \text{symbolic variable}
\end{array}
$$

Figure 5. The grammar accepted by G2Q. y and τ represent standard Haskell variables and types, respectively. e represents a standard Haskell expression, which must be of type `Bool`.

3.1 The `g2` Quasiquoter

The principal feature of G2Q is the `g2` quasiquoter which, as shown in the grammar in Figure 5, uses a slightly edited version of standard Haskell syntax: concrete arguments $x_1 :: \tau_1 \ldots x_m :: \tau_m$ are bound by a lambda expression; symbolic variables $s_1 :: \tau_1^s \ldots s_n :: \tau_n^s$ are then specified. Finally, a predicate e is written over the full set of variables.

The quasiquoter generates a function of type:

$$\tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \texttt{IO}\,(\texttt{Maybe}\,(\tau_1^s\ \ldots\ \tau_n^s))$$

At runtime, this function sets the concrete arguments in the predicate e to the values passed by the user. Next, the backend attempts to find satisfying instantiations of $s_1 \ldots s_n$. If it succeeds, `Just` a tuple of the found values is returned. Otherwise, `Nothing` is returned. Note that there is no guarantee that the backend is deterministic, and as such, the value is returned in the `IO` Monad.

3.2 The `G2Rep` Typeclass

The types of all concrete arguments and symbolic inputs in a `g2` quasiquoter are required to be instances of the `G2Rep` typeclass which we further describe in Section 4.2. This `G2Rep` typeclass is defined by G2Q, to allow lifting instances to and from the representation required by the `g2` quasiquoter. Defining an instance of `G2Rep` manually requires knowledge of the internals of G2Q. To allow programmers to easily use their own first-order datatypes with the `g2` quasiquoter, we provide `derivingG2Rep`, a TemplateHaskell function to automatically derive instances of `G2Rep` with a single line of code. `derivingG2Rep` requires only an instance of the Data typeclass (which is also derivable, using the DeriveDataTypeable language extension), and for the ScopedTypeVariables language extension [25] to be turned on (for reasons discussed in Section 4.2).

3.3 Strictness and Fairness

Here, we discuss the strictness and fairness properties of the `g2` quasiquoter.

(a)
```
[g2| \(xs :: [Int]) ->
        ?(x :: Int) | x == head xs |] [1..]
```
Out: `Just 1`

(b)
```
[g2| \(xs :: [Int]) (t :: Int) ->
      ?(ys :: [Int])
      | ys == take t xs |] [1..] 4
```
Out: `Just [1, 2, 3, 4]`

(c)
```
[g2| \(xs :: [Int]) -> ?(y :: Int)
        | head xs > y && y > 0 |] [0..]
```
Out: `Nothing`

Figure 6. Here, we show several examples of G2Q's behavior on infinite lists, for which the quasiquoter will give output. The key requirement is that the predicate require evaluation of only a finite amount of the infinite input.

(a)
```
[g2| \(xs :: [Int]) ->
        ?(ys :: [Int]) | xs == ys |] [0..]
```
Out: ⊥

(b)
```
[g2| \(xs :: [Int]) -> ?(y :: Int)
      | all (\x -> y > x) xs |] [0..]
```
Out: ⊥

Figure 7. Here, we show two examples of G2Q's behavior on infinite lists, that will result in non-termination.

```
data InfList a = InfCons a (InfList a)
                 deriving Data

headInf :: InfList a -> a
headInf (InfCons x _) = x
```

(a)
```
[g2| \(t :: Int) -> ?(ys :: InfList Int)
      | headInf ys == t |] 1
```
Out: `Just (InfCons 1 (InfCons 1 (InfCons 2 (...))))`

(b)
```
[g2| \(x :: Int) -> ?(ys :: InfList Int)
      | allInf (> x) ys |] 0
```
Out: ⊥

Figure 8. Here, we show two examples of `g2` quasiquoter's, with an output type that is an infinite data structure. When only a finite amount of the output has to be evaluated to check the predicate, the quasiquoter can return such an infinite data structure. However, trying to satisfy a predicate that requires evaluating an infinite amount of the infinite data structure results in non-termination.

3.3.1 Strictness

Strictness refers to the reduction order of an expression during program execution. The `g2` quasiquoter preserves Haskell's lazy evaluation semantics [24].

```
mult :: Int -> Int -> Int
mult n x
    | n == 0 = 0
    | n >= 0 = x + mult (n - 1) x
    | otherwise = mult (n + 1) x - x

[g2| \(x :: Int) ->
     ?(n :: Int) | mult n x == 10 |] 3
```
Out: ⊥

Figure 9. A quasiquoter that will fail to terminate, because of an unsatisfiable predicate involving a recursive function.

Infinite Data Structures As may be expected, lazy evaluation allows the `g2` quasiquoter to both consume and produce infinite data structures. When consuming infinite data structures, the quasiquoter must be able to fully evaluate the predicate after evaluating only a finite portion of the structure. Figure 6 shows several examples where a quasiquoter can terminate on infinite input, because finding a correct output requires only a finite portion of the input. Figure 7 shows two examples that will not terminate because there is no bound on the amount of the input that must be evaluated.

Finally, Figure 8 shows a quasiquoter that produces an infinite data structure. Similarly to the input, such quasiquotes return if and only if checking the correctness of the predicate requires evaluating only a finite amount of the output.

Recursive Functions G2Q allows arbitrary Haskell code, including the use of recursive functions. While this can be quite powerful, it also means care must be exercised if the input to a recursive function is symbolic. When executed in a `g2` quasiquoter on symbolic values, recursive unrollings of functions can lead to non-termination even if the function is guaranteed to terminate when normally executed.

To see why, consider the code in Figure 9. `mult` is simply an implementation of multiplication based on repeated addition, and will always terminate. However, the `g2` quasiquoter is searching for an integer `n` such that `mult n 3 == 10`. Of course no such integer exists, so the predicate is unsatisfiable. However, the recursive search over the symbolic variable will result in a deeper and deeper search to find such an `n`, resulting in non-termination.

It follows from the halting problem that any automatic approach to prevent this kind of error would, unfortunately, rule out at least some good programs. Given that Haskell itself does not prove termination, we therefore leave it up to programmers to ensure their `g2` quasiquoters terminate. To prevent non-termination, it is sufficient to ensure that either (1) no recursive function call's termination depends on a symbolic variable, or (2) whenever a recursive function call depends on a symbolic variable, the predicate is satisfiable.

3.3.2 Fairness

We offer two *fairness guarantees* to users of G2Q. Here, we present the minimal information needed for users of G2Q. In Section 4.4, we will revist these guarantees, and provide justifications. Both are relative to the completeness of the underlying SMT solver. That is, they are true to the extent that G2's underlying SMT solver is able to answer every query correctly:

1. In a predicate with no recursive function calls or let bindings, if the predicate is unsatisfiable we will eventually return `Nothing`.
2. If the `g2` quasiquoter's predicate will evaluate to `True` given some instantiation of the symbolic variables, G2Q will eventually return a solution.

In Section 4.4, we will return to and justify these fairness guarantees.

4 Solver-Aided Backend

The backend of G2Q relies on an existing Haskell symbolic execution engine, G2 [20]. We give a brief overview of symbolic execution and G2. Then, we elaborate on the design of each of G2Q's components. Finally, we state some limitations of our approach and how they may be alleviated in the future.

4.1 Symbolic Execution

Symbolic execution [10] is a technique that executes a program with *symbolic* variables constrained by logical formulas, in place of *concrete* values. This abstraction allows the symbolic execution engine to explore *all* paths of a conditional in a single execution, and using this, yield logical descriptions of what constraints on the symbolic variables are needed to execute each path.

4.1.1 Symbolic Execution with G2

G2 extracts Core Haskell from GHC before further translating Core Haskell into a custom intermediate language. Symbolic execution in G2 works over a set of *states*, where each state is a tuple (E, H, P). Here E is the expression under evaluation, H is a heap that maps variables to other expressions, and P is a set of formulas that tracks constraints on the symbolic variables.

While performing symbolic execution, G2 tracks multiple states at once. For each *step* of symbolic execution, one application of Haskell-like reduction rules [24] are applied to a state. These rules are augmented to handle symbolic variables such as *splitting* (or *branching*) for case statements and adding to the path constraint P when necessary. At the end of executing a state, P can be solved using a SMT solver to determine concrete inputs that traverse the same execution path as the state. Symbolic execution finishes when some stopping criteria is met: in the case of G2Q, when a state satisfying a quasiquoter's predicate is found, or all states have been exhaustively searched.

4.1.2 Symbolic Execution Example

To better illustrate symbolic execution with G2, consider a lookup function for associative lists as shown in Figure 11. Using symbolic execution, we can gradually explore the possible outputs of `kvLookup` in terms of formulaic constraints on the input values, as shown in Figure 10.

State 0 We begin symbolic execution on `kvLookup` by calling it with the symbolic variables `myKey` and `myAssocs`. Here, symbolic variables differ from concrete variables in the sense that they receive special mappings in the heap. At present, no path constraints are specified on the symbolic variables.

State 1 A one-step application of the execution semantics creates the appropriate variable bindings in the heap and our current expression is now a case statement. In our intermediate language (and also Core Haskell), all conditional branching is eventually reduced to case statements, and for G2 this is where *branching* occurs: from State 1 we create State 2 and State 3.

State 2 This is the state that occurs should we decide to satisfy the first branch of the case statement. Here, the variable `kvs` (which maps to the symbolic variable `myAssocs` in H) is required to be an empty list. Since `Nothing` cannot be reduced anymore, we are done with execution. The resulting path constraint can be passed off to a SMT solver to produce solutions for the values of `myKey` and `myAssocs` – the former can be any `Int`, and the latter must be an empty list.

State 3 Should we choose the other branch of the case statement, we introduce two additional symbolic bindings for `k` and `v`. Additionally, the path constraint now includes the condition that `kvs` must have a head value of `(k,v)` and a tail value of a symbolic list called `rest`.

State 4 Here if we take positive branch for `key == k`, then G2 finishes symbolic execution since `Just v` is evaluated to its constructor (which all that is required, under Haskell's semantics [24]). Passing this set of path constraints to a SMT solver will reveal that `myAssocs` must be a list of pairs of at least length 1, and that the first pair must have a key value equal to `myKey`.

State 5 Taking the negative branch for `key == k` leads to a recursive call on `kvLookup`. Similar to State 4, the list `myAssocs` must still be non-empty except that the first pair must have a key value that is **not** equal to `myKey`.

State 6 Another recursive call begins. We must ensure that the variables are appropriately bounded in the heap. Since `kvLookup` has recursive calls that are guaraded by symbolic variables, G2 will continue symbolic execution indefinitely to enumerate all the possible return results.

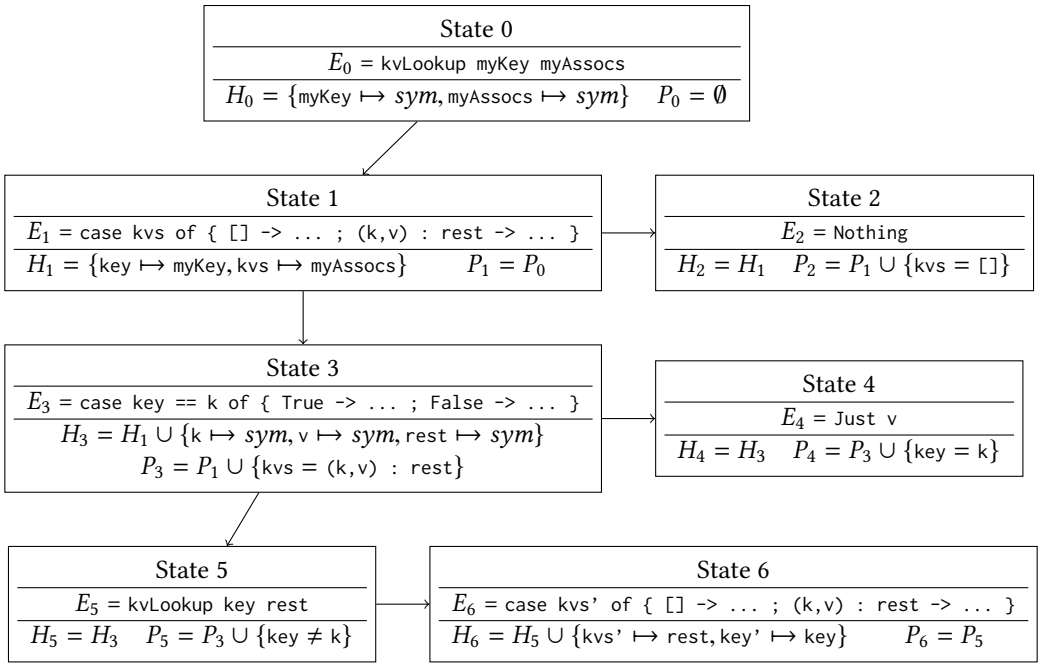

Figure 10. Symbolic execution example for Figure 11

```
kvLookup :: Int -> [(Int, a)] -> Maybe a
kvLookup key kvs =
  case kvs of
    [] -> Nothing
    (k, v) : rest ->
      case key == k of
        True -> Just v
        False -> kvLookup key rest
```

Figure 11. A lookup function for associative lists

```
class G2Rep g where
    g2Rep :: g -> G2Expr
    g2UnRep :: G2Expr -> g
    g2Type :: g -> G2Type
```

Figure 12. The `G2Rep` typeclass, which converts values to and from G2's representation.

As a quick summary to highlight important points: states 1 and 3 are states in which new states are formed via *branching*. State 5 is an example of a recursive call on `kvLookup` which leads to non-termination of symbolic execution unless otherwise bounded by means such as step limits or timeouts.

4.2 The `G2Rep` Typeclass

A slightly simplified version of the `G2Rep` typeclass is shown in Figure 12 (some types from G2 that do mundane mapping have been hidden, to simplify the presentation.) It includes three functions: `g2Rep`, `g2UnRep`, and `g2Type`. The `g2Rep` and `g2UnRep` functions map from real Haskell values to G2's representations of those values and back. `g2Type` is a helper function for `g2Rep` and `g2UnRep`: polymorphic type arguments are explicitly represented in G2's core language, and so we require `g2Type` to give us access to the G2 representation of the type of polymorphic arguments.

```
class G2Rep a => G2Rep [a] where
  g2Rep [] = g2Nil (g2Type (undefined :: a))
  g2Rep (x:xs) =
    g2Cons (g2Type (undefined :: a))
                   (g2Rep x) (g2Rep xs)
  ...
```

Figure 13. The `g2Rep` definition for lists. We denote the standard Haskell list constructors as `:` and `[]`, and G2's representation of a list as `g2Cons` and `g2Nil`, respectively.

Figure 13 shows part of the definition of `G2Rep` for lists. For the most part, the mapping is very routine – the sole point of interest is the use of `g2Type`. Sometimes (as in the expression for a nil list constructor) we require calling the instance of `g2Type` for a type of which we do not have a value. Fortunately, we can insist that `undefined` is a value of any type and use it to call the appropriate `g2Type`. This does require the ScopedTypeVariables language extension to be enabled, so that the type variable in the instance declaration and the type variable in the function body are bound to the same type. If a programmer tries to deriving `G2Rep` without enabling the extension, we show a error message, reminding them to turn it on.

4.3 `g2` Quasiquoter Compilation

We now describe the translation of a `g2` quasiquoter into a Template Haskell expression.

Imported Modules G2Q allows making use of functions from imported modules, as long as the source code is available for G2 to compile into its internal representation. We use Template Haskell to pull the list of imported modules from the current file and use a cabal file [23] to search for them. We also utilize a custom version of the standard Haskell Base and Prelude [34], which supports many of the commonly used types and functions (and which we are working on expanding).

Parsing G2Q accepts a lightly modified version of traditional Haskell syntax. The only differences are:

1. We require type annotations to be given in the concrete variable lambda binding.
2. We introduce a new notation to specify symbolic variables.

Our parser extracts the concrete variables along with their types `x1 :: t1 ... xm :: tm`, the symbolic variables and types `s1 :: ts1 ... sn :: tsn`, and the predicate expression `e` from the `g2` quasiquote. Then, it rewrites the user's query as a Haskell predicate function with both the concrete and symbolic variables bound by lambda expressions, in addition to an explicit type signature:

```
pred :: t1 -> ... tm -> ...
            ts1 -> ... tsn -> Bool
pred x1 ... xm s1 ... sn = e
```

We then use G2's existing parser (which itself makes use of GHC's parser) to translate this traditional Haskell code into G2's internal representation.

State Construction We construct a state $s = (E, H, P)$. H is a heap containing functions from imported modules, and P is initialized to empty. We initalize E to:

```
let r = pred x1 ... xm s1 ... sn
in assume r (s1, ..., sn)
```

`x1` to `xm` are placeholders for the concrete arguments (which will be replaced in the next step), `assume p e` assumes the predicate `p` holds and then returns `e`. Thus, the code in E will force the quasiquoter's predicate to hold, and if it does, return a tuple of the values of the symbolic variables.

Argument Bindings For each concrete argument `x1 ... xm` we construct a Template Haskell expression that will bind `g2_xi` to `g2Rep xi` at runtime. Then, we construct a runtime call to a function named `floodConsts`, which receives the state `s = (e, h, p)` and a list of the `g2_xi` as arguments. `floodConsts` lazily (so as to not force too much of the input values) replaces each concrete argument in the states expression `e` with the `g_xi` from the list.

Solving Symbolic Variables Given a state with the concrete arguments filled in via `floodConsts`, G2 is able to symbolically execute the state (as discussed in Section 4.1) at runtime. Assuming it terminates (as discussed in Section 3.3.1), G2's constraint solving finds concrete values for the symbolic variables and returns a tuple of the found solutions. We then use `g2UnRep` to lazily translate the tuple from G2's representation into regular Haskell values. `g2UnRep`'s laziness allows returning infinite data structures when needed, as discussed in Section 3.3.1.

Type Safety We use Template Haskell to, at compile time, wrap each input to `g2Rep` and call to `g2UnRep` with an explicit type annotation. These annotations provide programmers with type errors if they try to mistype an argument or the returned value.

4.4 Fairness and Heuristic Search

As discussed in Section 3.3.2, G2Q offers two fairness guarantees. We justify theses fairness guarantees here and then discuss some heuristics we implement as well as how those heuristics preserve the fairness guarantees.

4.4.1 Fairness Guarantees

We begin by presenting and justifying our fairness guarantees. Both guarantees are relative to the completeness of the underlying constraint solver.

Guarantee 1 First, we guarantee that, if the predicate in the quasiquoter contains no recursive function calls or recursive let bindings, and is unsatisfiable, the quasiquoter will eventually terminate by returning `Nothing`. This can be trivially seen from an examination of the reduction rules used by G2, as shown in [20]. The only possible source of an infinite loop is a recursive call, since all other reduction rules reduce the size of the expression being evaluated and thus will lead to termination. Therefore, in the absence of a recursive function call or recursive let binding we can fully explore the set of possible states and return `Nothing` if all are unsatisfiable.

Guarantee 2 The second guarantee is that, if there is some instantiation of the symbolic variables such that the predicate in the quasiquoter q will evaluate to true, some solution will eventually be returned.

To ensure this, it is sufficient to show that:

1. Whenever we hit a branch in the code, we create states to explore along each possible branch.
2. If some state exists, and it has not yet fully executed to `True` or `False`, either the state will eventually be executed, or some other state will be executed that returns a solution.

(1) can be seen from an examination of the reduction rules, in [20]. The sources of branching are limited, all case expression branches are initialized as separate states.

(2) is trickier, as it requires us to fairly evaluate all states. Otherwise, we might only evaluate some subset of states where the predicate is false and miss some state where the predicate is true. To ensure that (2) holds, we fix a predicate p that we know any state we execute will eventually violate (an example of such a guarantee is given below, in Section 4.4.2.) We then store the states in a queue.

During symbolic execution, we pop the state at the head of the queue. We symbolically execute this state either until we discover that it satisfies the quasiquoter's predicate q or until it has violated the predicate p. If the state is not yet fully evaluated but p becomes false, we re-insert the state at the end of the queue. If the state splits at a branch, we arbitrarily choose one of the states to continue executing. The others get inserted at the end of the queue.

With this scheme we can see via a classic argument that all states not yet fully evaluated will eventually be executed (unless another state that satisfies q is found first). At any point such a state s is in the queue. Suppose there are $s\#$ states ahead of the s in the queue. Since we (at least temporarily) halt execution of every state eventually, either one of those $s\#$ states will turn out to be a solution or s will eventually be executed.

4.4.2 Symbolic Execution Heuristics

A challenge in making symbolic execution effective for finding solutions are the heuristics employed. In particular, because symbolic execution tracks multiple states at once, yet (outside of parallelization) only one state is executed at a time, the *order* in which states are chosen for reduction is crucial. A bad ordering causes an increase in the time required to find a satisfying solution to the predicate.

G2Q employs a heuristic that prioritizes states with fewer symbolic variables. The intuition is that such states will (often) lead to fewer new path constraints – resulting in cheaper calls to the SMT solver, and reduced future state splitting.

Preserving Fairness As described in the previous section, our fairness guarantee depends on a queue, with some predicate p that will eventually be violated. To ensure that we violate the p with this heuristic, we choose p to be that the state (1) contains fewer symbolic variables than the state at the head of the queue, and (2) has increased its symbolic variable count in the last k steps (for some fixed k).

If condition (2) is consistently met, then eventually condition (1) will be violated, and the state will be sent to the back of the queue. Otherwise, condition (2) will send the state to the back of the queue. Thus our predicate is sufficient to guarantee fairness.

4.5 Limitations

Scoping and Module Imports A `g2` quasiquoter requires that all functions and datatypes used in it are defined in an imported module, and *cannot* use functions or datatypes declared in the same module. This is because in order to perform symbolic execution, G2 has to be able to compile the code in the quasiquoter – and the code's dependencies – to G2's internal representation. Trying to compile the module the quasiquoter is in would result in an infinite loop.

Argument Types Within a quasiquoter, G2Q requires that type signatures be provided (rather than inferred) for the concrete and symbolic arguments to the `g2` quasiquoter. In addition, we require that such types be monomorphic and are also first-order. It should be stressed that these restrictions apply only to the quasiquoters *arguments*. Within the rest of the predicate's code, polymorphic and higher order functions may be used.

The source of each of these limitations is in fact the same. Currently, G2Q relies on having access to all the code used in `g2` quasiquoter available at compile time, so that it can compile the code into G2's intermediate representation. If a programmer tries to use some library that G2Q cannot access the code for, an error is given at compile time. Allowing passing arbitrary higher order functions would require G2Q to have some way of dynamically converting the passed function to G2's representation at runtime. Typeclasses are, in the internals of both GHC and G2, just a dictionary of higher order functions [19] and thus present the same issue.

To simplify for end users and hopefully prevent confusion, we disallow polymorphism entirely since we cannot support typeclasses. It is possible that this decision is overly conservative, and if that proves to be the case, we could relax the restriction to allow limited polymorphism in the future. However, we want to get further experience using G2Q before making this decision.

Base Support In order to make use of datatypes and functions from Base, G2Q requires them to be compiled into our intermediate language. As compiling Base is a complicated process which relies closely on GHC, G2 currently make use of a custom Base library. As such, G2, and by extension G2Q, currently supports only a subset of functions and datatypes.

In the future, we plan on expanding this subset. In addition, we plan to investigate means by which we could compile the whole of Base (such as instrumenting GHC to write out our intermediate language, for example).

5 Evaluation and Case Studies

We have made G2Q available on Hackage at http://hackage.haskell.org/package/g2q. Additionally, G2 is available on Hackage at http://hackage.haskell.org/package/g2. Here, we present case studies and an evaluation, showing how G2Q can be used to solve a variety of problems.

```
type Ident = Int

data Expr = Var Ident
          | Lam Expr
          | App Expr Expr
          deriving (Show, Read, Eq, Data)

$(derivingG2Rep ''Expr)

type Stack = [Expr]

eval :: Expr -> Expr
eval = eval' []

eval' :: Stack -> Expr -> Expr
eval' (e:stck) (Lam e') =
        eval' stck (rep 1 e e')
eval' stck (App e1 e2) = eval' (e2:stck) e1
eval' stck e = app $ e:stck

rep :: Int -> Expr -> Expr -> Expr
rep i e v@(Var j)
    | i == j = e
    | otherwise = v
rep i e (Lam e') = Lam (rep (i + 1) e e')
rep i e (App e1 e2) =
        App (rep i e e1) (rep i e e2)

app :: [Expr] -> Expr
app = foldl1 App

num :: Int -> Expr
num n = Lam $ Lam $
    foldr1 App (replicate n (Var 2) ++ [Var 1])
```

Figure 14. De Bruijn index based lambda calculus and evaluator.

5.1 Programming by Example in Lambda Calculus

Programming by example is a paradigm that allows code to be synthesized from input-output examples.

Consider a lambda calculus language and evaluator based on De Bruijn indexing [11] as shown in Figure 14. De Bruijn indexing eliminates variable names, by writing bound variables as an integer that indicates the number of lambdas between the variable and its binder.

The evaluator function `eval` for the lambda calculus is standard: it simplifies a lambda expression as much as possible and then outputs it to the user.

Inspired by the programming by example paradigm [26, 30], one can use G2Q to not only *execute* lambda calculus expressions but to also *synthesize* expressions based on input-output examples.

To do this, we may write a function as follows:

```
solveDeBruijn :: [([Expr], Expr)]
  -> IO (Maybe Expr)
solveDeBruijn =
  [g2| \(es :: [([Expr], Expr)])
          -> ?(func :: Expr) |
      all (\e -> (eval (app (func : fst e)))
                  == snd e) es |]
```

The `solveDeBruijn` function takes an input list of pairs of the form `(arguments, result)`. The goal is to then synthesize a new function `func` such that when all the `arguments` are applied to `func`, the result yielded is `result`.

As a simple example consider the Haskell `const` function, which takes two arguments and returns the first unmodified. By writing the function call:

```
solveDeBruijn [ ([num 1, num 2], num 1)
              , ([num 2, num 3], num 2)]
```

we can synthesize a lambda expression with this effect:

```
Lam (Lam (Var 2))
```

Somewhat less trivially, we can use the Church encoding of Booleans [6] to synthesize Boolean functions. In Church encoding we denote `True` as `Lam (Lam (Var 2))` and `False` as `Lam (Lam (Var 1))`.

Using these definitions, we can write examples for Boolean functions, such as or:

```
solveDeBruijn [ ([trueLam, trueLam], trueLam)
              , ([falseLam, falseLam], falseLam)
              , ([falseLam, trueLam], trueLam)
              , ([trueLam, falseLam], trueLam) ]
```

and synthesize correct definitions for those functions:

```
Lam (App (Var 1) (Var 1))
```

5.2 n-Queens

A mathematical puzzle called the n-queens problem asks how n queen pieces may be placed on an $n \times n$ chess board such that no two queens threaten each other [36]. That is, no two queens may be in the same row, column, or diagonal. We demonstrate how this problem may be solved with G2Q via an encoding in Figure 15.

Since no two queens may be in the same row and we have n queens to be placed in n rows, there is clearly a queen in every row. Thus, we represent the set of queens by a list of `Int`'s, where the Int j in the ith position in the list, indicates there is a queen at (i, j). `allQueensSafe` checks if the given list of `Queen`s is a valid solution to the n-queens problem. Specifically, it checks if the list is the correct length, that all the queens on in legal positions, and that none of the queens can attack each other.

A classic version of this problem is for a traditional chessboard with $n = 8$. The solution that G2Q produces for the 8-queens problem is shown in Figure 16.

```
type Queen = Int

indexPairs :: Int -> [(Int,Int)]
indexPairs n =
  [(i, j) | i <- [0..n-1], j <- [i+1..n-1]]

legal :: Int -> Queen -> Bool
legal n qs = 1 <= qs && qs <= n

queenPairSafe :: Int -> [Queen]
              -> (Int, Int) -> Bool
queenPairSafe n qs (i, j) =
  let qs_i = qs !! i
      qs_j = qs !! j
  in (qs_i /= qs_j)
      && qs_j - qs_i /= j - i
      && qs_j - qs_i /= i - j

allQueensSafe :: Int -> [Queen] -> Bool
allQueensSafe n qs =
  (n == length qs)
  && all (legal n) qs
  && (all (queenPairSafe n qs) (indexPairs n))
```

Figure 15. N-Queens

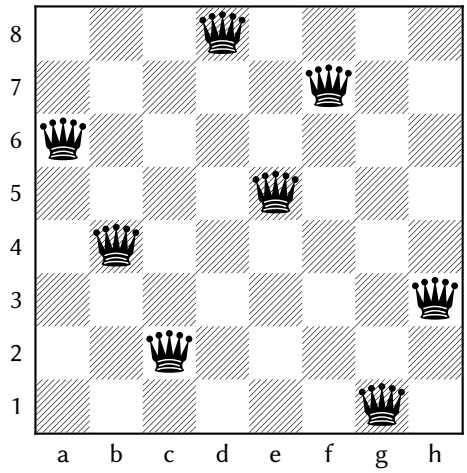

Figure 16. A solution to 8-queens produced by G2Q

5.3 Regular Expressions

Borrowing an example from SmtEn [41], suppose a user has written a regular expression implementation as a domain specific language. The implementation includes a `match` function which, given a regular expression and a string, returns whether the string matches the regular expression.

Provided a regular expression written in such a DSL, it may be helpful to search for examples of strings that are accepted by this regular expression. Such a problem is examined in SmtEn, but requires the user to also implement several functions to assist the search. We demonstrate how G2Q and `match` can be used to solve this problem, with very little additional code required from the user. We defer further discussion of SmtEn to Section 6.

First, we augment the `RegEx` algebraic data type with a `Data` derivation, as well as a `derivingG2Rep`. Other parts of the code may be left untouched.

```
data RegEx =
    Empty -- The empty language
  | Epsilon -- The empty string
  | Atom Char | Star RegEx
  | Concat RegEx RegEx | Or RegEx RegEx
  deriving (Show, Eq, Data)
$(derivingG2Rep ''RegEx)

match :: RegEx -> String -> Bool
match = ...
```

SmtEn's implementation of regular expressions includes both `Epsilon` to denote the empty string, as well as `Empty` for the empty language. Next, we encode a query into the `g2` quasiquoter

```
stringSearch :: RegEx -> IO (Maybe String)
stringSearch =
  [g2| \(r :: RegEx) -> ?(str :: String) |
          match r str |]
```

Finally, we can call this search function with a regular expression:

```
-- (a + b)* c (d + (e f)*)
stringSearch $
  Concat (Star (Or (Atom 'a') (Atom 'b')))
    (Concat (Atom 'c')
       (Or (Atom 'd')
           (Concat (Atom 'e') (Atom 'f'))))
```

G2Q is able to successfully find a string – `Just "cd"` – matched by this regular expression.

5.4 Evaluation

While writing the case studies we discovered two key factors that affect performance: predicate order in the quasiquoter and state explosion. Here we discuss both these factors and then address the runtimes of our case studies.

Predicate Order Simple changes to the predicate can have a dramatic affect on runtime. For example, consider `allQueensSafe'` in Figure 18. This function is the same as `allQueensSafe` from Figure 15, except that the constraint on the length of the list has been moved from being the first conjoined constraint to the last. However, solving the 8-queens problem with `allQueensSafe` takes only 2.30 seconds while solving it with `allQueensSafe'` takes 36.42 seconds. This is because constraining the length of the list allows quickly filtering out many states where the list is either too long or too short. In future work it would be valuable to explore

Task	Time (secs)
`badEnvSearch prog`	59.32
Search for non-zero `Mul x y == Add x y`	0.56
`solveDeBruijn` for id	0.04
`solveDeBruijn` for `const`	1.06
`solveDeBruijn` for `NOT`	Timeout
`solveDeBruijn` for `OR`	86.22
`solveDeBruijn` for `AND`	Timeout
`solveQueens 4`	0.36
`solveQueens 5`	0.55
`solveQueens 6`	0.90
`solveQueens 7`	1.47
`solveQueens 8`	2.30
`stringSearch` for $(a+b)^*c(d+(ef)^*)$	0.26
`stringSearch` for $abcdef$	4.23
`stringSearch` for $a+b+c+d+e+f$	0.05
`stringSearch` for $a^*b^*c^*d^*e^*f^*$	0.02

Figure 17. Case study running times.

```
allQueensSafe' :: Int -> [Queen] -> Bool
allQueensSafe' n qs =
   all (legal n) qs
  && (all (queenPairSafe n qs) (indexPairs n))
  && (n == length qs)
```

Figure 18. `allQueensSafe'` is the same as `allQueensSafe`, from Figure 15, except the constraint on the list length is moved to the end of the function. This change has a dramatic affect on running time. Solving 8-queens with `allQueensSafe` takes only 2.30 seconds, while using `allQueensSafe'` requires 36.42 seconds.

ways of automatically reordering predicates to optimize performance.

State Explosion Programs with large amount of branch can cause symbolic execution to suffer from *state explosion*, in which the number of states the symbolic execution engine generates and must evaluate grows exponentially. In particular, G2's current handling of symbolic algebraic datatypes can cause it to branch into many states. In future work we hope to implement state merging in order to reduce the number of states that must be individually evaluated.

Evaluation Results Figure 17 shows evaluation results from our case studies and some other associated benchmarks. We ran all tests with a timeout of two minutes. Two of the tests did not terminate in this time. These timeouts are largely due to state explosion from G2's handling of algebraic datatypes. In future work we hope to improve there performance by implementing state merging.

Despite the timeouts, we view these results as very positive. All our benchmarks involve programs with recursion and yet G2Q manages to solve eight of the benchmarks in under a second. In contrast, unassisted SMT solvers are known to struggle with recursion or loops [22]. Thus, even with the timeouts, our results indicate an improvement over direct encoding of SMT formulas.

6 Related Work

Here we give an overview of related work, with a particular focus on work that aims to simplify the use of SMT solving in high level languages.

Solver-Aided Languages Like G2Q, SmtEn [41] is designed to ease using SMT solvers in Haskell. However, the interfaces provided by the two tools are quite different. SmtEn provides users with functions to build up a Space (that is, a set) and then use a SMT solver to search through the Space, for a value that satisfies some condition. In contrast, with our tool, the quasiquoter can be called directly, with no need to provide a set of possible instantiations.

In fact, both SmtEn and G2Q's APIs have advantages. SmtEn's treatment of Spaces as first class values allows them to be passed around and manipulated in code before being queried. On the other hand, as treated in SmtEn, a Space either has to be built from other Spaces, or constructed from scratch as a singleton. As such, constructing a Space results in a great deal of often tedious code, which can be avoided by our approach.

We see great potential in combining the approach of Spaces, (or some close equivalent) and our quasiquoter approach. For example, one could imagine a hybrid approach that uses a `g2`-like quasiquoter to construct Space-like values. We leave such considerations to future work.

Curry [21] is a logic driven functional programming language. Curry may be seen as an approach to design a functional language around logic programming. In contrast, G2Q is an attempt to fit constraint solving into an existing functional language. As such, G2Q's semantics (that is, really Haskell's semantics) are likely more comfortable for existing Haskell programmers.

Rosette [39, 40] is an extension of Racket, which allows for constraint based programming. Unlike G2Q, Rosette requires that all constraint generation be self-finitizing; that is, all constraint generation must terminating. The trade-off here is that, Rosette, unlike G2Q, offers guaranteed termination. However, this also means that Rosette rules out some valid programs. Rosette supports only symbolic integers and booleans, while G2Q supports lifting any first-order value to a symbolic value (given an instance of `G2Rep`).

Like Rosette, Kaplan [28] allows for constraint based programming, although in Scala rather than in Racket. Like G2Q, Kaplan supports a variety of types, including algebraic datatypes. Due to Racket and Scala both being strict languages, however, neither Rosette nor Kaplan account for non-strict execution.

Compile-time Theorem Proving HALO [45] and [46] aim to translate Haskell programs into first-order logic in order to apply contract verification to Haskell programs. G2Q, on the other hand, is aimed at supporting runtime constraint solving in Haskell while these tools instead focus on ensuring that Haskell programs satisfy contract specifications.

Constraint-Based Synthesis Complete Functional Synthesis [29] also describes a technique to write programs by writing constraints. Unlike the other discussed work, it relies on synthesis of code at compile time, rather than constraint solving at runtime. This presents a trade-off: the code it synthesizes is more efficient, but the logic it can reason about is more restricted. For example, Complete Functional Synthesis does not support recursive functions or algebraic datatypes.

SMT APIs A number of SMT solvers, including Z3 [12], CVC4 [7], and Yices [15], have API interface for a variety of languages. Depending on the language the API is intended for, some of these offer strong type guarantees. Relatedly, there are Haskell packages [3, 18] and packages for other languages [4, 27] that expose strongly typed bindings to SMT solvers. However, these sorts of API interfaces are all very close to the abstraction level of the SMT solver, as they directly expose SMT constructs. In addition, these strongly typed API interfaces still require a great deal of manual work related to duplicating and copying data.

7 Conclusion

We present G2Q, a quasiquoter for Haskell to ease access to constraint solving. By leveraging the G2 symbolic execution engine, G2Q allows users to easily encode constraints with minimal engineering overhead, and a higher level of abstraction than with tools like SMT solvers.

Acknowledgments

We thank the anonymous reviewers for their feedback on this paper. This work was supported by the the National Science Foundation under Grant Numbers CCF-1553168 and CCF-1302327.

References

[1] [n. d.]. Difference in behavior between ‘declare-const + assert‘ and ‘define-fun‘. https://github.com/Z3Prover/z3/issues/2139. Accessed: 2019-10-05.

[2] [n. d.]. Strange performance for (= x y) vs. (= y x). https://github.com/Z3Prover/z3/issues/1822. Accessed: 2019-10-05.

[3] Iago Abal. [n. d.]. z3 (Hackage Package). http://hackage.haskell.org/package/z3

[4] Siddharth Agarwal. 2012. Functional SMT solving: A new interface for programmers. *Master's thesis, Indian Institute of Technology Kanpur* (2012).

[5] Markus Aronsson and Mary Sheeran. 2017. Hardware Software Co-design in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 162–173. https://doi.org/10.1145/3122955.3122970

[6] Henk P Barendregt. 1992. Lambda calculi with types. (1992).

[7] 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.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. The SMT-LIB Standard 2.6. 103. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[10] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[11] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.

[12] 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

[13] Iavor S. Diatchki. 2015. Improving Haskell Types with SMT. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2804302.2804307

[14] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 181–192.

[15] Bruno Dutertre and Leonardo De Moura. 2006. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2, 2 (2006), 1–2.

[16] Anton Ekblad. 2017. A meta-EDSL for Distributed Web Applications. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 75–85. https://doi.org/10.1145/3122955.3122969

[17] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt Free Ivory. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/2804302.2804318

[18] Levent Erkök. [n. d.]. SBV: SMT based verification in Haskell. http://hackage.haskell.org/package/sbv

[19] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996), 109–138. https://doi.org/10.1145/227699.227700

[20] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 411–424. https://doi.org/10.1145/3314221.3314618

[21] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. 1995. Curry: A truly functional logic language. In *Proc. ILPS*, Vol. 95. 95–107.

[22] 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 (SOSP '15)*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

[23] Isaac Jones. 2005. The Haskell Cabal, a common architecture for building applications and libraries. (2005).

[24] 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.

[25] Simon Peyton Jones and Mark Shields. 2004. Lexically-scoped type variables. (2004).

[26] Susumu Katayama. 2013. MagicHaskeller on the Web: Automated programming as a service. In *Haskell Symposium*.

[27] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2011. Scala to the Power of Z3: Integrating SMT and Programming. In *International Conference on Automated Deduction*. Springer, 400–406.

[28] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2012. Constraints As Control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 151–164. https://doi.org/10.1145/2103656.2103675

[29] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete Functional Synthesis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 316–329. https://doi.org/10.1145/1806596.1806632

[30] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

[31] Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/1291201.1291211

[32] Mladen Nikoli\'c. 2012. Statistical Methodology for Comparison of SAT Solvers. In *EMSQMS 2010. Workshop on Evaluation Methods for Solvers, and Quality Metrics for Solutions (EPiC Series in Computing)*, Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli (Eds.), Vol. 6. EasyChair, 33–38. https://doi.org/10.29007/bhvj

[33] Divesh Otwani and Richard A. Eisenberg. 2018. The Thoralf Plugin: For Your Fancy Type Needs. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 106–118. https://doi.org/10.1145/3242744.3242754

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

[35] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[36] Igor Rivin, Ilan Vardi, and Paul Zimmermann. 1994. The n-queens problem. *The American Mathematical Monthly* 101, 7 (1994), 629–639.

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

[38] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[39] 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 (Onward! 2013)*. ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[40] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[41] Richard Uhler and Nirav Dave. 2014. Smten with satisfiability-based search. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 157–176.

[42] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 132–144. https://doi.org/10.1145/3242744.3242756

[43] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with Refinement Types in the Real World. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 39–51. https://doi.org/10.1145/2633357.2633366

[44] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

[45] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: Haskell to logic through denotational semantics. In *ACM Sigplan Notices*, Vol. 48. ACM, 431–442.

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