Class 11: Property-Based Testing

April 9

Q: why are the `IO()`s combined into a single `IO()` in 2c?

questions from homework

```
:: IO ()
do
  when ...
  print result
```

questions from homework

```
do
  f1
  f2

=

f1 >> f2
```

questions from homework

Q: is it more common to use “do” notation or “>>=”?

questions from homework

Q: monads vs. functors?

questions from homework

powerful type system

+

appetite for abstraction

motivation

```
sort :: [Int] -> [Int]
```

inputs	**outputs**
[2, 1, 3]	[1, 2, 3]
[]	[]
[3, 3, 1, 1]	[1, 1, 3, 3]
⋮	
[-16, 13, 20, 11, 0, 11, -8, -14, -16, 20, 3, 12, -3, 18, 19, 14]	

```
prop_sort :: [Int] -> Bool
prop_sort xs = _______ (sort xs)
```

a property for sort

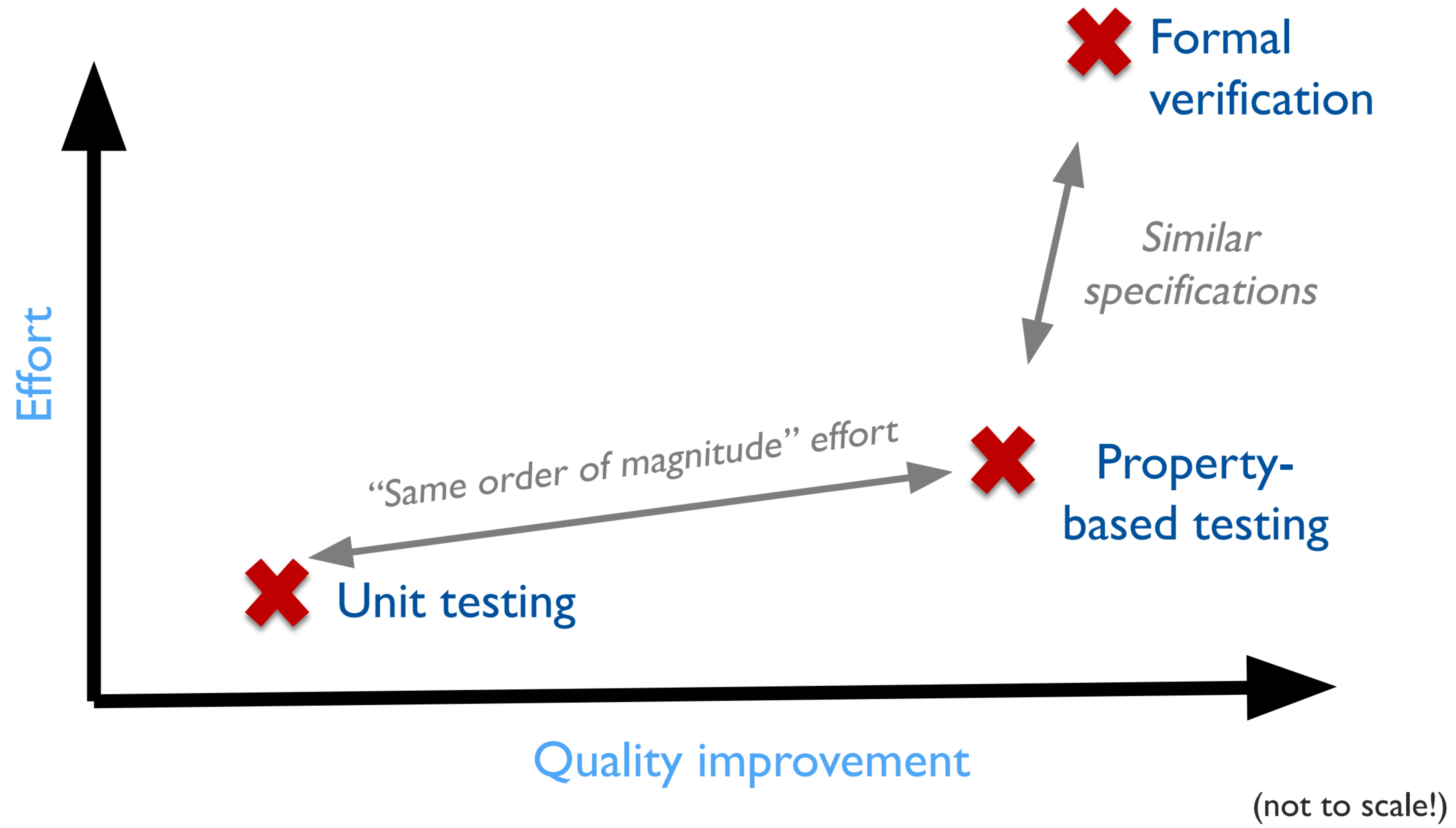
```
prop_sort :: [Int] -> Bool
prop_sort xs = ordered (sort xs)

ordered :: [Int] -> Bool
ordered [] = True
ordered [x] = True
ordered (x1 : x2 : xs) =
  x1 <= x2 && ordered (x2 : xs)
```

a property for sort

(example: quickCheck on good and bad sort #1)

(exercise: property for bad sort #2)



source: Benjamin Pierce's slides

generating random data

random seed

```
data Gen a = Gen (Rand -> a)
```

the Gen type

```
instance Monad Gen where

   return :: a -> Gen a

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

the Gen monad

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

the Gen monad

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->



            )
```

the Gen monad

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r

               )
```

the Gen monad

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
                    (fa r1)
               )
```

the Gen monad

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
                  k (fa r1)
               )
```

the Gen monad

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
         Gen fb = k (fa r1)
               )
```

the Gen monad

```
instance Monad Gen where

   return :: a -> Gen a
   return a = Gen (\_ -> a)

   (>>=) :: Gen a -> (a -> Gen b) -> Gen b
   Gen fa >>= k = Gen (\r ->
     let (r1, r2) = split r
         Gen fb = k (fa r1)
       in fb r2)
```

the Gen monad

```
genBool :: Gen Bool
genBool = return True
```

generator for booleans

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

genBool :: Gen Bool
genBool = oneof [return True, return False]
```

generator for booleans

```
genTwoBool :: Gen (Bool, Bool)
genTwoBool = do
  b1 <- genBool
  b2 <- genBool
  return (b1, b2)
```

combining generators

```
class Arbitrary a where
   arbitrary :: Gen a
```

the Arbitrary type class

```
instance Arbitrary Bool where
   arbitrary :: Gen Bool
   arbitrary =
     oneof [return True, return False]
```

instances of the Arbitrary type class

```
instance
(Arbitrary a, Arbitrary b) => Arbitrary (a, b)
where
```

instances of the Arbitrary type class

```
instance
(Arbitrary a, Arbitrary b) => Arbitrary (a, b)
where
   arbitrary :: Gen (a, b)
   arbitrary =
```

instances of the Arbitrary type class

```
instance
(Arbitrary a, Arbitrary b) => Arbitrary (a, b)
where
   arbitrary :: Gen (a, b)
   arbitrary = do
     a <- (arbitrary :: Gen a)
     b <- (arbitrary :: Gen b)
     return (a, b)
```

instances of the Arbitrary type class

```
instance Arbitrary a => Aribtrary [a] where
   arbitrary :: Gen [a]
   arbitrary =
     oneof
       [ return [],
         do
           x <- arbitrary
           xs <- arbitrary
           return (x : xs)
       ]
```

instances of the Arbitrary type class

```
instance Arbitrary a => Aribtrary [a] where
   arbitrary :: Gen [a]
   arbitrary =
     frequency
       [ (1, return []),
         (4, do
           x <- arbitrary
           xs <- arbitrary
           return (x : xs)
         )
       ]
```

instances of the Arbitrary type class

(exercise: generator for Expr)

size

```
data Gen a = Gen (Rand -> Int -> a)
```

dealing with sizes

```
genNum :: Gen Expr
genNum = do
  n <- arbitrary
  return (Num n)
```

dealing with sizes

```
genExpr :: Int -> Gen Expr
```

dealing with sizes

```
genExpr :: Int -> Gen Expr
genExpr 0 = genNum
```

dealing with sizes

```
genExpr :: Int -> Gen Expr
genExpr 0 = genNum
genExpr n =
  frequency
    [(1, genNum),
     (4, do
       e1 <-
       e2 <-
       return (Add e1 e2)
     )]
```

dealing with sizes

```
genExpr :: Int -> Gen Expr
genExpr 0 = genNum
genExpr n =
  frequency
    [(1, genNum),
     (4, do
       e1 <- genExpr (n `div` 2)
       e2 <- genExpr (n `div` 2)
       return (Add e1 e2)
     )]
```

dealing with sizes

```
instance Arbitrary Expr where
   arbitrary :: Gen Expr
   arbitrary = sized genExpr
```

dealing with sizes

writing with properties

```haskell
quickCheck :: Testable prop => prop -> IO ()
```

the top-level function

```
instance Testable Bool

instance (Arbitrary a, Show a, Testable prop)
  => Testable (a -> prop)
```

the Testable type class

(example: properties for ordered remove)

PBT at Penn