MOD08: Functional Programming

Marco Gerards

14-05-2019

This lecture: plan for today

- More background and examples:
  - Function application
  - Functors
  - Applicative
  - QuickCheck generators
  - Monoids

This lecture: plan for today

- More background and examples:
  - Function application
  - Functors
  - Applicative
  - QuickCheck generators
  - Monoids

- Quiz: 6 questions
  - VoxVote; as before
  - Work in pairs
  - End scores are shown
    - ...but you may vote anonymously
  - For each question:
    - One minute: reading question; think about it for yourself
    - Two minutes: discussion and voting
    - Class discussion
    - Explanation and background

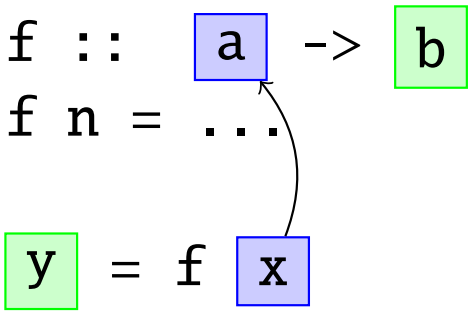
Function application

```
f ::   a -> b
f n = ...

y  = f  x
```

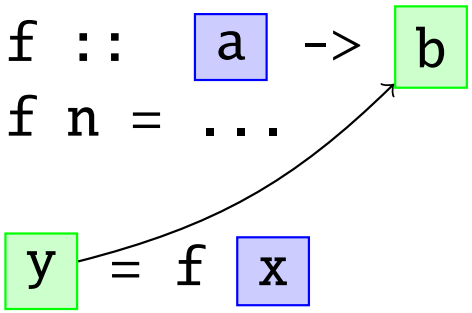
Function application

```
f ::  a -> b
f n = ...

y = f x
```

Function application

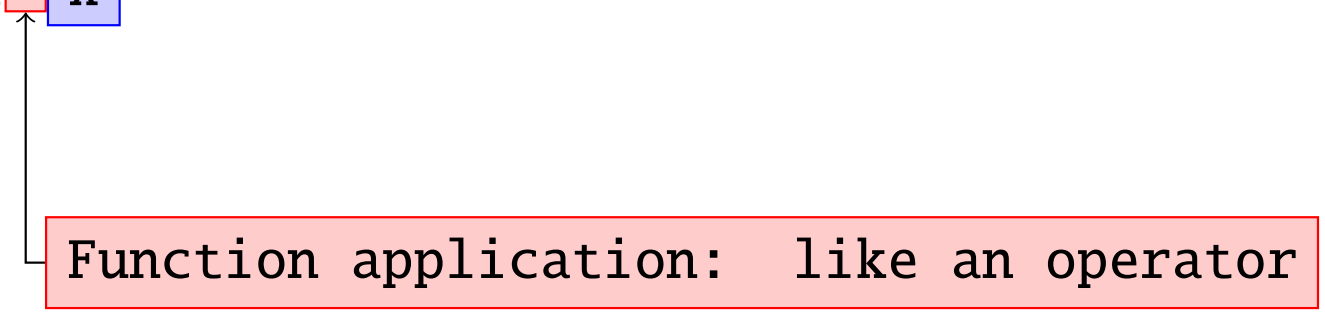


Function application



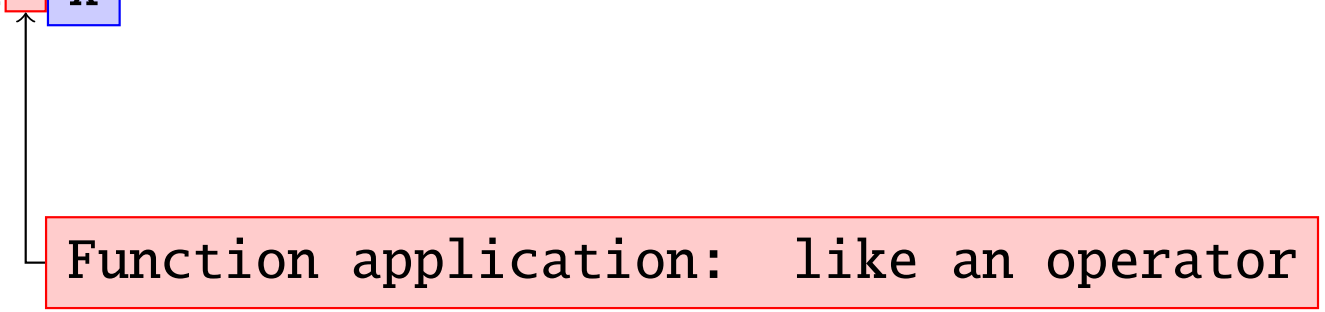
Function application

```
f ::  a -> b
f n = ...

y = f x
```



Function application

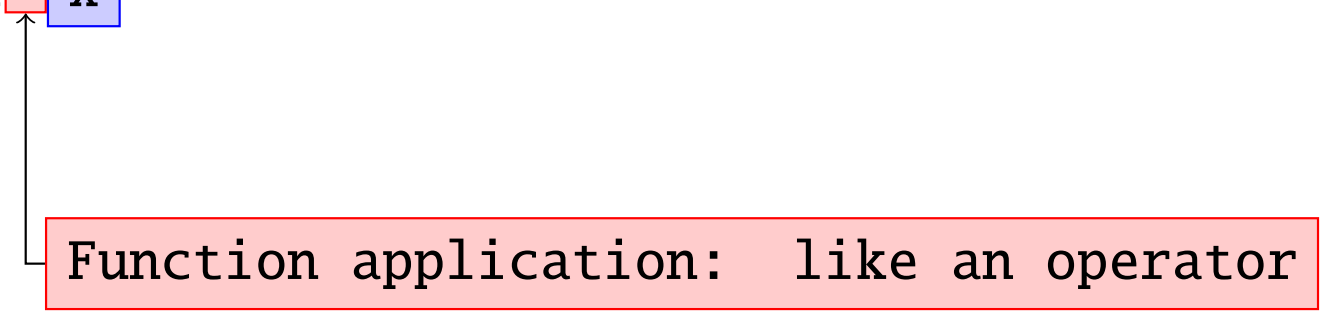
```
f ::  a -> b
f n = ...

y = f x
```



- Left associative: `f x + g y == (f x) + (g y)`

Function application

```
f ::  a -> b
f n = ...

y = f x
```



- Left associative: `f x + g y == (f x) + (g y)`
- `$` Right associative: `f $ x + g y == f (x + g y)`

```
($) :: (a -> b) -> a -> b
```

Functors

Functors

- Functor:
  `fmap :: Functor f => (a -> b) -> f a -> f b`
  - Applies a function in some context `f a`, resulting in `f b`
  - Mapping over a container: apply to all elements

Functors

- Functor:
  `fmap :: Functor f => (a -> b) -> f a -> f b`
  - Applies a function in some context `f a`, resulting in `f b`
  - Mapping over a container: apply to all elements
- Results in a function:
  `fmap :: Functor f => (a -> b) -> (f a -> f b)`
  - *Transforms* a function to a function for some another context

Functors

- Functor:
  `fmap :: Functor f => (a -> b) -> f a -> f b`
  - Applies a function in some context `f a`, resulting in `f b`
  - Mapping over a container: apply to all elements
- Results in a function:
  `fmap :: Functor f => (a -> b) -> (f a -> f b)`
  - *Transforms* a function to a function for some another context
- `f a`
  - `f`: type constructor of kind `* -> *`
  - `a`: some concrete type
  - Only consider the *last* argument of the type constructor

Functor examples

- Functor laws:
  - `fmap id = id`
  - `fmap (f . g) = fmap f . fmap g`
  - Functors are unique!

Functor examples

- Functor laws:
  - `fmap id = id`
  - `fmap (f . g) = fmap f . fmap g`
  - Functors are unique!
- Examples of functors:
  `fmap :: Functor f => (a -> b) -> f a -> f b`
  - `f a = [] a`
  - `f a = Maybe a`
  - `f a = (Either b) a`
  - `f a = Parser a`

Functor examples

- Functor laws:
  - `fmap id = id`
  - `fmap (f . g) = fmap f . fmap g`
  - Functors are unique!
- Examples of functors:
  `fmap :: Functor f => (a -> b) -> f a -> f b`
  - `f a = [] a`
  - `f a = Maybe a`
  - `f a = (Either b) a`
  - `f a = Parser a`
  - `f a = ((->) b) a`

Functor on functions

```
instance Functor (->) c where
  fmap f g = ...
```

- Let's derive it from the types!

Functor on functions

```
instance Functor (->) c where
  fmap f g = ...
```

- Let's derive it from the types!
  - `fmap :: (a -> b) -> ((->) c a) -> ((->) c b)`

Functor on functions

```
instance Functor (->) c where
  fmap f g = ...
```

- Let's derive it from the types!
  - `fmap :: (a -> b) -> ((->) c a) -> ((->) c b)`
  - The type of the function result changes!

Functor on functions

```
instance Functor (->) c where
  fmap f g = ...
```

- Let's derive it from the types!
  - `fmap :: (a -> b) -> ((->) c a) -> ((->) c b)`
  - The type of the function result changes!
  - `fmap f g = (\x -> f (g x))`

Functor on functions

```
instance Functor (->) c where
  fmap f g = ...
```

- Let's derive it from the types!
  - `fmap :: (a -> b) -> ((->) c a) -> ((->) c b)`
  - The type of the function result changes!
  - `fmap f g = (\x -> f (g x))`
  - Do you recognise this?

Functor on functions

```
instance Functor (->) c where
  fmap f g = ...
```

- Let's derive it from the types!
  - `fmap :: (a -> b) -> ((->) c a) -> ((->) c b)`
  - The type of the function result changes!
  - `fmap f g = (\x -> f (g x))`
  - Do you recognise this?
- Definition:

```
Functor (->) c where
  fmap f g = (.)
```

Applicative Functors

Applicative Functor

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
```

- `(,,) $ 1 2 3`
- `(,,) <$> (Just 1) <*> (Just 2) <*> (Just 3)`
- Partial function application for Functors
- Unlike Functor: not unique!

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g     = ...
```

- Let's derive it from the types!

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g     = ...
```

- Let's derive it from the types!
- `pure x = ...`
  - `pure :: a -> ((->) b a)`

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g     = ...
```

- Let's derive it from the types!
- `pure x = ...`
  - `pure :: a -> ((->) b a)`
  - Results in a function from `b` to `a`

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g     = ...
```

- Let's derive it from the types!
- `pure x = ...`
  - `pure :: a -> ((->) b a)`
  - Results in a function from `b` to `a`
  - How to make such function for *every* `a`?

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g     = ...
```

- Let's derive it from the types!
- `pure x = ...`
  - `pure :: a -> ((->) b a)`
  - Results in a function from `b` to `a`
  - How to make such function for *every* `a`?
  - `pure x = (\y -> x) -- constant function`

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g     = ...
```

- Let's derive it from the types!
- `pure x = ...`
  - `pure :: a -> ((->) b a)`
  - Results in a function from `b` to `a`
  - How to make such function for *every* `a`?
  - `pure x = (\y -> x) -- constant function`
- `f <*> g = ...`
  -

```
(<*>) :: (->) c (a -> b) -> ((->) c a) -> ((->) c b)
```

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g  = ...
```

- Let's derive it from the types!
- `pure x = ...`
  - `pure :: a -> ((->) b a)`
  - Results in a function from `b` to `a`
  - How to make such function for *every* `a`?
  - `pure x = (\y -> x) -- constant function`
- `f <*> g = ...`
  -
    ```
    (<*>) :: (->) c (a -> b) -> ((->) c a) -> ((->) c b)
    ```
  - `(<*>) :: (c -> (a -> b)) -> (c -> a) -> (c -> b)`
    - What if all functions are applied (i.e., remove `c`)

Applicative on functions (1/2)

```
instance Applicative (->) c where
  pure x      = ...
  f <*> g  = ...
```

- Let's derive it from the types!
- `pure x = ...`
  - `pure :: a -> ((->) b a)`
  - Results in a function from `b` to `a`
  - How to make such function for *every* `a`?
  - `pure x = (\y -> x) -- constant function`
- `f <*> g = ...`
  -
    ```
    (<*>) :: (->) c (a -> b) -> ((->) c a) -> ((->) c b)
    ```
  - `(<*>) :: (c -> (a -> b)) -> (c -> a) -> (c -> b)`
    - What if all functions are applied (i.e., remove `c`)
  - `f <*> g = \x -> f x (g x)`

Applicative on functions (2/2)

```
instance Applicative (->) c where
  pure      = \x -> (\y -> x)
  f <*> g   = \x -> f x (g x)
```

- How to use this applicative?

Applicative on functions (2/2)

```
instance Applicative (->) c where
  pure      = \x -> (\y -> x)
  f <*> g   = \x -> f x (g x)
```

- How to use this applicative?
  - `(,) <$> f <*> g == \x -> (,) (f x) (g x)`

Applicative on functions (2/2)

```
instance Applicative (->) c where
  pure      = \x -> (\y -> x)
  f <*> g   = \x -> f x (g x)
```

- How to use this applicative?
  - `(,) <$> f <*> g == \x -> (,) (f x) (g x)`
  - `f <$> g <*> h <*> ... == \x -> f (g x) (h x) ...`

Applicative on functions (2/2)

```
instance Applicative (->) c where
  pure      = \x -> (\y -> x)
  f <*> g   = \x -> f x (g x)
```

- How to use this applicative?
  - `(,) <$> f <*> g == \x -> (,) (f x) (g x)`
  - `f <$> g <*> h <*> ... == \x -> f (g x) (h x) ...`

- Example: `prop_sort xs = mysort xs == sort xs`

Applicative on functions (2/2)

```
instance Applicative (->) c where
  pure      = \x -> (\y -> x)
  f <*> g   = \x -> f x (g x)
```

- How to use this applicative?
  - `(,) <$> f <*> g == \x -> (,) (f x) (g x)`
  - `f <$> g <*> h <*> ... == \x -> f (g x) (h x) ...`

- Example: `prop_sort xs = mysort xs == sort xs`
-
```
prop_sort' = (==) <$> sort <*> mysort

mkprop :: (Eq b) => (a->b) -> (a->b) -> (a->Bool)
mkprop f g = (==) <$> f <*> g
```

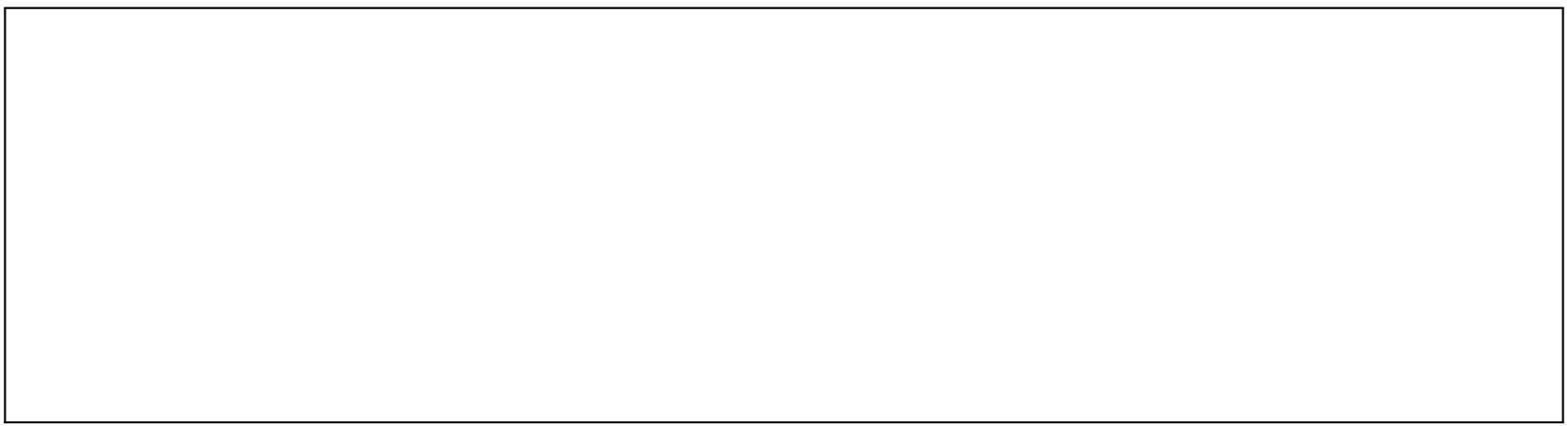
QuickCheck

QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator

QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator
- Example:



QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator
- Example:

```
*QCTest> generate (arbitrary ::  Gen Int)
```

QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator
- Example:

```
*QCTest> generate (arbitrary ::  Gen Int)
-11
```

QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator
- Example:

```
*QCTest> generate (arbitrary ::  Gen Int)
-11
*QCTest> generate (arbitrary ::  Gen String)
```

QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator
- Example:

```
*QCTest> generate (arbitrary ::  Gen Int)
-11
*QCTest> generate (arbitrary ::  Gen String)
"pl>[&"
```

QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator
- Example:

```
*QCTest> generate (arbitrary ::  Gen Int)
-11
*QCTest> generate (arbitrary ::  Gen String)
"pl>[&"
*QCTest> generate (arbitrary ::  Gen (Int,Char))
```

QuickCheck generators

- `Gen a` is a QuickCheck generator for a "value" of type `a`
- `arbitrary :: Arbitrary a => Gen a`
  - gives a generator of type `a`
  - `generate :: Gen a -> IO a` uses the generator
- Example:

```
*QCTest> generate (arbitrary ::  Gen Int)
-11
*QCTest> generate (arbitrary ::  Gen String)
"pl>[&"
*QCTest> generate (arbitrary ::  Gen (Int,Char))
(0,'2')
```

Arbitrary type class

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

- Required: `arbitrary` to create an arbitrary (random) value
- Functions you may use to define `arbitrary`:
  - `oneof`
  - `frequency`
  - `sized` and `resize`
  - `arbitrary`

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
```

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
-25
```

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
-25
QC> data Test a = Test a
```

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
-25
QC> data Test a = Test a
QC> generate (Test <$> (arbitrary ::  Gen Int))
```

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
-25
QC> data Test a = Test a
QC> generate (Test <$> (arbitrary ::  Gen Int))
Test 6
```

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
-25
QC> data Test a = Test a
QC> generate (Test <$> (arbitrary ::  Gen Int))
Test 6
QC> data Tuple a b = Tuple a b deriving Show
```

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
-25
QC> data Test a = Test a
QC> generate (Test <$> (arbitrary ::  Gen Int))
Test 6
QC> data Tuple a b = Tuple a b deriving Show
QC> generate (Tuple <$> (arbitrary ::  Gen Int) <*>
(arbitrary ::  Gen Char))
```

`Arbitrary` as Applicative Functor

```
Prelude Test.QuickCheck> generate (arbitrary ::  Gen Int)
-25
QC> data Test a = Test a
QC> generate (Test <$> (arbitrary ::  Gen Int))
Test 6
QC> data Tuple a b = Tuple a b deriving Show
QC> generate (Tuple <$> (arbitrary ::  Gen Int) <*>
(arbitrary ::  Gen Char))
Tuple 6 'H'
```

QuickCheck: anyof

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

QuickCheck: anyof

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

```
data Direction = East | West | North | South
               deriving Show
```

QuickCheck: anyof

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

```
data Direction = East | West | North | South
               deriving Show
```

```
instance Arbitrary Direction where
  -- all probability of 1/4
  arbitrary = oneof [ pure East,
                      pure West,
                      pure North,
                      pure South ]
```

QuickCheck: `frequency`

```
frequency :: [(Int, Gen a)] -> Gen a
```

QuickCheck: frequency

```
frequency :: [(Int, Gen a)] -> Gen a
```

```
data MyList a = Nil
              | Cons a (MyList a)
               deriving Show
```

QuickCheck: frequency

```haskell
frequency :: [(Int, Gen a)] -> Gen a
```

```haskell
data MyList a = Nil
              | Cons a (MyList a)
               deriving Show
```

```haskell
instance Arbitrary a => Arbitrary (MyList a) where
  arbitrary = frequency
    [  -- probability of 1/3
      (1, pure Nil),
       -- probability of 2/3
      (2, Cons <$> arbitrary <*> arbitrary) ]
```

QuickCheck: generating trees

```
data BinTree a = Leaf
               | Node (BinTree a) (BinTree a) a
               deriving Show


instance Arbitrary a => Arbitrary (BinTree a) where
  arbitrary = frequency
   [ (1, pure Leaf),
     (2, Node <$> arbitrary <*> arbitrary <*> arbitrary) ]
```

QuickCheck: generating trees

```
data BinTree a = Leaf
               | Node (BinTree a) (BinTree a) a
               deriving Show


instance Arbitrary a => Arbitrary (BinTree a) where
  arbitrary = frequency
   [ (1, pure Leaf),
     (2, Node <$> arbitrary <*> arbitrary <*> arbitrary) ]
```

Incorrect: high probability that this does not terminate

QuickCheck: generating trees (correct)

```
sized :: (Int -> Gen a) -> Gen a
```

QuickCheck: generating trees (correct)

```
sized :: (Int -> Gen a) -> Gen a

data BinTree a = Leaf
               | Node (BinTree a) (BinTree a) a
               deriving Show
```

QuickCheck: generating trees (correct)

```
sized :: (Int -> Gen a) -> Gen a

data BinTree a = Leaf
               | Node (BinTree a) (BinTree a) a
               deriving Show

instance Arbitrary a => Arbitrary (BinTree a) where
  arbitrary = sized arbtree

arbtree :: (Arbitrary a) => Int -> Gen (BinTree a)
arbtree 0 = pure Leaf
arbtree n = frequency
   [ (1, pure Leaf),
     (2, Node <$> (arbtree (n `div` 2))
              <*> (arbtree (n `div` 2))
              <*> arbitrary) ]
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
[13,-18,18,-5,-2,17,-14,-14,-11,26,-27,-10,27,-23,-27,-16,-9]
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
[13,-18,18,-5,-2,17,-14,-14,-11,26,-27,-10,27,-23,-27,-16,-9]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
[13,-18,18,-5,-2,17,-14,-14,-11,26,-27,-10,27,-23,-27,-16,-9]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1,-2]
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
[13,-18,18,-5,-2,17,-14,-14,-11,26,-27,-10,27,-23,-27,-16,-9]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1,-2]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
[13,-18,18,-5,-2,17,-14,-14,-11,26,-27,-10,27,-23,-27,-16,-9]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1,-2]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1]
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
[13,-18,18,-5,-2,17,-14,-14,-11,26,-27,-10,27,-23,-27,-16,-9]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1,-2]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
```

QuickCheck: `resize`

```
resize :: Int -> Gen a -> Gen a
```

```
*QCTest> generate (arbitrary ::  Gen [Int])
[-14,-26,-2,5,28,-13,23,26,1,-9,22,-12]
*QCTest> generate (arbitrary ::  Gen [Int])
[13,-18,18,-5,-2,17,-14,-14,-11,26,-27,-10,27,-23,-27,-16,-9]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1,-2]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[1]
*QCTest> generate (resize 2 $ arbitrary ::  Gen [Int])
[-2,-1]
```

Monoids

Reductions

- `sum [1,2,3,4] == 1 + 2 + 3 + 4`
- `product [1,2,3,4] == 1 * 2 * 3 * 4`
- `concat ["ab", "cde", ""] == "ab" ++ "cde" ++ ""`
- `mconcat [Just "a",Nothing,Just "bc"]==Just "abc"`
- Note:
  - Order of operations does not matter: `(1+2)+3 = 1+(2+3)`, ...
  - Identity operations:
    - `x+0==x`
    - `x*1==x`
    - `x++[]==x`
    - `x <> Nothing == x`

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`
- Expresses combining two values(/objects) to a single

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`
- Expresses combining two values(/objects) to a single
- Infix operator <>
  - `mappend x y == x <> y`

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                 -- identity element e
  mappend :: a -> a -> a      -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`
- Expresses combining two values(/objects) to a single
- Infix operator <>
  - `mappend x y == x <> y`
- `mconcat [x1,x2,...] == mempty <> x1 <> x2 <> ...`

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                 -- identity element e
  mappend :: a -> a -> a      -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`
- Expresses combining two values(/objects) to a single
- Infix operator `<>`
  - `mappend x y == x <> y`
- `mconcat [x1,x2,...] == mempty <> x1 <> x2 <> ...`
- Required properties:
  - Associative: `a <> (b <> c) == (a <> b) <> c`
  - Identity: `a <> mempty == mempty <> a = a`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?
  - `mempty = \x -> mempty -- constant function`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?
  - `mempty = \x -> mempty -- constant function`
- `mappend f g = ...`
  - `mappend :: (Monoid b) => (a->b)->(a->b)->(a->b)`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?
  - `mempty = \x -> mempty -- constant function`
- `mappend f g = ...`
  - `mappend :: (Monoid b) => (a->b)->(a->b)->(a->b)`
  - Note: `b` values can be combined

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?
  - `mempty = \x -> mempty -- constant function`
- `mappend f g = ...`
  - `mappend :: (Monoid b) => (a->b)->(a->b)->(a->b)`
  - Note: `b` values can be combined
  - How to create a new function from two functions?

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?
  - `mempty = \x -> mempty -- constant function`
- `mappend f g = ...`
  - `mappend :: (Monoid b) => (a->b)->(a->b)->(a->b)`
  - Note: `b` values can be combined
  - How to create a new function from two functions?
  - `mappend f g = \x -> f x <> g x`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?
  - `mempty = \x -> mempty -- constant function`
- `mappend f g = ...`
  - `mappend :: (Monoid b) => (a->b)->(a->b)->(a->b)`
  - Note: `b` values can be combined
  - How to create a new function from two functions?
  - `mappend f g = \x -> f x <> g x`
- Test it: `(reverse <> id) "abc" == "cbaabc"`

Example: Monoid on functions

```
instance (Monoid b) => Monoid ((->) a b) where
  mempty = ...
  mappend f g = ...
```

- Let's derive it from the types!
- `mempty = ...`
  - `mempty :: (Monoid b) => a -> b`
  - Results in a function from `a` to `b`
  - How to make such function for *every* `b`?
  - `mempty = \x -> mempty -- constant function`
- `mappend f g = ...`
  - `mappend :: (Monoid b) => (a->b)->(a->b)->(a->b)`
  - Note: `b` values can be combined
  - How to create a new function from two functions?
  - `mappend f g = \x -> f x <> g x`
- Test it: `(reverse <> id) "abc" == "cbaabc"`

Available in Haskell; note that this is not the Endo Monoid!

Quiz

Question 1

Recall the functor instance for tuples:

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

How to change it such that maps over the *first* element of the tuple?

A Define `fmap` as `fmap f (a,b) = (f a, b)`

B Change
`instance Functor ((,) a) where`
to
`instance Functor ((,a)) where`

C Both of the suggested changes are required

D This is not possible

Question 2

The function
`l :: [String] -> IO [Int]`
defines an action to determine the file lengths that correspond to a list of filenames.

How is the base case of this function defined?

A `l [] = []`

B `l [] = pure []`

C `l [] = IO []`

D `l [] = fmap pure`

Question 3

The function
`l :: [String] -> IO [Int]`
defines an action to determine the file lengths that correspond to a list of filenames.

How is the recursive case of this function defined?

A `l (x:xs) = (length <$> readFile xs) : (l xs)`

B `l (x:xs) = (:) <$> (length $ readFile x) <*> l xs`

C `l (x:xs) = (length $ readFile xs) : (l xs)`

D `l (x:xs) = (:) <$> (length<$>readFile x) <*> l xs`

Question 4

Why is the following Monoid definition incorrect?

```
instance Monoid Bool where
    mempty = False
    mappend True  True  = True
    mappend False True  = False
    mappend True  False = True
    mappend False False = False
```

A `Bool` is not of the correct kind, namely `* -> *`

B The binary operation is not associative

C The binary operation is not commutative

D `mempty` is not an identity element

Question 5

Consider the following definition:

```
name :: IO String
name = getLine
```

How can we define a function `get :: IO String -> String` to retrieve the name that was read?

A `get (IO x) = x`

B `get x = pure id <$> x`

C Using Monads, which we did not see in this course :-(

D This is impossible since there is no name read

Question 6

The function `rep :: Int -> Parser a -> Parser [a]` receives a number `n` and a parser `p`, and parses `n` times consecutively using `p`.

How can we define this parser?

A
```
rep 0 p = []
rep n p = p : rep (n-1) p
```

B
```
rep 0 p = pure []
rep n p = p : rep (n-1) p
```

C
```
rep 0 p = []
rep n p = (:) <$> p <*> rep (n-1) p
```

D
```
rep 0 p = pure []
rep n p = (:) <$> p <*> rep (n-1) p
```

Remaining lectures

Remaining lectures

- 23-05: Introduction “Project Functional Programming”
  - Project: covers all material
  - Lecture to introduce the assignment

Remaining lectures

- 23-05: Introduction “Project Functional Programming”
  - Project: covers all material
  - Lecture to introduce the assignment
- 13-06: Q&A Functional Programming
  - Last opportunity to ask questions
  - Send me an e-mail before 12-06 with requests for additional explanations of the material

Remaining lectures

- 23-05: Introduction “Project Functional Programming”
  - Project: covers all material
  - Lecture to introduce the assignment
- 13-06: Q&A Functional Programming
  - Last opportunity to ask questions
  - Send me an e-mail before 12-06 with requests for additional explanations of the material
- 14-06: Exam Functional Programming