

# Functional Programming

## Test data generators

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2022/23

# An application of type classes and monads

## Remember QuickCheck for testing

- Automatic generation of test cases to test properties specified by the programmer
- So far restricted to properties on predefined types

# An application of type classes and monads

## Remember QuickCheck for testing

- Automatic generation of test cases to test properties specified by the programmer
- So far restricted to properties on predefined types

## But really...

Test data can be generated for the instances of a type class `Arbitrary` (defined by QuickCheck)

# An application of type classes and monads

## Remember QuickCheck for testing

- Automatic generation of test cases to test properties specified by the programmer
- So far restricted to properties on predefined types

## But really...

Test data can be generated for the instances of a type class `Arbitrary` (defined by QuickCheck)

## To extend the scope of QuickCheck...

- we only need to write new instance of `Arbitrary`!
- (requires the IO monad to generate random numbers)

## An example

```
prop_binomi :: Integer -> Integer -> Bool
prop_binomi a b = (a + b) ^ 2 == a ^ 2 + 2 * a * b + b ^ 2
```

can be checked

```
Main> quickCheck prop_binomi
+++ OK, passed 100 tests.
```

# Arbitrary and Gen

## Type class Arbitrary

```
class Arbitrary a where  
  arbitrary :: Gen a -- generate values of type a  
  shrink :: a -> [a] -- shrink values of type a
```

Type Gen a: instructions for creating a random value of type a (a monad)

# Arbitrary and Gen

## Type class Arbitrary

```
class Arbitrary a where
  arbitrary :: Gen a -- generate values of type a
  shrink    :: a -> [a] -- shrink values of type a
```

Type Gen a: instructions for creating a random value of type a (a monad)

## Functions for sampling a random generator

```
sample    :: Show a => Gen a -> IO ()
sample'   :: Gen a -> IO [a]
generate  :: Gen a -> IO a
```

# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`



# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`
- `[(),(),(),(),(),(),(),(),(),(),(),()]`

# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`
- `[(),(),(),(),(),(),(),(),(),(),(),()]`
- `Main> sample' (arbitrary :: Gen Bool)`

# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`
- `[(),(),(),(),(),(),(),(),(),(),()]`
- `Main> sample' (arbitrary :: Gen Bool)`
- `[True,False,False,False,False,True,True,False,False,True,False]`

# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`
- `[(),(),(),(),(),(),(),(),(),(),()]`
- `Main> sample' (arbitrary :: Gen Bool)`
- `[True,False,False,False,False,True,True,False,False,True,False]`
- `Main> sample' (arbitrary :: Gen Int)`

# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`
- `[(),(),(),(),(),(),(),(),(),(),()]`
- `Main> sample' (arbitrary :: Gen Bool)`
- `[True,False,False,False,False,True,True,False,False,True,False]`
- `Main> sample' (arbitrary :: Gen Int)`
- `[0,2,0,6,5,2,-12,9,-15,-2,20]`

# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`
- `[(),(),(),(),(),(),(),(),(),(),()]`
- `Main> sample' (arbitrary :: Gen Bool)`
- `[True,False,False,False,False,True,True,False,False,True,False]`
- `Main> sample' (arbitrary :: Gen Int)`
- `[0,2,0,6,5,2,-12,9,-15,-2,20]`
- `Main> sample' (arbitrary :: Gen Int)`

# Sampling test data

Remember `sample' :: Gen a -> IO [a]`

- `Main> sample' arbitrary`
- `[(),(),(),(),(),(),(),(),(),(),()]`
- `Main> sample' (arbitrary :: Gen Bool)`
- `[True,False,False,False,False,True,True,False,False,True,False]`
- `Main> sample' (arbitrary :: Gen Int)`
- `[0,2,0,6,5,2,-12,9,-15,-2,20]`
- `Main> sample' (arbitrary :: Gen Int)`
- `[0,-1,0,5,3,-1,-11,-8,14,-10,-19]`

# Building generators

```
elements  :: [a] -> Gen a
oneof     :: [Gen a] -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
listOf    :: Gen a -> Gen [a]
vectorOf  :: Int -> Gen a -> Gen [a]
choose    :: Random a => (a,a) -> Gen a
```

- Random is a predefined type class for generating random data.
- (some experiments)



# Generating a Suit

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

# Generating a Rank

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

# Generating a card

```
data Card = Card { rank :: Rank, suit :: Suit }  
    deriving (Show)
```

- need to combine a generator for Rank and one for Suit
- no provision in the QuickCheck library, but ...

## Gen is a monad

- `Gen a` is the type of instructions to generate random values of type `a`.
- `IO a` is the type of instructions for IO operations with result `a`.
- Both are monads  $\Rightarrow$  use `bind >>=` to combine generators.
- Alternatively, the `do` notation can be used with `Gen`.

# Examples

## Generate

- Card
- constant, twice
- even integers, non-negative integers
- Hand

## Task: Check the generator

- `Rank` contains useless values.
- Does its generator `rRank` only yield useful values?

## Task: Check the generator

- Rank contains useless values.
- Does its generator rRank only yield useful values?

### Test it!

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

prop_all_validRank = forAll rRank validRank
```

# Checking properties of test data

```
prop_all_valid_rank_collect r = collect r (validRank r)
```

- `collect x` does not change the test.
- collects values of `x` and creates a histogram.



# Task

- Define a property that yields a histogram of generated Hands.

# Testing properties of `insert`

# Testing properties of `insert`

## Example

- `insert x xs` inserts a value `x` in an ordered list `xs`.
- The output should be ordered again (along with other properties).
- How do we test that?

## First attempt

```
prop_insert_1 :: Integer -> [Integer] -> Bool  
prop_insert_1 x xs = isOrdered (insert x xs)
```

## Second attempt

```
prop_insert_2 :: Integer -> [Integer] -> Property
prop_insert_2 x xs = isOrdered xs ==> isOrdered (insert x xs)
```

## Third attempt

A dedicated generator for sorted lists

```
orderedList :: (Arbitrary a, Ord a) => Gen [a]
```

(How would you implement this generator?)

## Third attempt

A dedicated generator for sorted lists

```
orderedList :: (Arbitrary a, Ord a) => Gen [a]
```

(How would you implement this generator?)

Usage

```
prop_insert_3 x =  
  forAll orderedList (\xs->isOrdered (insert x xs))
```

## Fourth attempt

A dedicated generator for sorted lists (defined by QuickCheck)

```
data OrderedList a = Ordered [a]

instance (Ord a, Arbitrary a)
    => Arbitrary (OrderedList a) where
    arbitrary = orderedList >>= (return . Ordered)
```



## Fourth attempt

A dedicated generator for sorted lists (defined by QuickCheck)

```
data OrderedList a = Ordered [a]

instance (Ord a, Arbitrary a)
    => Arbitrary (OrderedList a) where
    arbitrary = orderedList >=> (return . Ordered)
```

### Usage

```
prop_insert_4 x (Ordered xs) = isOrdered (insert x xs)
```

## Roll your own test data generators

- Populate the class `Arbitrary` with the types you want to generate.
- Generation of values managed by monad `Gen`.
- Conditional test generation is important for meaningful testing.