#### **Test Data Generators**

Based on original slides by Koen Claessen and John Hughes

#### A recap: Instructions

- A new built-in type < Instructions to the Operating System</li>
   IO a
- Standard functions:
  - putStr :: String -> IO ()

() is the "empty tuple"– no interestingcontents

- readFile :: FilePath -> IO String
- writeFile :: FilePath -> String -> IO ()

#### An example

Recall putStr :: String -> IO() What happens with this expression:

last [putStr "apa", putStr "bepa", putStr "cepa"]



## **Repeating Instructions**

doTwice io =
 do a <- io
 b <- io
 return (a,b)
dont io =
 return ()</pre>

An instruction to compute the given result

Main> doTwice (print "hello")

"hello"

"hello"

((),())

Writing instructions and obeying them are two different things!

Main> dont (print "hello")

# 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?

# Instructions are in the Monad Gang

• What is the type of doTwice?

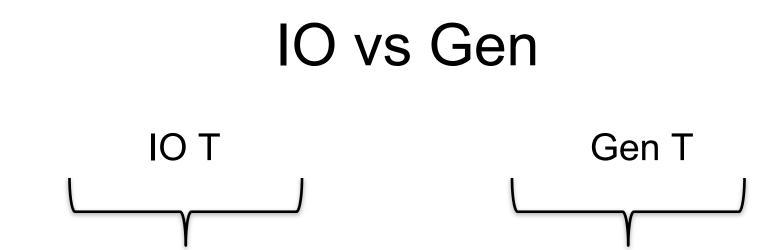
Main> :i doTwice doTwice :: Monad a => a b -> a (b,b)

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.

#### **QuickCheck Instructions**

- QuickCheck can perform random testing with values of any type which is in class Arbitrary
- For any type *T* in Arbitrary there is a random value generator, Gen T
- Gen is a Monad so things of type Gen T are another kind of "instruction"



- Instructions to build a value of type T by interacting with the operating system
- Run by the ghc runtime system

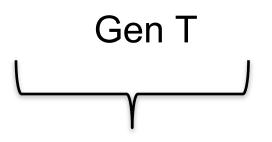
Instructions to create a random value of type T

 Run by the QuickCheck library functions to perform random tests

# 

- Instructions to build a value of type T by interacting with the operating system
- Run by the ghc runtime system

Give ghci an expression of type IO a and it will run the instruction automatically



- Instructions to create a random value of type T
- Run by the QuickCheck library functions to perform random tests

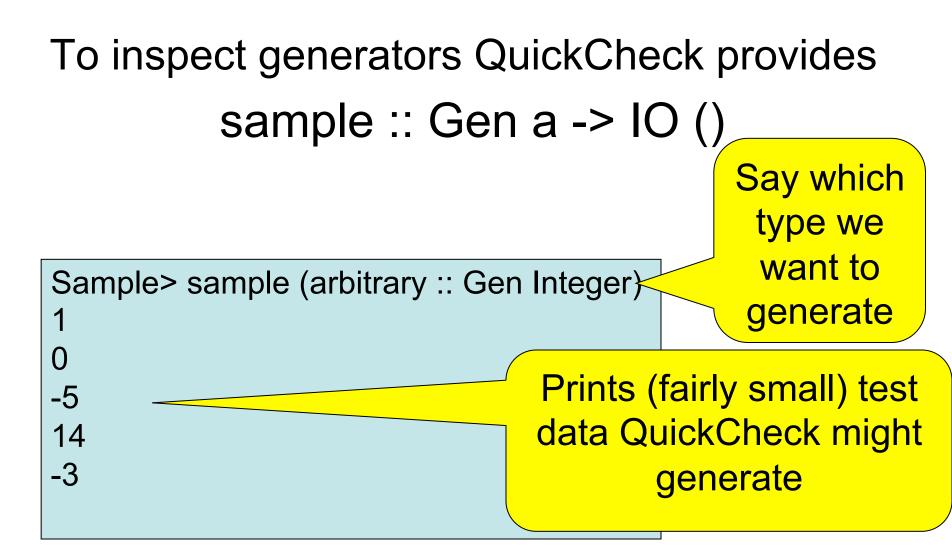
You can also run them using generate, sample, and sample'

#### 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 ≠ IO a
- Generating data of different types?

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

#### Sampling



#### Sampling Booleans

- Sample > sample (arbitrary :: Gen Bool)
- True
- False
- True
- True

True

 Note: the definition of sample is not important here – it is just a way for QuickCheck users to "inspect" something of type Gen a.

#### Sampling Doubles

- Sample > sample (arbitrary :: Gen Double)
- -5.75
- -1.75
- 2.1666666666667
- 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

 We build generators in the same way we build other instructions (like IO): using exiting generators, return and do:

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)It's important that the (5,5)instructions are followed (-1, -9)*twice*, to generate two (4,2)

(13, -6)

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 10 1 1

#### Generating a Suit

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

Main> sample rSuit Spades Hearts Diamonds Diamonds

Clubs

QuickCheck chooses one set of instructions from the list

#### Generating a Suit

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

Alternative definition:

Quiz: define elements using oneof rSuit :: Gen Suit rSuit = elements [Spades, Hearts, Diamonds, Clubs]

> QuickCheck chooses one of the elements from the list

#### Generating a Rank

rRank = oneof [return Jack, return Queen, Main> sample rRank Numeric 4 Numeric 5 Numeric 3 Queen King

#### Generating a Card

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

Main> sample rCard Card Ace Hearts Card King Diamonds Card Queen Clubs Card Ace Hearts Card Queen Clubs rCard = **do** r <- rRank s <- rSuit return (Card r s)

#### Generating a Hand

**type** Hand = [Cards]

If we tell quickCheck how to generate Cards then it will automatically knows how to generate a list of cards.

Disadvantage: we cannot control it (hands with 100 cards possible)

#### Generating a Deck

# data Deck = Deck {cards :: [Card]} deriving (Eq, Show)

rDeck = do cs <- listOf rCard
 return (Deck (nub cs))</pre>

Data.List.nub removes duplicates from the list. What property does that give us?

#### Making QuickCheck Use Our Generators

- QuickCheck can generate any type which is a member of class Arbitrary:
  - Main> :i Arbitrary
  - -- type class
  - class Arbitrary a where
    - arbitrary :: Gen a
    - shrink :: a -> [a]
  - -- instances:

...

instance Arbitrary ()
instance Arbitrary Bool
instance Arbitrary Int

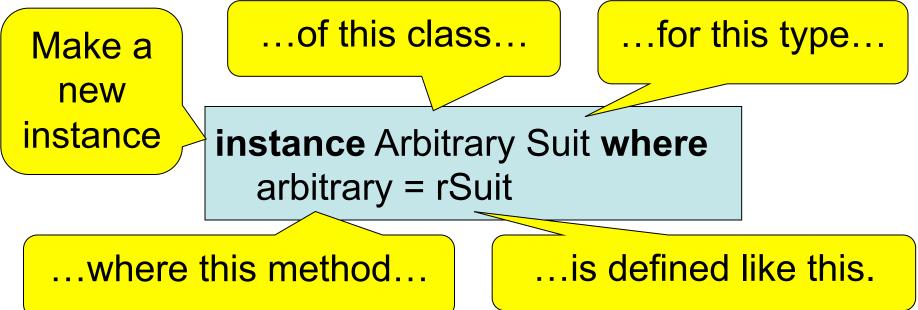
how to generate values

This tells QuickCheck

This helps QuickCheck find small counterexamples (we won't be using this)

#### Making QuickCheck Use Our Generators

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



#### **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</pre>

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. We see a summary, 25% Queen. showing how often 19% Jack. each value occured 17% Ace. 7% Numeric 9. Face cards occur much 2% Numeric 7. more frequently than 1% Numeric 8. numeric cards! 1% Numeric 6. 1% Numeric 5. 1% Numeric 2.

#### Fixing the Generator

rRank = *frequency* [(4,elements [Jack, Queen, King, 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 9 to 4 ratio.

#### Datatype Invariant?

prop\_Deck d = collect (length (cards d)) True

We're not testing any particular property of Decks, just inspecting the distribution

 Are there properties that every deck should have?

#### Datatype Invariant?

prop\_Deck d = collect len ( len <= 52) where len = length (cards d)

#### **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

Of course, the result won't be ordered unless the input is [0,1,-1]

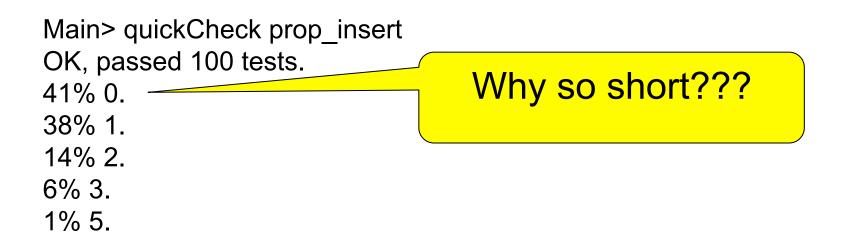
prop insert :: Integer -> [Integer] -> Property prop insert x xs = ordered xs ==> ordered (insert x xs)

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)
```



# 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:
  - Choose a number for the first element
  - Choose a *positive* number to add to it for the next
  - And so on

#### The Ordered List Generator

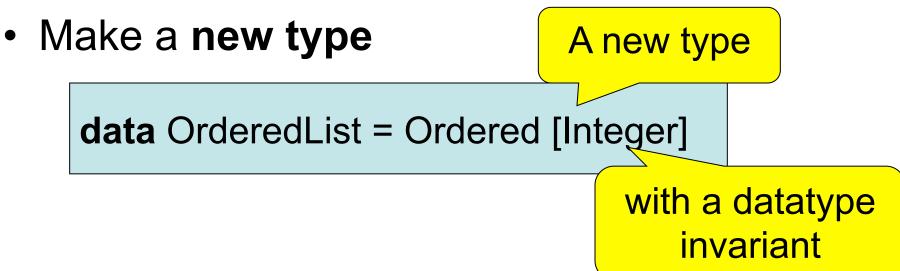
```
orderedList :: Gen [Integer]
orderedList =
  do n <- arbitrary
     listFrom n
  where listFrom n =
         frequency
          [(1, return []),
           (5, do i <- arbitrary
                  ns <- listFrom (n + abs i)</pre>
                  return (n:ns))]
```

# Trying it

Main> sample orderedList [10,21,29,31,40,49,54,55] [3,5,5,7,10] [0,1,2] [7,7,11,19,28,36,42,51,61] []

#### Making QuickCheck use a Custom Generator

 Can't redefine arbitrary: the type doesn't say we should use orderedList



#### Making QuickCheck use a Custom Generator

• Make a **new type** 

data OrderedList = Ordered [Integer] deriving Show

• Make an instance of Arbitrary

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

### **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....

## Summary

- We have seen how to generate test data for quickCheck
  - Custom datatypes (Card etc)
  - Custom invariants (ordered lists)
- Seen that IO A and Gen A are members of the Monad class (the class of "instructions")
- Later: how to create our own "instructions" (i.e. creating an instance of Monad)

# Reading

 About IO and do notation: 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.
  - Real World Haskell, Thompson (3rd edition)