Quick Check

A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen, John Hughes

Verification versus Validation

# We want a program to be correct.

# Problem: To verify it, we need specifications.

# We can validate it by testing it.

# In Haskell, testing is quite efficient, because of purity.
(When every function is correct and has no side-effects, the whole program will be correct)

Example

```
fac_naive n
    | n<2              = 1
    |otherwise  =  n * fac_naive (n-1)

fac n = foldr (*) 1 [0..n]

prop_fac          :: Int -> Bool
prop_fac x        = fac x == fac_naive x
```

```
Main> quickCheck prop_fac
Falsifiable, after 1 tests:
1

Main> fac 1
0
```

Example

```
fac_naive n
    | n<2            = 1
    |otherwise  =  n * fac_naive (n-1)

fac n = foldr (*) 1 [1..n]

prop_fac        :: Int -> Bool
prop_fac x      = fac x == fac_naive x


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

How to generate test data?

```
                       (α → Bool)
Main> quickCheck property
```

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

# Bool:

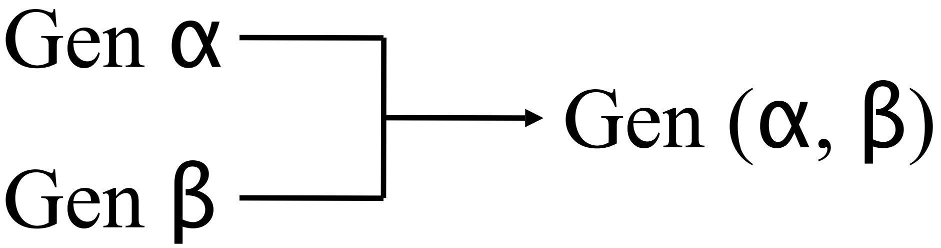
```
instance Arbitrary Bool where
   arbitrary      = elements [True, False]
```

# Int:

```
instance Arbitrary Int where
   arbitrary      = choose (–1000, 1000)
```

# Int → (Int → Bool) → [Char] → Int

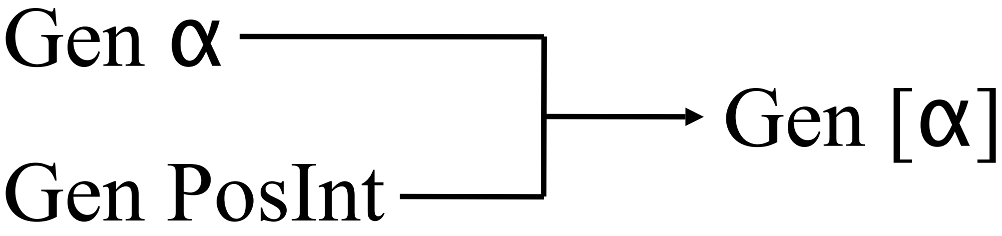
Generating more complex data



Gen α

Gen β

Gen (α, β)



Gen α

Gen PosInt

Gen [α]

choose (0, 100)

Combinators

return	:: α → Gen α
elements	:: [α] → Gen α
choose	:: (Int, Int) → Gen Int
oneof	:: [Gen α] → Gen α
frequency	:: [(Int, Gen α)] → Gen α
sized	:: (Int → Gen α) → Gen α

Generating user defined data

```
data Colour = Red | Blue | Green

instance Arbitrary Colour where
arbitrary =  oneof [return Red,return Blue, return Green]

data Tree a = L a | T (Tree a) (Tree a)

instance Arbitrary a => instance Arbitrary Tree a where
arbitrary = oneof [liftM L arbitrary,
                         liftM2 T arbitrary arbitrary]
```

```
return  :: a -> Gen a
oneof   :: [Gen a] -> Gen a
liftM   :: (a -> t) -> Gen a -> Gen t
liftM2  :: (a -> b -> t) -> Gen a -> Gen b -> Gen t
```

Generating user defined data

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

```
data Tree a = L a | T (Tree a) (Tree a)
```

```
instance Arbitrary a => instance Arbitrary Tree a where
arbitrary = oneof [liftM L arbitrary,
                   liftM2 T arbitrary arbitrary]
```

Generating user defined data

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

```
data Tree a = L a | T (Tree a) (Tree a)
```

```
instance Arbitrary a => instance Arbitrary Tree a where
arbitrary =  frequency [(1, liftM L arbitrary),
                        (2, liftM2 T arbitrary arbitrary)]
```

Generating user defined data

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

data Tree a = L a | T (Tree a) (Tree a)

instance Arbitrary a => instance Arbitrary Tree a where
arbitrary =  sized arbTree


arbTree         :: Int -> Gen a
arbTree 0       = liftM L arbitrary
arbTree n       = frequency [(1, liftM L arbitrary),
            (2, liftM2 T (arbTree (n `div` 2))
                            (arbTree (n `div` 2)) ) ]
```

What about functions?

Generating functions

newtype Gen = Int → Rand → α

Gen (α → β) = Int → Rand → α → β

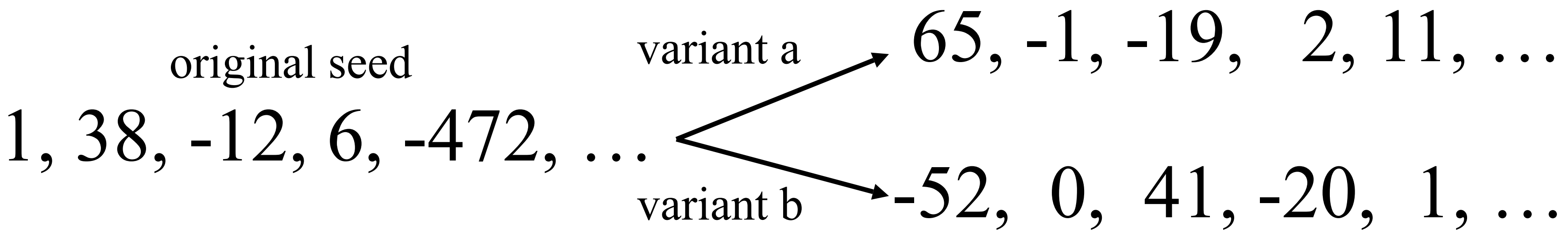
α → Gen β =α → Int → Rand → β

promote :: (α → Gen β) → Gen (α → β)

Modifying the Random Number Seed

We need a function: α → Gen β

We have: variant :: Int → Gen α → Gen α



original seed

1, 38, -12, 6, -472, …

variant a

65, -1, -19, 2, 11, …

variant b

-52, 0, 41, -20, 1, …

How does variant solve our problem?

Coarbitrary

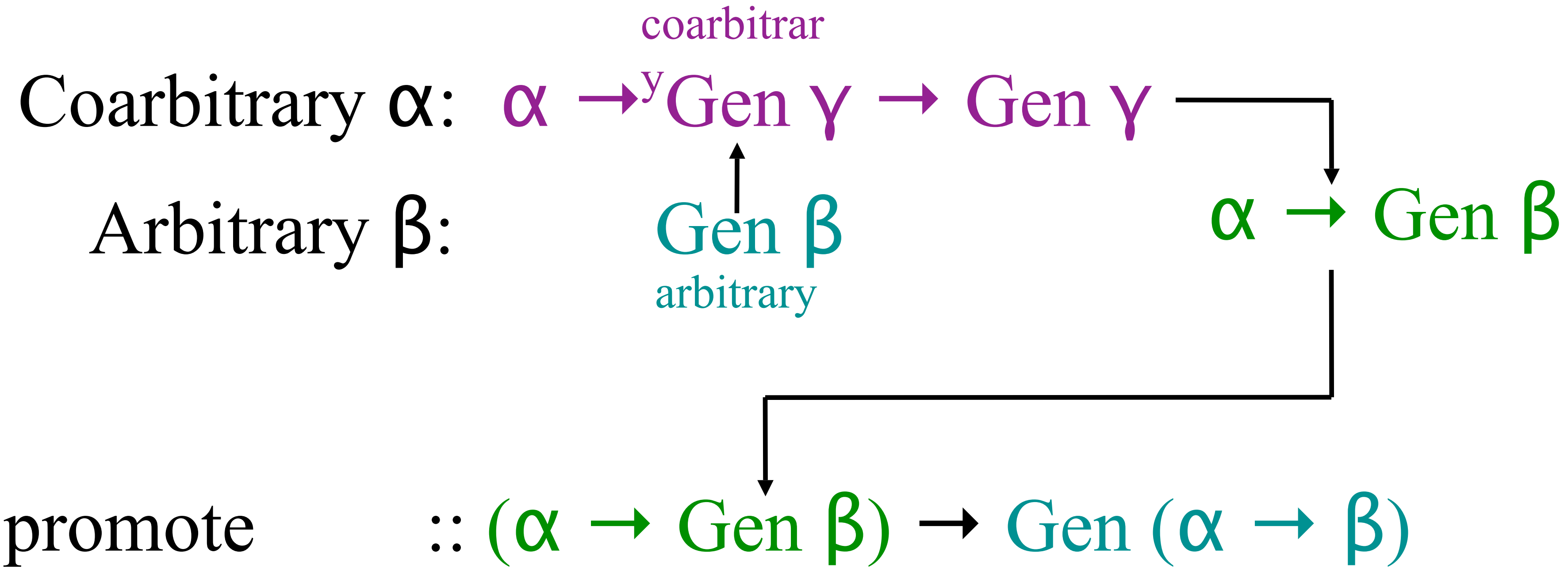
We still need a function: α → Gen β

```
variant      :: Int → Gen α → Gen α
coarbitrary  :: α → Gen β → Gen β
```

Bool:

```
instance Coarbitrary Bool where
   coarbitrary b  g =
         if b then variant 0 g else variant 1 g
```

Putting the stuff together



coarbitrar

Coarbitrary α: α →y Gen γ → Gen γ

Arbitrary β: Gen β

arbitrary

α → Gen β

promote :: (α → Gen β) → Gen (α → β)

```
instance (Coarbitrary a, Arbitrary b) => Arbitrary (a -> b) where
  arbitrary = promote  (\x -> coarbitrary x arbitrary)
```

Gen (α → (α) (Gen β)

3 kinds of errors:

- Errors in the test data generator
  - Diverging Generators
  - Generators that produce nonsense
- Errors in the program
  - `fac n = foldr (*) 1 [0..n]`
- Errors in the specification
  - Ill-defined properties
  - Missunderstanding of the code

Monitoring Test Data

```
prop_fac       :: Int -> Property
prop_fac x     = classify (x `mod` 2 == 0) „even“
                   (fac x == fac_naive x)
```

```
Main> quickCheck prop_fac
OK, passed 100 tests (52% even).
```

```
prop_fac       :: Int -> Property
prop_fac x     = collect (x `mod` 3) (fac x == fac_naive x)
```

```
Main> quickCheck prop_fac
OK, passed 100 tests.
38% 2.
27% 0.
25% 1.
```

Advanced Properties

```
prop_fac        :: Int -> Property
prop_fac x      = x < 1 ==> fac x == 1

prop_fac        :: Property
prop_fac        = forAll niceInt (\x ->  fac x == fac_naive x)
```

The trivial data Problem

```
Prop_Insert      :: Int -> [Int] -> Property
Prop_Insert x xs = ordered xs ==> ordered (insert x xs)
```

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

The trivial data Problem

```
Prop_Insert          :: Int -> [Int] -> Property
Prop_Insert x xs = ordered xs ==> classify (length xs < 3)
                           „trivial“ (ordered (insert x xs))
```

```
Main> quickCheck prop_Insert
OK, passed 100 tests (95% trivial).
```