

Test Data Generators

Why Distinguish Instructions?

- *Functions* always give the same result for the same arguments
- *Instructions* can behave differently on different occasions
- Confusing them (as in most programming languages) is a major source of bugs
 - This concept a major breakthrough in programming languages in the 1990s
 - How would you write doTwice in C?

Monads = Instructions

- What is the type of doTwice?

```
Main> :i doTwice  
doTwice :: Monad m => m a -> m (a,a)
```

Even the *kind of instructions* can vary!
Different kinds of instructions, depending on who obeys them.

Whatever kind of result argument produces, we get a pair of them

IO means operating system.

Instructions for Test Data Generation

- Generate *different* test data every time
 - Hence need "instructions to generate an a"
 - Instructions to QuickCheck, not the OS
 - Gen a \neq IO a
- Generating data of different types?

```
QuickCheck> :i Arbitrary
-- type class
class Arbitrary a where
  arbitrary :: Gen a
```

Sampling

- We provide *sample* to print some sampled values:

```
sample :: Gen a -> IO ()
```

- Example:

```
Sample> sample (arbitrary :: Gen Integer)
```

```
1
```

```
0
```

```
-5
```

```
14
```

```
-3
```

Fix the
type we
generate

Prints (fairly small) test
data QuickCheck might
generate

Sampling Booleans

```
Sample> sample (arbitrary :: Gen Bool)
```

```
True
```

```
False
```

```
True
```

```
True
```

```
True
```

Sampling Doubles

Sample> sample (arbitrary :: Gen Double)

-5.75

-1.75

2.166666666666667

1.0

-9.25

Sampling Lists

```
Sample> sample (arbitrary :: Gen [Integer])
```

```
[-15,-12,7,-13,6,-6,-2,4]
```

```
[3,-2,0,-2,1]
```

```
[]
```

```
[-11,14,2,8,-10,-8,-7,-12,-13,14,15,15,11,7]
```

```
[-4,10,18,8,14]
```


Writing Generators

- Write instructions using **do** and return:

```
Sample> sample (return True)
```

```
True
```

```
True
```

```
True
```

```
True
```

```
True
```

Writing Generators

- Write instructions using **do** and return:
Main> sample (doTwice (arbitrary :: Gen Integer))
(12,-6)
(5,5)
(-1,-9)
(4,2)
(13,-6)

It's important that the instructions are followed *twice*, to generate two *different* values.

Writing Generators

- Write instructions using **do** and return:

```
Main> sample evenInteger
```

```
-32
```

```
-6
```

```
0
```

```
4
```

```
0
```

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

Generation Library

- QuickCheck provides *many* functions for constructing generators

```
Main> sample (choose (1,10) :: Gen Integer)
```

```
6
```

```
7
```

```
10
```

```
6
```

```
10
```

Generation Library

- QuickCheck provides *many* functions for constructing generators

```
Main> sample (oneof [return 1, return 10])
```

```
1
```

```
1
```

```
10
```

```
1
```

```
1
```

```
oneof :: [Gen a] -> Gen a
```

Generating a Suit

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

```
Main> sample suit  
Spades  
Hearts  
Diamonds  
Diamonds  
Clubs
```

```
suit :: Gen Suit  
suit = oneof [return Spades,  
             return Hearts,  
             return Diamonds,  
             return Clubs]
```

QuickCheck chooses one set of instructions from the list

Generating a Rank

```
data Rank = Numeric Integer  
          | Jack | Queen | King | Ace  
deriving (Show,Eq)
```

```
Main> sample rank  
Numeric 4  
Numeric 5  
Numeric 3  
Queen  
King
```

```
rank = oneof  
      [return Jack,  
       return Queen,  
       return King,  
       return Ace,  
       do r <- choose (2,10)  
         return (Numeric r)]
```

Generating a Card

```
data Card = Card Rank Suit  
  deriving (Show,Eq)
```

```
Main> sample card  
Card Ace Hearts  
Card King Diamonds  
Card Queen Clubs  
Card Ace Hearts  
Card Queen Clubs
```

```
card =  
  do r <- rank  
      s <- suit  
      return (Card r s)
```


Generating a Hand

```
data Hand = Empty | Some Card Hand  
deriving (Eq, Show)
```

```
Main> sample hand
```

```
Some (Card Jack Clubs) (Some (Card Jack Hearts) Empty)
```

```
Empty
```

```
Some (Card Queen Diamonds) Empty
```

```
Empty
```

```
Empty
```

```
hand = oneof  
  [return Empty,  
   do c <- card  
     h <- hand  
     return (Some c h)]
```

Making QuickCheck Use Our Generators

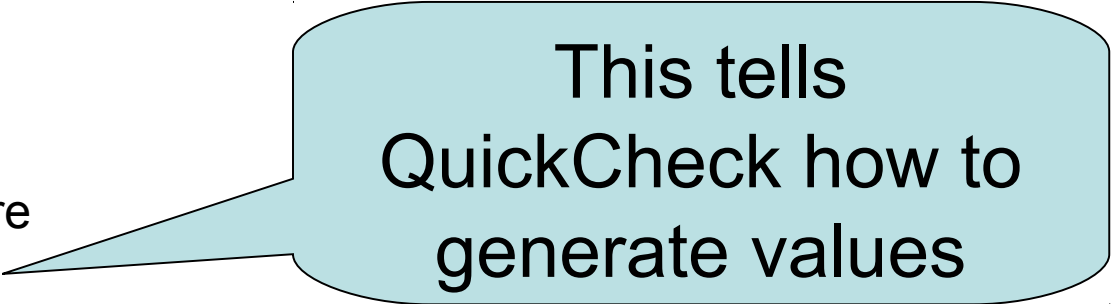
- QuickCheck can generate any type of class

Arbitrary:

```
Main> :i Arbitrary
-- type class
class Arbitrary a where
  arbitrary :: Gen a
```

```
-- instances:
```

```
instance Arbitrary ()
instance Arbitrary Bool
instance Arbitrary Int
...
```



This tells QuickCheck how to generate values

Making QuickCheck Use Our Generators

- QuickCheck can generate any type of class Arbitrary
- So we have to make our types instances of this class

Make a new instance

...of this class...

...for this type...

**instance Arbitrary Suit where
arbitrary = suit**

...where this method...

...is defined like this.

Datatype Invariants

- We design types to *model our problem* – but rarely perfectly
 - Numeric (-3) ??
- Only certain values are valid

```
validRank :: Rank -> Bool
validRank (Numeric r) = 2<=r && r<=10
validRank _           = True
```

- This is called the *datatype invariant* – should always be True

Testing Datatype Invariants

- Generators should only produce values satisfying the datatype invariant:

```
prop_Rank r = validRank r
```

- Stating the datatype invariant helps us understand the program, avoid bugs
- Testing it helps uncover errors in test data generators!

Testing code needs testing too!

Test Data Distribution

- We don't see the test cases when quickCheck succeeds
- Important to know what kind of test data is being used

`prop_Rank r = collect r (validRank r)`

This property *means* the same as `validRank r`, but when tested, collects the values of `r`

Distribution of Ranks

```
Main> quickCheck prop_Rank
```

```
OK, passed 100 tests.
```

```
26% King.
```

```
25% Queen.
```

```
19% Jack.
```

```
17% Ace.
```

```
7% Numeric 9.
```

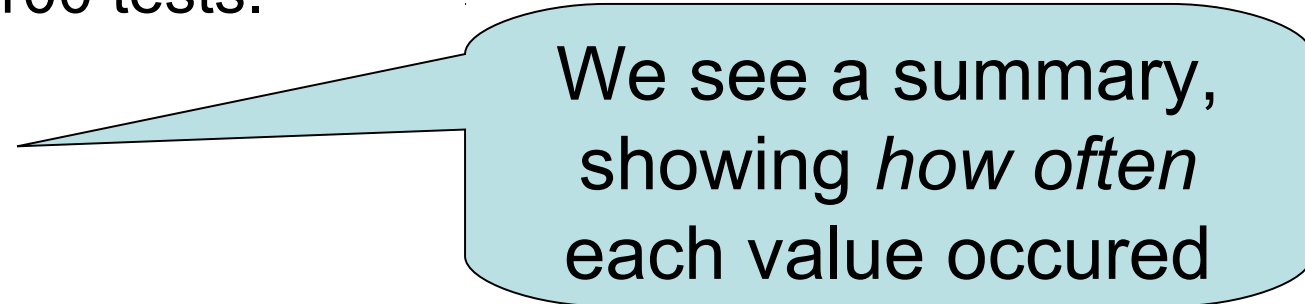
```
2% Numeric 7.
```

```
1% Numeric 8.
```

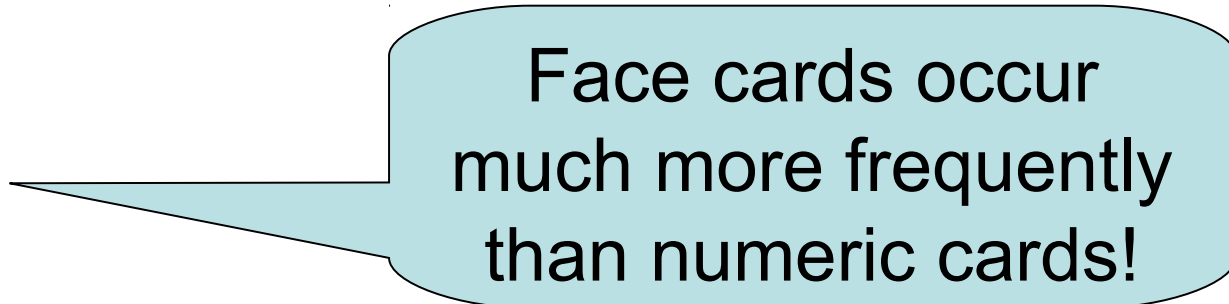
```
1% Numeric 6.
```

```
1% Numeric 5.
```

```
1% Numeric 2.
```



We see a summary, showing *how often* each value occurred



Face cards occur much more frequently than numeric cards!

Fixing the Generator

```
rank = frequency  
  [(1,return Jack),  
   (1,return Queen),  
   (1,return King),  
   (1,return Ace),  
   (9, do r <- choose (2,10)  
     return (Numeric r))]
```

Each alternative is paired with a *weight* determining how often it is chosen.

Choose number cards 9x as often.

Distribution of Hands

- Collecting each hand generated produces too much data—hard to understand
- Collect a summary instead—say the number of cards in a hand

```
numCards :: Hand -> Integer
```

```
numCards Empty      = 0
```

```
numCards (Some _ h) = 1 + numCards h
```

Distribution of Hands

```
prop_Hand h = collect (numCards h) True
```

```
Main> quickCheck prop_Hand
```

OK, passed 100 tests.

53% 0.

25% 1.

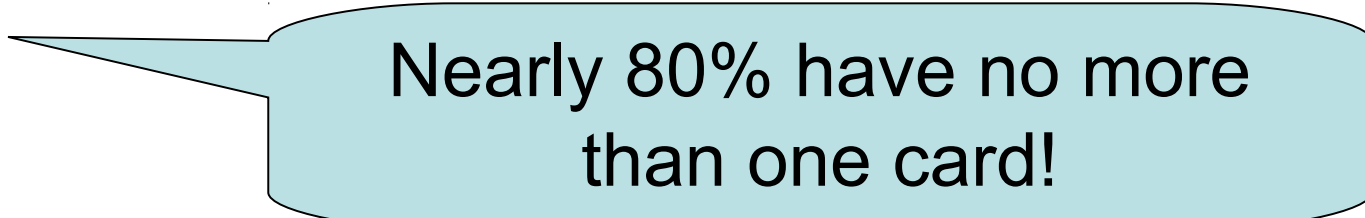
9% 2.

5% 3.

4% 4.

2% 9.

2% 5.



Nearly 80% have no more than one card!

Fixing the Generator

```
hand = frequency [(1,return Empty),  
                  (4, do c <- card  
                      h <- hand  
                      return (Some c h))]
```

- Returning Empty
20% of the time
gives average
hands of 5 cards

```
Main> quickCheck prop_Hand  
OK, passed 100 tests.  
22% 0.  
13% 2.  
13% 1.  
12% 5.  
12% 3.  
6% 4.  
4% 9.  
4% 8.  
...
```

Testing Algorithms

Testing insert

- `insert x xs`—inserts `x` at the right place in an ordered list

```
Main> insert 3 [1..5]
```

```
[1,2,3,3,4,5]
```

- The result should always be ordered

```
prop_insert :: Integer -> [Integer] -> Bool
prop_insert x xs = ordered (insert x xs)
```

Testing insert

Main> quickCheck prop_insert

Falsifiable, after 2 tests:

3

[0,1,-1]

Of course, the result won't be ordered unless the input is

```
prop_insert :: Integer -> [Integer] -> Property
prop_insert x xs =
  ordered xs ==> ordered (insert x xs)
```

Testing succeeds, but...

Testing insert

- Let's observe the test data...

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

```
Main> quickCheck prop_insert
OK, passed 100 tests.
41% 0.
38% 1.
14% 2.
6% 3.
1% 5.
```

Why so short???

What's the Probability a Random List is Ordered?

| Length | Ordered? |
|--------|----------|
| 0 | 100% |
| 1 | 100% |
| 2 | 50% |
| 3 | 17% |
| 4 | 4% |

Generating Ordered Lists

- Generating random lists and choosing ordered ones is silly
- Better to generate ordered lists to begin with—but how?
- One idea:
 - Generate an arbitrary list
 - sort it

The Ordered List Generator

```
orderedList :: Gen [Integer]
orderedList =
  do xs <- arbitrary
      return (sort xs)
```

Trying it

```
Main> sample orderedList
```

```
[]
```

```
[-4,-1,3]
```

```
[-5,-4,-3,1,2]
```

```
[-6,0,4,7]
```

```
[-10,-9,-9,-7,1,2,2,8,10,10]
```

Making QuickCheck use a Custom Generator

- Can't redefine arbitrary: the *type* doesn't say we should use `orderedList`
- Make a **new type**

```
data OrderedList = Ordered [Integer]
```

A new type

with a datatype invariant

Making QuickCheck use a Custom Generator

- Make a **new type**

```
data OrderedList = Ordered [Integer]
```

- Make an instance of Arbitrary

```
instance Arbitrary OrderedList where  
  arbitrary =  
    do xs <- orderedList  
      return (Ordered xs)
```

Testing insert Correctly

```
prop_insert :: Integer -> OrderedList -> Bool
```

```
prop_insert x (Ordered xs) =  
  ordered (insert x xs)
```

```
Main> quickCheck prop_insert  
OK, passed 100 tests.
```

Collecting Data

```
prop_insert x (Ordered xs) =  
  collect (length xs) $  
  ordered (insert x xs)
```

```
Main> quickCheck prop_insert
```


```
OK, passed 100 tests.
```

```
17% 1.
```

```
16% 0.
```

```
12% 3.
```

```
12% 2.....
```



Wide variety of
lengths

Reading

- About I/O: Chapter 9 of Learn You a Haskell
- About QuickCheck: read the *manual* linked from the course web page.
 - There are also several research papers about QuickCheck, and advanced tutorial articles.