1 Monads: An interface for instructions

1.1 primitive instructions
1.2 combining instructions (bind)
1.3 injecting values (return)

2 QuickCheck Instructions

2.1 QuickCheck can perform random testing with any value of a type which is a member of type class Arbitrary

2.2 For any type a in Arbitrary there is a random value generator of type Gen a

2.3 Gen is a monad

2.4 What are the instructions for this monad?

class NotArbitrary a where notArbitrary :: a
  defines a constant value for each type
3 IO vs GEN

3.1 IO a

3.1.1 Instructions to build a value of type a by interacting with the operating system.

3.1.2 Executed by the run-time system.

3.2 Gen a

3.2.1 Instructions to create a random value of type a

3.2.2 Executed by QuickCheck library functions.

4 Instructions for Test Data Generation

4.1 Why monad / instructions?

4.1.1 want to generate different data every time

4.1.2 construction method remains the same

4.2 Need data generation at different types

Prelude Test.QuickCheck> :i Arbitrary

class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]

..

5 Sampling

5.1 Testing of generators

sample :: Gen a -> IO ()

generates a few value and prints them

Prelude Test.QuickCheck> sample (arbitrary :: Gen Integer)
Prelude Test.QuickCheck> sample (arbitrary :: Gen Boolean)
Prelude Test.QuickCheck> sample (arbitrary :: Gen Doubles)
Prelude Test.QuickCheck> sample (arbitrary :: Gen [Integer])

,,
6 Writing generators

6.1 The constant generator

6.1.1 return True always returns True

6.2 Using do notation

Prelude Test.QuickCheck> sample $ doTwice (arbitrary :: Gen Integer)
...

6.3 Even integers

evenInteger :: Gen Integer
evenInteger = do
  n <- arbitrary
  return (2*n)

7 Generation Library

7.1 Choosing from a range

choose :: Random a => (a, a) -> Gen a

7.2 Choosing between generators

oneof :: [Gen a] -> Gen a

8 Example: Generating a Suit

8.1 Recall

data Suit = Spades | Hearts | Diamonds | Clubs
  deriving (Show, Eq)

8.2 Generator for suits

rSuit :: Gen Suit
rSuit = oneof [return Spades,
  return Hearts,
  return Diamonds,
  return Clubs]
9  More generators

9.1  Choosing between elements

\[ \text{elements :: \{a\} -> Gen a} \]

9.2  Can you define \text{elements} using \text{oneof}?

10  Generating a Rank

\[
\text{data Rank = Numeric Integer | Jack | Queen | King | Ace}
\]

\[
\text{deriving (Show, Eq)}
\]

\[
\text{rRank = undefined}
\]

11  Generating a Card

\[
\text{data Card = Card Rank Suit}
\]

\[
\text{deriving (Show, Eq)}
\]

\[
\text{rCard = undefined}
\]

12  Generating a Hand

\[
\text{data Hand = Empty | Add Card Hand}
\]

\[
\text{deriving (Show, Eq)}
\]

\[
\text{rHand = undefined}
\]

13  Making QuickCheck Use Our Generators

13.1  QuickCheck is type agnostic

13.2  It works with any type that is an instance of \text{Arbitrary}

\[
\text{instance Arbitrary Suit where}
\]

\[
\text{arbitrary = rSuit}
\]
14 Datatype Invariants

14.1 Sometimes datatype contain unwanted values

14.2 Example: Numeric 0

14.3 Filtering the valid values

\[
\text{validRank} :: \text{Rank} \rightarrow \text{Bool} \\
\text{validRank} \ (\text{Numeric} \ r) = 2 \leq r \land r \leq 10 \\
\text{validRank} \ _ = \text{True}
\]

14.4 A datatype invariant which should always by True

14.5 Test it

15 Test Data Distribution

15.1 Problem: what are the successful test cases?

15.2 They could be insignificant values

15.3 It’s important to know the distribution of the test data

\[
\text{prop\_Rank} \ r = \text{collect} \ r \ (\text{validRank} \ r)
\]

15.4 collects and prints the tested values

16 Observing the Distribution of Ranks

17 Fixing the Generator

\[
\text{rRank} = \text{frequency} \ [
\begin{array}{ll}
(1, \text{return Jack}), \\
(1, \text{return Queen}), \\
(1, \text{return King}), \\
(1, \text{return Ace}), \\
(9, \text{do} \ \{r \leftarrow \text{choose} \ (2,10); \text{return} \$ \text{Numeric} \ r\})
\end{array}
\]
\]
18 Distribution of Hands

18.1 Collecting each individual hand generates too much data

18.2 Collect a summary instead, e.g., the number of cards

\[\text{numCards :: Hand} \rightarrow \text{Integer}\]
\[\text{numCards Empty} = 0\]
\[\text{numCards (Add } x \text{ h)} = 1 + \text{numCards h}\]

18.3 Collecting the distribution

\[\text{prop_Hand } h = \text{collect (numCards h) True}\]

19 Fixing the generator

19.1 Returning Empty 20% of the time gives an average of 5 cards per hand

\[\text{rHand} = \text{frequency}\]
\[(1, \text{return Empty}),\]
\[(4, \text{do } c \leftarrow \text{rCard}; h \leftarrow \text{rHand}; \text{return } \text{Add } c \text{ h})]\]

20 Testing Algorithms

20.1 insert \(x\) \(xs\)

20.2 Inserts an element \(x\) into an ordered list \(xs\)

20.3 Result is also ordered

\[\text{prop_insert :: Integer} \rightarrow [\text{Integer}] \rightarrow \text{Bool}\]
\[\text{prop_insert } x \text{ xs} = \text{ordered (insert } x \text{ xs)}\]

20.4 Too weak: Precondition missing

21 Testing insert

\[\text{prop_insert’ :: Integer} \rightarrow [\text{Integer}] \rightarrow \text{Property}\]
\[\text{prop_insert’ } x \text{ xs} = \text{ordered xs} \Rightarrow \text{ordered (insert } x \text{ xs)}\]
21.1 However, it turns out that many tests are very short:

```haskell
prop_insert' :: Integer -> [Integer] -> Property
prop_insert' x xs =
    collect (length xs) $
    ordered xs => ordered (insert x xs)
```

22 Probability that a random list is ordered

22.1 Length 0: 100%
22.2 Length 1: 100%
22.3 Length 2: 50%
22.4 Length 3: 17%
22.5 Length 4: 4%

23 Generating ordered lists from the start

```haskell
orderedList :: Gen [Integer]
orderedList = undefined
```

24 Using a Custom Generator

24.1 The type should say that the list is ordered
24.2 Define a new type

```haskell
data OrderedList = Ordered [Integer]
instance Arbitrary OrderedList where
    arbitrary = do {ol <- orderedList; return Orderedlist ol}
```

24.3 Testing insert properly

```haskell
prop_insert' :: Integer -> Orderedlist -> Bool
prop_insert' x (Orderedlist xs) = ordered (insert x xs)
```
25 Summary

25.1 How to generate test data for quickCheck

25.1.1 Custom datatypes

25.1.2 Custom invariants

25.2 IO and Gen are both members of the Monad class (instructions)

25.3 How to create our own instructions?