John Bedette and Vivek Srirama
Functional Programming Winter 2024

An Examination of QuickCheck and Generators in Haskell

Introduction:

QuickCheck is a property-based testing library originally made in Haskell. A massive success, QuickCheck, initially released in 1999 by Koen Claessen and John Hughes, has since been ported to multiple programming languages with most major programming languages having a quickcheck inspired library available. QuickCheck's property-based testing generates a randomized set of inputs and puts them through a series of predefined functions and asserts that for each and any input within certain parameters, the functions should return some kind of "True" response. Whether that response is a list that is indeed sorted, or a string that excludes escape characters, the QuickCheck library can quickly generate varied inputs and check that nothing breaks.

Unit testing is often a long and difficult task and QuickCheck is a fantastic tool to examine a variety of inputs and can be modified to specifically check for edge cases. A fun fact, Claesson and Hughes cited former Portland State Professor Sergio Antoy in their first citation for their academic paper "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs" announcing their QuickCheck library.

Overview of Generator and IO Monads:

Generators are a Monad of the QuickCheck library. They generate values based on an arbitrary data type which can either be predefined or a custom data type. They contain different combinators which allow users to define how they want the generators to generate output. In this program, we define a generator for a custom Card data type (between a value of 1 to 11) to represent the different cards that exist in a deck of cards as well as a generator to generate a list of Cards. We also define a generator for a custom data type known as Action representing the different actions a player might take. The generator used to generate an action is written with the frequency combinator meaning that some values are more likely to be generated than others whereas the Card generator is written using the elements combinator meaning that every value has an equal probability of being picked. Using a generator returns an element with the type Gen a, where a is the type of element being generated. There is also a function known as generate that produces an IO of the generated data type when applied to a generator.

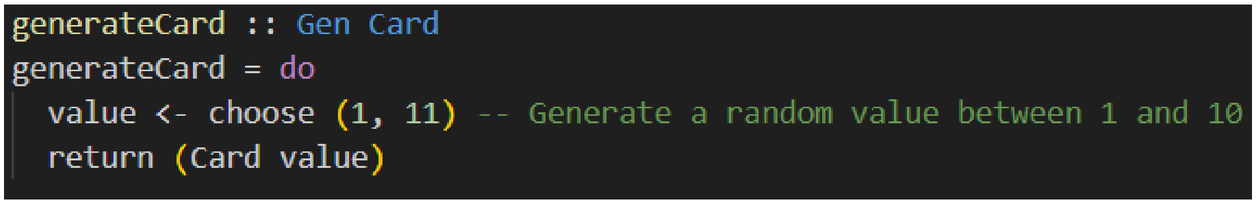
The IO monad comes with its own concerns. While the IO monad is intended to be largely inert, by using data with the IO monad attached, the program is no longer purely functional because data is coming from a place outside Haskell's purely functional environment. IO is meant to encapsulate the data as best as possible, but input and output from an external source can have side effects because the data might go beyond the realm of strict input/output. When using 'generate' to create input, the returned IO monadic data is external to the Haskell environment and violates Haskell's strictly functional paradigm. Because of this, it is best to use the QuickCheck function to provide arguments to the function in question, rather than trying to use generate to create values to use within the function. However, if you must, you may use ‘unsafePerfomIO’ to extract values from a generated IO monadic item.

The Arbitrary typeclass has two main methods: arbitrary :: Gen a and shrink :: a -> [a]. QuickCheck uses tests that are generated from the Arbitrary typeclass. This typeclass has a predefined method for data types that already exist in Haskell such as integers and booleans. However, in order to use it for custom data types, the arbitrary method must be defined by the user for that data type. The purpose of the Arbitrary type class is to provide a default implementation of a generator for a data type. This is because one could define a new generator that works differently since the default method of generation might not provide the desired behavior that the user desires. Therefore, in our program, we had to define an arbitrary method for generating a card. When using the Arbitrary typeclass, the QuickCheck method receives test cases for the type used in the arbitrary method call.

On the other hand, the shrink method takes counterexamples from a generator and tries to “shrink” it. This essentially means that it tries to create a simpler counterexample to the function being tested using quickCheck. The goal of this is to provide more well defined counterexamples to the user so that they can have a better understanding of what kinds of inputs do not match the expected behavior. If the element being “shrunk” is empty, then no further shrinking occurs since an empty element cannot be reduced further. Our program defines an instance of the Arbitrary type class but does not utilize an instance of it since our program always generates an output of the same length and is very specific. We also did not define a shrink method since we did not see how it would be useful for the context we are working with.

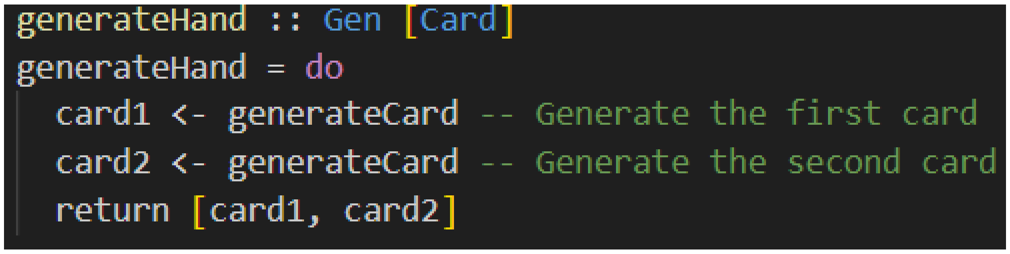
We defined two examples of generators for composite data types. The first one took two generators: a generator that generates a hand of cards and a generator that generates an integer. These two generators were combined to form a tuple with the first element as a list of cards and the second element being an integer. This represents a possible bet that a player could win with a hypothetical hand if that hand beats the dealer’s hand. The second one was a generator that took in a generator that generates a hand of cards and a generator that generates an action which was a custom data type defined by us. The action has two types: LeaveGame | ContinuePlaying. Standing alone, these represent different actions a player could take at the end of a round (i.e. they can continue playing or stop playing and leave the game). We combined these two generators to generate a tuple with the first element being a list of cards and the second element being a possible action. This was designed to represent an action that a player might take based on a hand that they got from that round.

Examples and Our Project:
Generators for Basic Types:
Card Generator



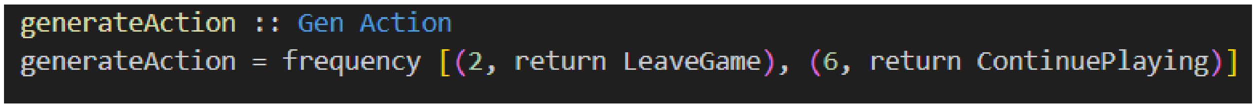
```haskell
generateCard :: Gen Card
generateCard = do
  value <- choose (1, 11) -- Generate a random value between 1 and 10
  return (Card value)
```

Hand of Cards (list of Cards) Generator:



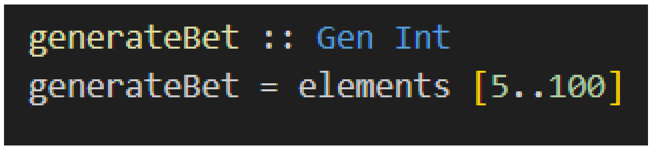
```haskell
generateHand :: Gen [Card]
generateHand = do
  card1 <- generateCard -- Generate the first card
  card2 <- generateCard -- Generate the second card
  return [card1, card2]
```

Action Generator:



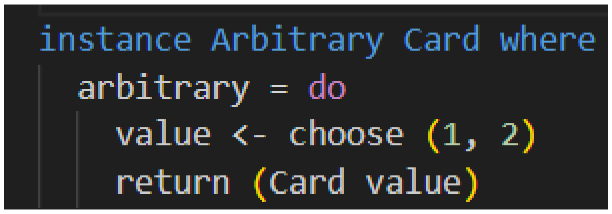
```haskell
generateAction :: Gen Action
generateAction = frequency [(2, return LeaveGame), (6, return ContinuePlaying)]
```

Bet Generator:



```haskell
generateBet :: Gen Int
generateBet = elements [5..100]
```

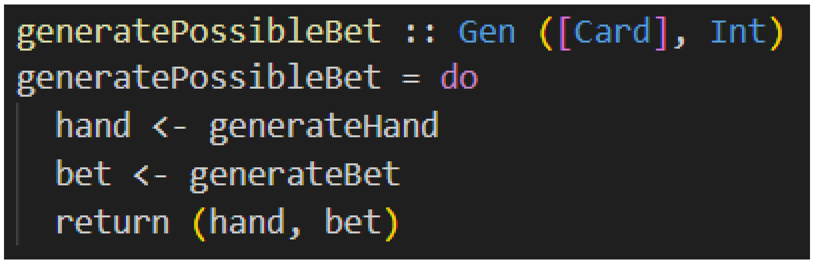
Arbitrary Example:



```haskell
instance Arbitrary Card where
  arbitrary = do
    value <- choose (1, 2)
    return (Card value)
```

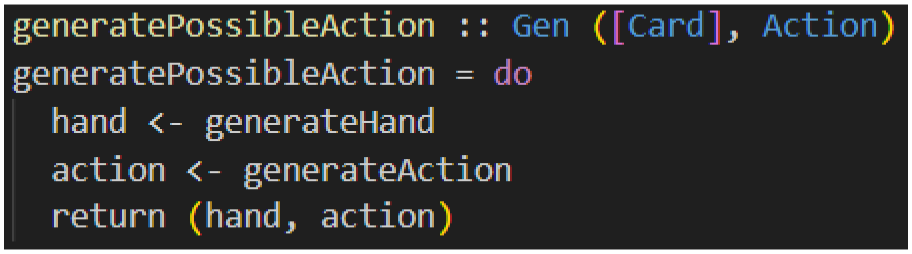
This is an example of an Arbitrary type class and arbitrary method defined for the Card Generator. It is different from the generator below since the arbitrary method only generates Cards that are 1s and 2s. However, the generator we use in our program (generateCard) generates Cards of all Values. This shows how the arbitrary method might not always be what we want to use for generating a data type.

Composite Generators:
Generator that takes in a list of Cards generator and an integer generator.



```haskell
generatePossibleBet :: Gen ([Card], Int)
generatePossibleBet = do
  hand <- generateHand
  bet <- generateBet
  return (hand, bet)
```

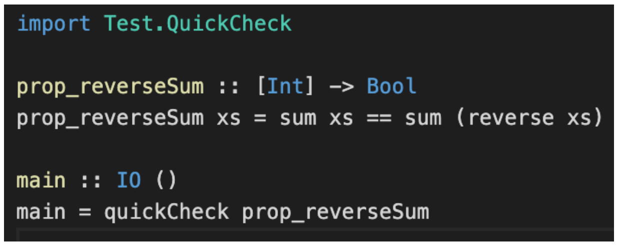
Generator that takes in a list of Cards generator and an Action generator.



```haskell
generatePossibleAction :: Gen ([Card], Action)
generatePossibleAction = do
  hand <- generateHand
  action <- generateAction
  return (hand, action)
```

Comparison of Haskell's QuickCheck and Python's Hypothesis:

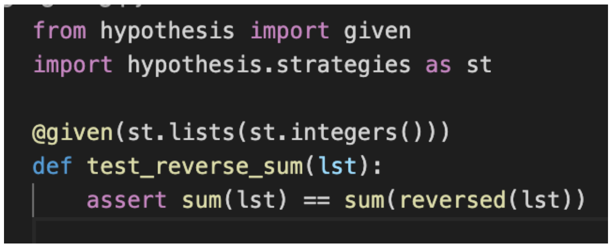
When comparing Haskell's QuickCheck to QuickCheck's implementations in other languages, it is important to examine the difference in design paradigms. Haskell's defining characteristics are that it uses strong typing, lazy evaluation, immutability, and is purely functional. Python's dynamic typing, eager evaluation, mutable variables, and largely object oriented design has resulted in a slightly different and possibly inferior implementation of QuickCheck. While the largest gap in functionality that we found is a result of eager evaluation vs lazy evaluation, we will first go over some of the more trivial differences.



```haskell
import Test.QuickCheck

prop_reverseSum :: [Int] -> Bool
prop_reverseSum xs = sum xs == sum (reverse xs)

main :: IO ()
main = quickCheck prop_reverseSum
```



```python
from hypothesis import given
import hypothesis.strategies as st

@given(st.lists(st.integers()))
def test_reverse_sum(lst):
    assert sum(lst) == sum(reversed(lst))
```

First there are syntactic differences, Python's property based testing suite is called Hypothesis. It uses decorator syntax with @given to set typing and takes in the following test_reverse_sum function as its argument, then defines a testing function that contains Hypothesis's assert function to return a boolean. In Haskell, QuickCheck is able to take prop_reverseSum as an argument, while Python generally would enclose the test within its own function. Haskell is much more concise, while these examples are similar in length, extending this out to a more complex testing suite would require more handwritten assertions in Python, while Haskell would just continue with quickCheck anyFunction.

The biggest difference in functionality we found was the difference in computational speed that results from lazy evaluation vs eager. In the example above, QuickCheck will only generate as much of the list as needed to compute the sum and will return as soon as the sum can be computed. Hypothesis in contrast will need to generate the entire list to be tested before reversing the entire generated list before getting the sum because of its eager evaluation. This, given a large enough list, could become computationally expensive where QuickCheck will avoid much of that overhead. This can become even more obvious when dealing with infinite lists, QuickCheck will again exit as soon as the property being tested has enough data to return, while Hypothesis will be forced to raise a 'StopIteration' exception after first expending the resources to get to that point.

Challenges:

Understanding IO and Generator Monad:

When we first attempted to write our BlackJack game, we were using generate to create hands and then use the generated values within our program. Our initial efforts were held back by our misunderstanding of how QuickCheck's generate worked. We were getting lists with the 'IO [[Card]]' typing and despite our best efforts, we couldn't extract the data from the IO monad in the ways we had previously used. We tried 'do' blocks to extract the data, we tried using lambda functions with the '>>=' binding operator, and we tried changing our typing to have the 'IO [[a]]' signature. None of these worked how we expected, and so we were forced to reexamine our program, our understanding of QuickCheck, and our understanding of the Haskell environment.

We eventually came to the conclusion that because of the way QuickCheck creates external outputs, we had to refactor our program. While we originally had tried to use the data generated inside our functions, we needed to separate our generation from our logic. While this clearly matches what was demonstrated in our lectures, grasping the reasons for separation was key to solving the problems in our code.

Setting Up Our Project Repository and Code:

Since this was the first time we were writing Haskell code from scratch instead of writing code for functions, we had some trouble setting up the repository and how to link main with our file that contained all of the functions. Eventually we had to learn how to work with the stack.yaml file and put the correct modules into it so that we could build the project properly. Then we figured out how to link our files together so we could run our code. Initially we were also not sure what to import but we figured out we could use the compiler from stack build to guide us on what we needed to import.

Comparing QuickCheck Shrink vs Hypothesis Assume:

In our research we found that shrink and assume can be used to largely the same effect. However it seems that while they are analogous, we had issues truly grasping the difference in usage. We know that shrink is iterative and will modify its testing after finding a failure to find the smallest test case that generates that failure. Assume is largely blind to its own functioning and will simply continue generating after finding a failure and it is up to the user to narrow down what exactly caused it. When investigating, we began finding contradictory information about how each one functions and felt uncomfortable trying to give a definitive explanation in this paper. Some sources held that Assume was more customizable and would point out how Shrink's automatic behavior was detrimental while others would write the opposite.

Sources:

1. Property Based Testing
   https://en.wikipedia.org/wiki/Property_testing

2. QuickCheck Brief Definition & History
   https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html

3. Original Quickcheck Paper
   https://users.cs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

4. Haskell Arbitrary TypeClass
   https://hackage.haskell.org/package/QuickCheck-2.14.3/docs/Test-QuickCheck.html#t:Arbitrary

5. Shrink Method Arbitrary TypeClass
   https://stackoverflow.com/questions/16968549/what-is-a-shrink-with-regard-to-haskells-quickcheck#:~:text=A%20single%20shrink%20is%20a,of%20them%20in%20a%20list.

6. IO and Safety
   https://wiki.haskell.org/Introduction_to_IO

7. Lazy vs Eager
   https://stackoverflow.com/questions/75680491/what-is-the-trade-off-between-lazy-and-strict-eager-evaluation

8. Assume vs Shrink
   https://hypothesis.works/articles/types-and-properties/

Distribution of work:

Vivek: Wrote generators for the Card, Hand, Action, and Bet. Composite and simple generators, Arbitrary typeclass, setup repository and code.

John: Blackjack Logic, Initial Program Flow and Structure, Introduction, Understanding IO and Generator Monad, IO Monad section, Comparison of Haskell's QuickCheck and Python's Hypothesis, Comparing QuickCheck Shrink vs Hypothesis Assume