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 →

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

\[
\text{Gen } \alpha 
\quad \rightarrow \quad \text{Gen } (\alpha, \beta) 
\]

\[
\text{Gen } \beta 
\quad \rightarrow \quad \text{Gen } [\alpha] 
\]

choose (0, 100)
Combinators

- **return** :: $\alpha \rightarrow \text{Gen } \alpha$
- **elements** :: $[\alpha] \rightarrow \text{Gen } \alpha$
- **choose** :: $(\text{Int}, \text{Int}) \rightarrow \text{Gen } \text{Int}$
- **oneof** :: $[\text{Gen } \alpha] \rightarrow \text{Gen } \alpha$
- **frequency** :: $[(\text{Int}, \text{Gen } \alpha)] \rightarrow \text{Gen } \alpha$
- **sized** :: $(\text{Int} \rightarrow \text{Gen } \alpha) \rightarrow \text{Gen } \alpha$
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

\[
\begin{align*}
\text{return} & : a \rightarrow \text{Gen} \ a \\
\text{oneof} & : [\text{Gen} \ a] \rightarrow \text{Gen} \ a \\
\text{frequency} & : [(\text{Int}, \text{Gen} \ a)] \rightarrow \text{Gen} \ a
\end{align*}
\]

data Tree a = L a \mid T (\text{Tree} \ a) \ (\text{Tree} \ a)

instance Arbitrary a \Rightarrow \text{instance Arbitrary} \ Tree \ a \ \text{where}
\begin{align*}
\text{arbitrary} & = \ \text{oneof} \ [\text{liftM} \ L \ \text{arbitrary}, \\
& \text{liftM2} \ T \ \text{arbitrary} \ \text{arbitrary}]
\end{align*}
Generating user defined data

\[
\begin{align*}
\text{return} & : a \rightarrow \text{Gen} \ a \\
\text{oneof} & : [\text{Gen} \ a] \rightarrow \text{Gen} \ a \\
\text{frequency} & : [(\text{Int}, \text{Gen} \ a)] \rightarrow \text{Gen} \ a \\
\text{sized} & : (\text{Int} \rightarrow \text{Gen} \ a) \rightarrow \text{Gen} \ a
\end{align*}
\]

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

\[
\text{Gen} \ (\alpha \to \beta) \ = \ \text{Int} \to \text{Rand} \to \alpha \to \beta \\
\alpha \to \text{Gen} \ \beta \ = \ \alpha \to \text{Int} \to \text{Rand} \to \beta
\]

promote :: (\alpha \to \text{Gen} \ \beta) \to \text{Gen} \ (\alpha \to \beta)
Modifying the Random Number Seed

We need a function: \( \alpha \rightarrow \text{Gen} \)

\( \beta \)

We have: variant :: Int \( \rightarrow \) Gen \( \alpha \rightarrow \text{Gen} \alpha \)

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: \( \alpha \rightarrow \text{Gen} \ \beta \)

\text{variant} :: \text{Int} \rightarrow \text{Gen} \ \alpha \rightarrow \text{Gen} \ \alpha

\text{coarbitrary} :: \alpha \rightarrow \text{Gen} \ \beta \rightarrow \text{Gen} \ \beta

\text{Bool:}

\text{instance Coarbitrary Bool where}

\text{coarbitrary} \ b \ g =

if \ b \ \text{then} \ \text{variant} \ 0 \ g \ \text{else} \ \text{variant} \ 1 \ g
Putting the stuff together

Coarbitrary $\alpha$: $\alpha \rightarrow^y \text{Gen } \gamma \rightarrow \text{Gen } \gamma$

Arbitrary $\beta$:

promote :: $(\alpha \rightarrow \text{Gen } \beta) \rightarrow \text{Gen } (\alpha \rightarrow \beta)$

instance (Coarbitrary a, Arbitrary b) => Arbitrary (a -> b) where

arbitrary = promote $(\lambda x \rightarrow \text{coarbitrary}_x \times \text{arbitrary})$
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).