Functional Programming

QuickCheck: Automated Random Testing

Jevgeni Kabanov

Department of Computer Science
University of Tartu

Random Testing

Outline

- Testing is very important in programming
- In JUnit and alike we collect test cases that are prearranged argument-result pairs
- In Haskell there is HUnit which does the same
- However we could do better
  - Since functions are pure we can test them against properties
  - Since data types are structural we can try generating random data samples
- Random testing enjoys some of the benefits of formal verification without nearly as much pain!

Random Testing

Outline

- Testing is very important in programming
- In JUnit and alike we collect test cases that are prearranged argument-result pairs
- In Haskell there is HUnit which does the same
- However we could do better
  - Since functions are pure we can test them against properties
  - Since data types are structural we can try generating random data samples
- Random testing enjoys some of the benefits of formal verification without nearly as much pain!

Random Testing

Outline

- Testing is very important in programming
- In JUnit and alike we collect test cases that are prearranged argument-result pairs
- In Haskell there is HUnit which does the same
- However we could do better
  - Since functions are pure we can test them against properties
  - Since data types are structural we can try generating random data samples
- Random testing enjoys some of the benefits of formal verification without nearly as much pain!

Random Testing

Outline

- Testing is very important in programming
- In JUnit and alike we collect test cases that are prearranged argument-result pairs
- In Haskell there is HUnit which does the same
- However we could do better
  - Since functions are pure we can test them against properties
  - Since data types are structural we can try generating random data samples
- Random testing enjoys some of the benefits of formal verification without nearly as much pain!

Random Testing

Outline

- Testing is very important in programming
- In JUnit and alike we collect test cases that are prearranged argument-result pairs
- In Haskell there is HUnit which does the same
- However we could do better
  - Since functions are pure we can test them against properties
  - Since data types are structural we can try generating random data samples
- Random testing enjoys some of the benefits of formal verification without nearly as much pain!

Random Testing

Outline

- Testing is very important in programming
- In JUnit and alike we collect test cases that are prearranged argument-result pairs
- In Haskell there is HUnit which does the same
- However we could do better
  - Since functions are pure we can test them against properties
  - Since data types are structural we can try generating random data samples
- Random testing enjoys some of the benefits of formal verification without nearly as much pain!

Random Testing

Outline

- Testing is very important in programming
- In JUnit and alike we collect test cases that are prearranged argument-result pairs
- In Haskell there is HUnit which does the same
- However we could do better
  - Since functions are pure we can test them against properties
  - Since data types are structural we can try generating random data samples
- Random testing enjoys some of the benefits of formal verification without nearly as much pain!

reverse examples

Property

We concatenated in a wrong order:

$$
\begin{aligned}
&\mathit{propRevApp2} :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Bool}\\
&\mathit{propRevApp2}\ \mathit{xs}\ \mathit{ys} =\\
&\quad \mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{reverse}\ \mathit{xs}
\end{aligned}
$$

Output

```
Test> quickCheck propRevApp2
OK, passed 100 tests.
```

reverse examples

Property

We concatenated in a wrong order:

$$
\begin{array}{l}
\mathit{propRevApp2} :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Bool} \\
\mathit{propRevApp2}\ \mathit{xs}\ \mathit{ys} = \\
\quad \mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{reverse}\ \mathit{xs}
\end{array}
$$

Output

```
Test> quickCheck propRevApp2
OK, passed 100 tests.
```

reverse examples

Property

Let's check if you can reverse before concatenating:

$$
\begin{aligned}
&\mathit{propRevApp1} :: [\mathit{Int}] \rightarrow [\mathit{Int}] \rightarrow \mathit{Bool} \\
&\mathit{propRevApp1}\ \mathit{xs}\ \mathit{ys} = \\
&\quad \mathit{reverse}\ (\mathit{xs} +\!\!+\ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{xs} +\!\!+\ \mathit{reverse}\ \mathit{ys}
\end{aligned}
$$

Output

```
Test> quickCheck propRevApp1
Falsifiable, after 4 tests:
[-3,-4,-4]
[-4,-1,1,1]
```

reverse examples

Property

Let's check if you can reverse before concatenating:

$$
\begin{aligned}
&\mathit{propRevApp1} :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Bool} \\
&\mathit{propRevApp1}\ \mathit{xs}\ \mathit{ys} = \\
&\quad \mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{xs} +\!\!+ \mathit{reverse}\ \mathit{ys}
\end{aligned}
$$

Output

```
Test> quickCheck propRevApp1
Falsifiable, after 4 tests:
[-3,-4,-4]
[-4,-1,1,1]
```

Distribution examples

Property

The following property asserts that addition and multiplication distribute:

$$\begin{aligned}
&\mathit{propDistributiveI} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Bool} \\
&\mathit{propDistributiveI}\ a\ b\ c = \\
&\quad a * (b + c) \equiv (a * b) + (a * c)
\end{aligned}$$

Output

```
Test> propDistributiveI
OK, passed 100 tests.
```

Distribution examples

Property

The following property asserts that addition and multiplication distribute:

$$\begin{aligned}
&\mathit{propDistributiveI} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool} \\
&\mathit{propDistributiveI}\ a\ b\ c = \\
&\quad a * (b + c) \equiv (a * b) + (a * c)
\end{aligned}$$

Output

```
Test> propDistributiveI
OK, passed 100 tests.
```

Distribution examples

Property

The same property for *Floats* fails:

$$propDistributiveF :: Float \to Float \to Float \to Bool$$
$$propDistributiveF\ a\ b\ c =$$
$$a * (b + c) \equiv (a * b) + (a * c)$$

Output

```
Test> quickCheck propDistributiveF
Falsifiable, after 7 tests:
3.0
-2.666667
3.75
```

Distribution examples

Property

The same property for *Floats* fails:

$$propDistributiveF :: Float \rightarrow Float \rightarrow Float \rightarrow Bool$$
$$propDistributiveF\ a\ b\ c =$$
$$a * (b + c) \equiv (a * b) + (a * c)$$

Output

```
Test> quickCheck propDistributiveF
Falsifiable, after 7 tests:
3.0
-2.666667
3.75
```

insert and *ordered*

Definition

For the next several slides we will consider a function which inserts an element into an ordered list.

$$
\begin{aligned}
&\mathit{insert}\ e\ (x : xs) = \\
&\quad \textbf{if}\ e < x\ \textbf{then}\ e : x : xs\ \textbf{else}\ x : (\mathit{insert}\ e\ xs) \\
&\mathit{insert}\ e\ [\,] = [\,e\,]
\end{aligned}
$$

ordered tests whether the list is ordered:

$$
\begin{aligned}
&\mathit{ordered} :: \mathit{Ord}\ a \Rightarrow [\,a\,] \rightarrow \mathit{Bool} \\
&\mathit{ordered}\ [\,] = \mathit{True} \\
&\mathit{ordered}\ (x : [\,]) = \mathit{True} \\
&\mathit{ordered}\ (x1 : x2 : xs) = \\
&\quad \textbf{if}\ x1 \leqslant x2\ \textbf{then}\ \mathit{ordered}\ (x2 : xs) \\
&\quad \textbf{else}\ \mathit{False}
\end{aligned}
$$

insert examples

Property

We would want to test whether *insert* works, but this has point only on ordered lists:

$$
\begin{aligned}
&\mathit{propInsert1} :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Bool}\\
&\mathit{propInsert1}\ x\ \mathit{xs} =\\
&\quad \textbf{if}\ \mathit{ordered}\ \mathit{xs}\\
&\qquad \textbf{then}\ \mathit{ordered}\ (\mathit{insert}\ x\ \mathit{xs})\\
&\qquad \textbf{else}\ \mathit{True}
\end{aligned}
$$

Since QuickCheck does not work on polymorphic types we choose *Int*s here.

Output

```
Test> propInsert1
OK, passed 100 tests.
```

insert examples

Property

We would want to test whether *insert* works, but this has point only on ordered lists:

$$
\begin{array}{l}
\mathit{propInsert1} :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Bool} \\
\mathit{propInsert1}\ x\ xs = \\
\quad \textbf{if}\ \mathit{ordered}\ xs \\
\qquad \textbf{then}\ \mathit{ordered}\ (\mathit{insert}\ x\ xs) \\
\qquad \textbf{else}\ \mathit{True}
\end{array}
$$

Since QuickCheck does not work on polymorphic types we choose *Int*s here.

Output

```
Test> propInsert1
OK, passed 100 tests.
```

insert examples

Property

But did this actually tell us anything? How do we know how many lists were ordered?

$$
\begin{array}{l}
propInsert2 :: Int \to [Int] \to Property \\
propInsert2\ x\ xs = \\
\quad (length\ xs \equiv 0 \lor \neg\,(ordered\ xs))\ `trivial` \\
\quad \textbf{if}\ ordered\ xs \\
\qquad \textbf{then}\ ordered\ (insert\ x\ xs) \\
\qquad \textbf{else}\ True
\end{array}
$$

Output

```
*Test> quickCheck propInsert2
OK, passed 100 tests (82% trivial).
```

insert examples

Property

But did this actually tell us anything? How do we know how many lists were ordered?

$$
\begin{array}{l}
propInsert2 :: Int \to [Int] \to Property \\
propInsert2\ x\ xs = \\
\quad (length\ xs \equiv 0 \lor \neg\,(ordered\ xs))\ `trivial` \\
\quad \textbf{if}\ ordered\ xs \\
\qquad \textbf{then}\ ordered\ (insert\ x\ xs) \\
\qquad \textbf{else}\ True
\end{array}
$$

Output

```
*Test> quickCheck propInsert2
OK, passed 100 tests (82% trivial).
```

insert examples

Property

$==>$ is the QuickCheck combinator that makes it test only the fitting values:

$$
\begin{array}{l}
\mathit{propInsert3} :: \mathit{Int} \rightarrow [\mathit{Int}] \rightarrow \mathit{Property} \\
\mathit{propInsert3}\ x\ \mathit{xs} = \\
\quad \mathit{ordered}\ \mathit{xs} ==> \mathit{ordered}\ (\mathit{insert}\ x\ \mathit{xs})
\end{array}
$$

Output

```
Test> propInsert3
OK, passed 100 tests.
```

insert examples

Property

$==>$ is the QuickCheck combinator that makes it test only the fitting values:

$$
\begin{array}{l}
propInsert3 :: Int \rightarrow [Int] \rightarrow Property \\
propInsert3\ x\ xs = \\
\quad ordered\ xs ==> ordered\ (insert\ x\ xs)
\end{array}
$$

Output

```
Test> propInsert3
OK, passed 100 tests.
```

insert examples

Property

How well do we actually test? Can this pass?

$$
\begin{aligned}
&insBad\ a\ [\,] = [\,a\,] \\
&insBad\ a\ y \\
&\quad \mid (length\ y) > 4 = y + [\,a\,] \\
&\quad \mid otherwise = insert\ a\ y \\
&propInsertBad1 :: Int \rightarrow [\,Int\,] \rightarrow Property \\
&propInsertBad1\ x\ xs = \\
&\quad ordered\ xs \Longrightarrow ordered\ (insBad\ x\ xs)
\end{aligned}
$$

Output

```
Test> quickCheck propInsertBad1
OK, passed 100 tests.
```

insert examples

Property

How well do we actually test? Can this pass?

$$
\begin{aligned}
&\mathit{insBad}\ a\ [\,] = [\,a\,]\\
&\mathit{insBad}\ a\ y\\
&\quad \mid (\mathit{length}\ y) > 4 = y +\!\!+ [\,a\,]\\
&\quad \mid \mathit{otherwise} = \mathit{insert}\ a\ y\\
&\mathit{propInsertBad1} :: \mathit{Int} \rightarrow [\,\mathit{Int}\,] \rightarrow \mathit{Property}\\
&\mathit{propInsertBad1}\ x\ \mathit{xs} =\\
&\quad \mathit{ordered}\ \mathit{xs} \Longrightarrow \mathit{ordered}\ (\mathit{insBad}\ x\ \mathit{xs})
\end{aligned}
$$

Output

```
Test> quickCheck propInsertBad1
OK, passed 100 tests.
```

insert examples

Property

$$
\begin{array}{l}
propInsertBad2 :: Int \rightarrow [Int] \rightarrow Property \\
propInsertBad2\ x\ xs = \\
\quad ordered\ xs ==> \\
\qquad collect\ (length\ xs)\ \$\ ordered\ (insBad\ x\ xs)
\end{array}
$$

Output

```
Test> quickCheck propInsertBad2
OK, passed 100 tests.
53% 0.
24% 1.
14% 2.
8% 3.
1% 4.
```

insert examples

Property

$$
\begin{aligned}
&\mathit{propInsertBad2} :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Property} \\
&\mathit{propInsertBad2}\ x\ \mathit{xs} = \\
&\quad \mathit{ordered}\ \mathit{xs} ==> \\
&\qquad \mathit{collect}\ (\mathit{length}\ \mathit{xs})\ \$\ \mathit{ordered}\ (\mathit{insBad}\ x\ \mathit{xs})
\end{aligned}
$$

Output

```
Test> quickCheck propInsertBad2
OK, passed 100 tests.
53% 0.
24% 1.
14% 2.
8% 3.
1% 4.
```

insert examples

Property

$$
\begin{array}{l}
propInsertBad3 :: Int \to [Int] \to Property \\
propInsertBad3\ x\ xs = \\
\quad ordered\ xs ==> \\
\quad\quad classify\ (ordered\ (x : xs))\ \texttt{"at-head"}\ \$ \\
\quad\quad classify\ (ordered\ (xs +\!\!+ [x]))\ \texttt{"at-tail"}\ \$ \\
\quad\quad\quad ordered\ (insBad\ x\ xs)
\end{array}
$$

Output

```
Test> quickCheck propInsertBad3
OK, passed 100 tests.
53% at-head, at-tail.
20% at-tail.
20% at-head.
```

insert examples

Property

$$
\begin{aligned}
&\mathit{propInsertBad3} :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Property}\\
&\mathit{propInsertBad3}\ x\ xs =\\
&\quad \mathit{ordered}\ xs \Longrightarrow\\
&\qquad \mathit{classify}\ (\mathit{ordered}\ (x : xs))\ \texttt{"at-head"}\ \$\\
&\qquad \mathit{classify}\ (\mathit{ordered}\ (xs +\!\!+ [x]))\ \texttt{"at-tail"}\ \$\\
&\qquad\quad \mathit{ordered}\ (\mathit{insBad}\ x\ xs)
\end{aligned}
$$

Output

```
Test> quickCheck propInsertBad3
OK, passed 100 tests.
53% at-head, at-tail.
20% at-tail.
20% at-head.
```

Generators

Outline

- We test mostly trivial or very simple cases (only one insert in the middle of the list!)
- Just checking whether list is ordered is not enough!
- We need a way to generate ordered lists!

Generators

Outline

- We test mostly trivial or very simple cases (only one insert in the middle of the list!)
- Just checking whether list is ordered is not enough!
- We need a way to generate ordered lists!

Generators

Outline

- We test mostly trivial or very simple cases (only one insert in the middle of the list!)
- Just checking whether list is ordered is not enough!
- We need a way to generate ordered lists!

$Gen\ a$

Definition

Generators are instances of the Monad class with the (simplified) concrete representation:

$$\textbf{newtype}\ Gen\ a = Gen\ (Rand \to a)$$

The types of bind and return suggest we can use them as combinators to build complex generators out of simpler ones:

$$return :: a \to Gen\ a$$
$$(\gg=) :: Gen\ a \to (a \to Gen\ b) \to Gen\ b$$

$Arbitrary\ a$

Definition

The type class $Arbitrary\ a$ denotes types for which we can generate random values:

$$
\begin{array}{l}
\textbf{class}\ Arbitrary\ a\ \textbf{where} \\
\quad arbitrary :: Gen\ a
\end{array}
$$

And these values are used in a property by applying $forAll$:

$$
\begin{array}{l}
forAll :: (Show\ a, Testable\ b) \Rightarrow \\
\quad Gen\ a \rightarrow (a \rightarrow b) \rightarrow Property
\end{array}
$$

Arbitrary instances

Definition

Given a function $choose :: (Int, Int) \rightarrow Gen\ Int$, we write:

$$\begin{array}{l}\textbf{instance}\ Arbitrary\ Int\ \textbf{where}\\ \quad arbitrary = choose\ (-42, 42)\end{array}$$

We can use the built-in *liftM2* monad function to add pairs to the *Arbitrary* class.

$$\begin{array}{l}\textbf{instance}\ (Arbitrary\ a, Arbitrary\ b) \Rightarrow\\ \quad Arbitrary\ (a, b)\ \textbf{where}\\ \qquad arbitrary = liftM2\ (,)\ arbitrary\ arbitrary\end{array}$$

Enumeration generator

Definition

The $\mathit{oneof} :: [\mathit{Gen}\ a] \rightarrow \mathit{Gen}\ a$ combinator randomly selects one generator from a list. Elements are weighted equally.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Prof} = \mathit{Steve} \mid \mathit{Stephanie} \mid \mathit{Benjamin} \\
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{Prof}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{oneof} \\
\quad [\mathit{return}\ \mathit{Steve}, \mathit{return}\ \mathit{Stephanie}, \mathit{return}\ \mathit{Benjamin}]
\end{array}
$$

We can also define $\mathit{Arbitrary}\ [a]$ using oneof:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arbitrary}\ a \Rightarrow \mathit{Arbitrary}\ [a]\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{oneof} \\
\qquad [\mathit{return}\ [\,], \mathit{liftM2}\ (:)\ \mathit{arbitrary}\ \mathit{arbitrary}]
\end{array}
$$

List generator

Definition

Our previous instantiation of $Arbitrary\ [a]$ created empty lists half the time. To fix this we use $frequency :: [(Int, Gen\ a)] \rightarrow Gen\ a$:

$$
\begin{array}{l}
\textbf{instance}\ Arbitrary\ a \Rightarrow Arbitrary\ [a]\ \textbf{where} \\
\quad arbitrary = frequency \\
\qquad [(1, return\ [\,]), \\
\qquad\quad (4, liftM2\ (:)\ arbitrary\ arbitrary)]
\end{array}
$$

Trees generator

Definition

We can also instantiate a tree generator:

$$\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
\textbf{instance}\ \mathit{Arbitrary}\ a \Rightarrow \\
\quad \mathit{Arbitrary}\ \mathit{Tree}\ a\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{frequency} \\
\qquad [(1, \mathit{LiftM}\ \mathit{Leaf}\ \mathit{arbitrary}), \\
\qquad\quad (2, \mathit{LiftM2}\ \mathit{Branch}\ \mathit{arbitrary}\ \mathit{arbitrary})]
\end{array}$$

What's wrong with this definition?

Trees generator

Definition

We can also instantiate a tree generator:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
\textbf{instance}\ \mathit{Arbitrary}\ a \Rightarrow \\
\quad \mathit{Arbitrary}\ \mathit{Tree}\ a\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{frequency} \\
\qquad [(1, \mathit{LiftM}\ \mathit{Leaf}\ \mathit{arbitrary}), \\
\qquad\quad (2, \mathit{LiftM2}\ \mathit{Branch}\ \mathit{arbitrary}\ \mathit{arbitrary})]
\end{array}
$$

What's wrong with this definition?

Sized generators

Definition

We can ensure generated data structures have finite size by adding an explicit size parameter to $Gen\ a$. Our deïňĄnition becomes

$$\mathbf{newtype}\ Gen\ a = Gen\ (Int \to Rand \to a)$$

and is used with a new combinator:

$$sized :: (Int \to Gen\ a) \to Gen\ a$$

Tree generator

Definition

The following tree deïňĄnition will produce a trees with no more elements than the parameter to *arbTree*. Note that this parameter is passed in by *sized* and is a global constant.

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Arbitrary a =>
  Arbitrary Tree a where
  arbitrary = sized arbTree

arbTree 0 = liftM Leaf arbitrary
arbTree n = frequency
  [(1, liftM Leaf arbitrary),
    (2, liftM2 Branch
      (arbTree (n `div` 2))
      (arbTree (n `div` 2)))]
```

insert examples

Definition

Back to our problem:

$$
\begin{aligned}
&\mathit{insBad}\ a\ [\,] = [\,a\,]\\
&\mathit{insBad}\ a\ y\\
&\quad \mid (\mathit{length}\ y) > 4 = y +\!\!+ [\,a\,]\\
&\quad \mid \mathit{otherwise} = \mathit{insert}\ a\ y\\
&\mathit{propInsertBad1} :: \mathit{Int} \to [\,\mathit{Int}\,] \to \mathit{Property}\\
&\mathit{propInsertBad1}\ x\ xs =\\
&\quad \mathit{ordered}\ xs ==> \mathit{ordered}\ (\mathit{insBad}\ x\ xs)
\end{aligned}
$$

Output

```
Test> quickCheck propInsertBad1
OK, passed 100 tests.
```

orderedList

Definition

Now we can define *orderedList* generator:

$$
\begin{aligned}
&\mathit{orderedList} = \mathbf{do}\\
&\quad a \leftarrow \mathit{frequency}\ [(1, \mathit{return}\ [\,]),\\
&\qquad (7, \mathit{liftM2}\ (:)\ \mathit{arbitrary}\ \mathit{arbitrary})]\\
&\quad \mathit{return}\ (\mathit{sort}\ a)
\end{aligned}
$$

Example

Definition

And finally fail the example!

$$propInsertBad4 :: Int \rightarrow Property$$
$$propInsertBad4\ x =$$
$$\quad forAll\ orderedList\ \$\ \lambda xs \rightarrow ordered\ (insBad\ x\ xs)$$

Output

```
*Test> quickCheck propInsertBad4
Falsifiable, after 10 tests:
-6
[-8,-4,-3,0,5]
```

Infinite Structures

Definition

Infinite structures will cause infinite loops:

$$
\begin{aligned}
&\mathit{propDoubleCycle1} :: [\mathit{Int}] \rightarrow \mathit{Property}\\
&\mathit{propDoubleCycle1}\ \mathit{xs} =\\
&\quad \neg\,(\mathit{null}\ \mathit{xs}) ==>\\
&\qquad \mathit{cycle}\ \mathit{xs} \equiv \mathit{cycle}\ (\mathit{xs} +\!\!+\ \mathit{xs})
\end{aligned}
$$

Infinite Structures

Definition

However we can control them up to any finite size:

$$
\begin{aligned}
&\mathit{propDoubleCycle2} :: [\mathit{Int}] \rightarrow \mathit{Int} \rightarrow \mathit{Property}\\
&\mathit{propDoubleCycle2}\ \mathit{xs}\ n =\\
&\quad \neg\,(\mathit{null}\ \mathit{xs}) \wedge n \geqslant 0 \mathrel{==>}\\
&\qquad \mathit{take}\ n\ (\mathit{cycle}\ \mathit{xs}) \equiv \mathit{take}\ n\ (\mathit{cycle}\ (\mathit{xs} +\!\!+ \mathit{xs}))
\end{aligned}
$$

Functions

Definition

Let's try to define random functions by throwing away the input and generating a random result. In this case:

$$
\begin{aligned}
&propFunc1 :: (Int \to Int) \to Int \to Bool \\
&propFunc1\ f\ x = (f \circ (+2))\ x \equiv (f \circ (*2))\ x
\end{aligned}
$$

Output

```
Test> quickCheck propFunc1 OK, passed 100 tests.
```

Functions

Outline

- We need a functional dependency between input and output, or we can get wrong results
- Type of $Gen\ (a \to b)$ is $Int \to Rand \to a \to b$
  - This is equivalent to $a \to Int \to Rand \to b$
  - And $a \to Gen\ b$
- It's not clear we can make a value of one type into a generator for another.
  - However maybe we can use arbitrary Ints to transform generators with $variant :: Int \to Gen\ a \to Gen\ a$.
  - We can certainly make specific types into Ints:

$$\begin{aligned} &coarbitrary\ b = \text{if}\ b \\ &\quad \text{then}\ variant\ 1 \\ &\quad \text{else}\ variant\ 0 \end{aligned}$$

Functions

Outline

- We need a functional dependency between input and output, or we can get wrong results
- Type of $Gen\ (a \to b)$ is $Int \to Rand \to a \to b$
  - This is equivalent to $a \to Int \to Rand \to b$
  - And $a \to Gen\ b$
- It's not clear we can make a value of one type into a generator for another.
  - However maybe we can use arbitrary Ints to transform generators with $variant :: Int \to Gen\ a \to Gen\ a$.
  - We can certainly make specific types into Ints:

$$\begin{aligned} &coarbitrary\ b = \text{if } b \\ &\quad \text{then } variant\ 1 \\ &\quad \text{else } variant\ 0 \end{aligned}$$

Functions

Outline

- We need a functional dependency between input and output, or we can get wrong results
- Type of $Gen\ (a \to b)$ is $Int \to Rand \to a \to b$
  - This is equivalent to $a \to Int \to Rand \to b$
  - And $a \to Gen\ b$
- It's not clear we can make a value of one type into a generator for another.
  - However maybe we can use arbitrary Ints to transform generators with $variant :: Int \to Gen\ a \to Gen\ a$.
  - We can certainly make specific types into Ints:

$$
\begin{aligned}
&coarbitrary\ b = \text{if } b \\
&\quad \text{then } variant\ 1 \\
&\quad \text{else } variant\ 0
\end{aligned}
$$

Functions

Outline

- We need a functional dependency between input and output, or we can get wrong results
- Type of $Gen\ (a \to b)$ is $Int \to Rand \to a \to b$
  - This is equivalent to $a \to Int \to Rand \to b$
  - And $a \to Gen\ b$
- It's not clear we can make a value of one type into a generator for another.
  - However maybe we can use arbitrary Ints to transform generators with $variant :: Int \to Gen\ a \to Gen\ a$.
  - We can certainly make specific types into Ints:

$$\begin{aligned}&coarbitrary\ b = \text{if } b\\ &\quad \text{then } variant\ 1\\ &\quad \text{else } variant\ 0\end{aligned}$$

Functions

Outline

- We need a functional dependency between input and output, or we can get wrong results
- Type of $Gen\ (a \rightarrow b)$ is $Int \rightarrow Rand \rightarrow a \rightarrow b$
  - This is equivalent to $a \rightarrow Int \rightarrow Rand \rightarrow b$
  - And $a \rightarrow Gen\ b$
- It's not clear we can make a value of one type into a generator for another.
  - However maybe we can use arbitrary Ints to transform generators with $variant :: Int \rightarrow Gen\ a \rightarrow Gen\ a$.
  - We can certainly make specific types into Ints:

$$
\begin{aligned}
&coarbitrary\ b = \textbf{if}\ b \\
&\quad \textbf{then}\ variant\ 1 \\
&\quad \textbf{else}\ variant\ 0
\end{aligned}
$$

Functions

Outline

- We need a functional dependency between input and output, or we can get wrong results
- Type of $Gen\ (a \to b)$ is $Int \to Rand \to a \to b$
  - This is equivalent to $a \to Int \to Rand \to b$
  - And $a \to Gen\ b$
- It's not clear we can make a value of one type into a generator for another.
  - However maybe we can use arbitrary Ints to transform generators with $variant :: Int \to Gen\ a \to Gen\ a$.
  - We can certainly make specific types into Ints:

$$\begin{aligned}&coarbitrary\ b = \text{if } b\\&\quad \text{then } variant\ 1\\&\quad \text{else } variant\ 0\end{aligned}$$

Functions

Outline

- We need a functional dependency between input and output, or we can get wrong results
- Type of $Gen\ (a \rightarrow b)$ is $Int \rightarrow Rand \rightarrow a \rightarrow b$
  - This is equivalent to $a \rightarrow Int \rightarrow Rand \rightarrow b$
  - And $a \rightarrow Gen\ b$
- It's not clear we can make a value of one type into a generator for another.
  - However maybe we can use arbitrary Ints to transform generators with $variant :: Int \rightarrow Gen\ a \rightarrow Gen\ a$.
  - We can certainly make specific types into Ints:

$$\begin{array}{l} coarbitrary\ b = \textbf{if}\ b \\ \quad \textbf{then}\ variant\ 1 \\ \quad \textbf{else}\ variant\ 0 \end{array}$$

Functions

Outline

- In Haskell, the right way to generalize this is with a type class.

$$
\begin{array}{l}
\textbf{class } \mathit{Coarbitrary}\ a\ \textbf{where} \\
\quad \mathit{coarbitrary} :: a \to \mathit{Gen}\ b \to \mathit{Gen}\ b
\end{array}
$$

- We then define *Arbitrary* in terms of *Coarbitrary* (and a helper function to match the types).

$$
\begin{array}{l}
\textbf{instance } (\mathit{Coarbitrary}\ a, \mathit{Arbitrary}\ b) \Rightarrow \\
\quad \mathit{Arbitrary}\ (a \to b)\ \textbf{where} \\
\quad\quad \mathit{arbitrary} = \\
\quad\quad\quad \mathit{promote}\ (\lambda a \to \mathit{coarbitrary}\ a\ \mathit{arbitrary})
\end{array}
$$

Functions

Outline

- In Haskell, the right way to generalize this is with a type class.

$$\textbf{class}\ \mathit{Coarbitrary}\ a\ \textbf{where}$$
$$\quad \mathit{coarbitrary} :: a \to \mathit{Gen}\ b \to \mathit{Gen}\ b$$

- We then define *Arbitrary* in terms of *Coarbitrary* (and a helper function to match the types).

$$\textbf{instance}\ (\mathit{Coarbitrary}\ a, \mathit{Arbitrary}\ b) \Rightarrow$$
$$\quad \mathit{Arbitrary}\ (a \to b)\ \textbf{where}$$
$$\qquad \mathit{arbitrary} =$$
$$\qquad\quad \mathit{promote}\ (\lambda a \to \mathit{coarbitrary}\ a\ \mathit{arbitrary})$$

Functions

Definition

$$
\begin{aligned}
&\mathit{variant} :: \mathit{Int} \to \mathit{Gen}\ a \to \mathit{Gen}\ a\\
&\mathit{variant}\ v\ (\mathit{Gen}\ m) =\\
&\quad \mathit{Gen}\ (\lambda n\ r \to m\ n\ (\mathit{rands}\ r\ !!\ (v+1)))\\
&\textbf{where}\\
&\quad \mathit{rands}\ r0 = r1 : \mathit{rands}\ r2\ \textbf{where}\ (r1, r2) = \mathit{split}\ r0\\
&\mathit{promote} :: (a \to \mathit{Gen}\ b) \to \mathit{Gen}\ (a \to b)\\
&\mathit{promote}\ f =\\
&\quad \mathit{Gen}\ (\lambda n\ r \to \lambda a \to \textbf{let}\ \mathit{Gen}\ m = f\ a\ \textbf{in}\ m\ n\ r)
\end{aligned}
$$

Functions

Definition

instance *Coarbitrary Bool* **where**
  coarbitrary b =
    if *b* **then** *variant* 0 **else** *variant* 1
instance *Coarbitrary Int* **where**
  coarbitrary n =
    variant (**if** $n \geqslant 0$ **then** $2 * n$ **else** $2 * (-n) + 1$)
instance *Coarbitrary Char* **where**
  coarbitrary c = *variant* (*ord c*)

Functions

Definition

And back to the example:

$$propFunc1 :: (Int \to Int) \to Int \to Bool$$
$$propFunc1\ f\ x = (f \circ (+2))\ x \equiv (f \circ (*2))\ x$$

Output

```
*Test> quickCheck propFunc1
Falsifiable, after 0 tests:
*function*
-3
```

Implementation

Definition

$$\textbf{newtype } \mathit{Property} = \mathit{Prop}\ (\mathit{Gen\ Result})$$

$$\textbf{class } \mathit{Testable}\ a \textbf{ where}$$
$$\quad \mathit{property} :: a \to \mathit{Property}$$

$$\textbf{instance } \mathit{Testable\ Bool} \textbf{ where}$$
$$\quad \mathit{property}\ b = \mathit{Prop}\ (\mathit{return}\ \$\ \mathit{resultBool}\ b)$$

$$\textbf{instance } \mathit{Testable\ Property} \textbf{ where}$$
$$\quad \mathit{property\ prop} = \mathit{prop}$$

$$\textbf{instance } (\mathit{Arbitrary}\ a, \mathit{Show}\ a, \mathit{Testable}\ b) \Rightarrow$$
$$\quad \mathit{Testable}\ (a \to b) \textbf{ where}$$
$$\qquad \mathit{property}\ f = \mathit{forAll\ arbitrary}\ f$$

Testing Monads

Outline

- It is impossible to random test IO monad
- ST monad can be tested by randomly generating lists of actions
- It is not too comfortable
- However since functions like ==> are defined on $Property$s, we need to redefine them on a monad transformer $PropertyM$
- QuickCheck2 provides support for that

Testing Monads

Outline

- It is impossible to random test IO monad
- ST monad can be tested by randomly generating lists of actions
- It is not too comfortable
- However since functions like ==> are defined on $Property$s, we need to redefine them on a monad transformer $PropertyM$
- QuickCheck2 provides support for that

Testing Monads

Outline

- It is impossible to random test IO monad
- ST monad can be tested by randomly generating lists of actions
- It is not too comfortable
- However since functions like $==>$ are defined on $Propertys$, we need to redefine them on a monad transformer $PropertyM$
- QuickCheck2 provides support for that

Testing Monads

Outline

- It is impossible to random test *IO* monad
- *ST* monad can be tested by randomly generating lists of actions
- It is not too comfortable
- However since functions like ==> are defined on *Property*s, we need to redefine them on a monad transformer *PropertyM*
- QuickCheck2 provides support for that

Testing Monads

Outline

- It is impossible to random test IO monad
- ST monad can be tested by randomly generating lists of actions
- It is not too comfortable
- However since functions like $==>$ are defined on $Propertys$, we need to redefine them on a monad transformer $PropertyM$
- QuickCheck2 provides support for that

Shrinking

Outline

- Often we find a counter example, but it's way too big to understand the underlying cause
- In such a case it is possible to start shrinking the example to find a subexample that still causes the function to fail
- This is implemented as an extra function *shrink* in *Arbitrary* class that generates all substructures
- QuickCheck2 implements these and some extra for most common structures

Shrinking

Outline

- Often we find a counter example, but it's way too big to understand the underlying cause
- In such a case it is possible to start shrinking the example to find a subexample that still causes the function to fail
- This is implemented as an extra function *shrink* in *Arbitrary* class that generates all substructures
- QuickCheck2 implements these and some extra for most common structures

Shrinking

Outline

- Often we find a counter example, but it's way too big to understand the underlying cause
- In such a case it is possible to start shrinking the example to find a subexample that still causes the function to fail
- This is implemented as an extra function *shrink* in *Arbitrary* class that generates all substructures
- QuickCheck2 implements these and some extra for most common structures

Shrinking

Outline

- Often we find a counter example, but it's way too big to understand the underlying cause
- In such a case it is possible to start shrinking the example to find a subexample that still causes the function to fail
- This is implemented as an extra function *shrink* in *Arbitrary* class that generates all substructures
- QuickCheck2 implements these and some extra for most common structures