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:

\[
\text{propRevApp2} :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool} \\
\text{propRevApp2} \hspace{1em} xs \hspace{1em} ys = \\
\text{reverse} \hspace{1em} (xs \hspace{1em} ++ \hspace{1em} ys) \equiv \text{reverse} \hspace{1em} ys \hspace{1em} ++ \hspace{1em} \text{reverse} \hspace{1em} xs
\]

Output
Test> quickCheck propRevApp2
OK, passed 100 tests.
reverse examples

Property

We concatenated in a wrong order:

\[ propRevApp2 :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool} \]

\[
propRevApp2 \; xs \; ys = \\
reverse \; (xs \; \mathbin{+\!\!+} \; ys) \equiv reverse \; ys \; \mathbin{+\!\!+} \; reverse \; xs
\]

Output

Test> quickCheck propRevApp2
OK, passed 100 tests.
reverse examples

Property

Let’s check if you can reverse before concatenating:

\[ propRevApp1 :: [Int] \to [Int] \to Bool \]
\[ propRevApp1 \; xs \; ys = \]
\[ \text{reverse} \; (xs \; \mathbin{\|} \; ys) \equiv \text{reverse} \; xs \; \mathbin{\|} \; \text{reverse} \; ys \]

Output

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

Let’s check if you can reverse before concatenating:

\[ propRevApp1 :: [\text{Int}] \to [\text{Int}] \to \text{Bool} \]
\[ propRevApp1 \; xs \; ys = \]
\[ \text{reverse} \; (xs \; \mathbin{+\!+} \; ys) \equiv \text{reverse} \; xs \; \mathbin{+\!+} \; \text{reverse} \; ys \]

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:

\[ propDistributiveI \colon \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \]

\[ propDistributiveI \ a \ b \ c = \]
\[ a \times (b + c) \equiv (a \times b) + (a \times c) \]

Output

Test> propDistributiveI
OK, passed 100 tests.
Distribution examples

Property

The following property asserts that addition and multiplication distribute:

\[ \text{propDistributiveI} :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Bool} \]
\[ \text{propDistributiveI } a \ b \ c = \]
\[ a \ast (b + c) \equiv (a \ast b) + (a \ast c) \]

Output

Test> propDistributiveI
OK, passed 100 tests.
Distribution examples

**Property**

The same property for *Floats* fails:

\[
\text{propDistributiveF} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Bool} \\
\text{propDistributiveF} \ a \ b \ c = \\
a \times (b + c) \equiv (a \times b) + (a \times c)
\]

**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 → Float → Float → Bool
propDistributiveF a b c =
    a * (b + c) ≡ (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.

\[
\text{insert } e \ (x : xs) = \\
\quad \text{if } e < x \text{ then } e : x : xs \text{ else } x : (\text{insert } e \ xs)
\]

\[
\text{insert } e \ [] = [e]
\]

\textit{ordered} tests whether the list is ordered:

\[
\text{ordered} :: \text{Ord } a \Rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{ordered } [] = \text{True}
\]

\[
\text{ordered } (x : []) = \text{True}
\]

\[
\text{ordered } (x1 : x2 : xs) = \\
\quad \text{if } x1 \leq x2 \text{ then } \text{ordered } (x2 : xs) \text{ else } \text{False}
\]
Property

We would want to test whether \textit{insert} works, but this has point only on ordered lists:

\[
\text{propInsert1} :: \text{Int} \to [\text{Int}] \to \text{Bool} \\
\text{propInsert1} \; x \; xs = \\
\quad \text{if ordered} \; xs \\
\quad \quad \text{then ordered} \; (\text{insert} \; x \; xs) \\
\quad \text{else True}
\]

Since QuickCheck does not work on polymorphic types we choose \textit{Int}s here.

Output

Test> propInsert1
OK, passed 100 tests.
We would want to test whether `insert` works, but this has point only on ordered lists:

```haskell
propInsert1 :: Int → [Int] → Bool
propInsert1 x xs =
  if ordered xs
  then ordered (insert x xs)
  else True
```

Since QuickCheck does not work on polymorphic types we choose `Ints` here.

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?

\[ propInsert2 :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Property} \]
\[ propInsert2 \ x \ xs = \]
\[ (\text{length } xs \equiv 0 \lor \neg (\text{ordered } xs)) \text{ 'trivial' if ordered } xs \]
\[ \text{then ordered } (\text{insert } x \ xs) \]
\[ \text{else True} \]

**Output**

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

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

\[
\text{propInsert2} :: \text{Int} \to [\text{Int}] \to \text{Property}
\]

\[
\text{propInsert2 } x \; xs =
\]

\[
(\text{length } xs \equiv 0 \lor \neg (\text{ordered } xs)) \text{‘trivial‘}
\]

\[
\begin{cases}
\text{if ordered } xs \\
\text{then ordered } (\text{insert } x \; xs) \\
\text{else True}
\end{cases}
\]

**Output**

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

Property

\[ \implies \text{ is the QuickCheck combinator that makes it test only the fitting values:} \]

\[
prop\text{Insert3} :: \text{Int} \to [\text{Int}] \to \text{Property}
\]

\[
prop\text{Insert3} \ x \ xs =
\]

\[
\text{ordered} \ xs \implies \text{ordered} \ (\text{insert} \ x \ xs)
\]

Output

Test\> prop\text{Insert3}

OK, passed 100 tests.
Property

\[ \Rightarrow \] is the QuickCheck combinator that makes it test only the fitting values:

\[
\text{propInsert3 :: Int} \rightarrow [\text{Int}] \rightarrow \text{Property}
\]
\[
\text{propInsert3 } x \; \text{xs} =
\]
\[
\text{ordered xs} \Rightarrow \text{ordered (insert } x \; \text{xs)}
\]

Output

Test\> propInsert3

OK, passed 100 tests.
Property

How well do we actually test? Can this pass?

\[
\begin{align*}
\text{insBad } a \; [\;] &= [\;a]\; \\
\text{insBad } a \; y &= \begin{cases} 
(length \; y) > 4 = y ++ [\;a] \\
\text{otherwise} = \text{insert } a \; y 
\end{cases}
\end{align*}
\]

\[
\text{propInsertBad1 :: Int} \rightarrow [\text{Int}] \rightarrow \text{Property}
\]

\[
\text{propInsertBad1 } x \; xs = \begin{cases} 
\text{ordered } xs ==\Rightarrow \text{ordered } (\text{insBad } x \; xs) 
\end{cases}
\]

Output

Test> quickCheck propInsertBad1
OK, passed 100 tests.
Property

How well do we actually test? Can this pass?

\[
\text{insBad } a \ [] = [a] \\
\text{insBad } a \ y \\
\quad | (\text{length } y) > 4 = y \ ++ \ [a] \\
\quad | \text{otherwise } = \text{insert } a \ y
\]

\text{propInsertBad1 } :: \text{Int } \rightarrow [\text{Int}] \rightarrow \text{Property}

\[
\text{propInsertBad1 } x \ xxs = \\
\quad \text{ordered } xxs =\Rightarrow \text{ordered } (\text{insBad } x \ xxs)
\]

Output

Test> quickCheck propInsertBad1
OK, passed 100 tests.
insert examples

Property

\[ prop\text{InsertBad2} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Property} \]
\[ prop\text{InsertBad2} x \text{ xs} = \]
\[ \text{ordered xs =>}
\] \[ \text{collect (length xs) } \$ \text{ ordered (insBad x xs)} \]

Output

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

**Property**

\[ propInsertBad2 :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Property} \]

\[ propInsertBad2 \ x \ xs = \]

\[ \text{ordered} \ xs \implies \]

\[ \text{collect} \ (\text{length} \ xs) \ \& \ \text{ordered} \ (\text{insBad} \ x \ xs) \]

**Output**

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

**Property**

\[ propInsertBad3 :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{Property} \]
\[ propInsertBad3 \ x \ xs = \]
\[ \text{ordered} \ \text{xs} \implies \]
\[ \text{classify} \ (\text{ordered} \ (x : xs)) \ "at-head" \$
\[ \text{classify} \ (\text{ordered} \ (xs \uplus [x])) \ "at-tail" \$
\[ \text{ordered} \ (\text{insBad} \ x \ xs) \]

**Output**

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

Property

\[
\text{propInsertBad3} :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{Property} \\
\text{propInsertBad3} \ x \ xs = \\
\text{ordered} \ xs \implies \\
\phantom{\text{propInsertBad3}} \ (\text{classify} \ (\text{ordered} \ (x : xs))) \ "\text{at-head}" \$
\phantom{\text{propInsertBad3}} \ (\text{classify} \ (\text{ordered} \ (xs ++ [x]))) \ "\text{at-tail}" \$
\phantom{\text{propInsertBad3}} \ (\text{ordered} \ (\text{insBad} \ x \ xs))
\]

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

\[
\text{newtype } \text{Gen } a = \text{Gen} \ (\text{Rand} \to a)
\]

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

\[
\text{return} :: a \to \text{Gen } a
\]
\[
(\gg=) :: \text{Gen } a \to (a \to \text{Gen } b) \to \text{Gen } b
\]
Arbitrary a

Definition

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

\[
\text{class Arbitrary } a \text{ where} \\
\quad \text{arbitrary :: Gen } a
\]

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

\[
forall :: (Show a, Testable b) \Rightarrow \\
\quad \text{Gen } a \to (a \to b) \to Property
\]
**Arbitrary instances**

**Definition**

Given a function \( \text{choose} :: (\text{Int}, \text{Int}) \rightarrow \text{Gen Int} \), we write:

\[
\text{instance Arbitrary Int where}
\]
\[
\text{arbitrary} = \text{choose} (-42, 42)
\]

We can use the built-in \( \text{liftM2} \) monad function to add pairs to the \( \text{Arbitrary} \) class.

\[
\text{instance (Arbitrary a, Arbitrary b) \Rightarrow }
\]
\[
\text{Arbitrary (a, b) where}
\]
\[
\text{arbitrary} = \text{liftM2} (,) \text{arbitrary arbitrary}
\]
Enumeration generator

Definition

The `oneof :: [Gen a] → Gen a` combinator randomly selects one generator from a list. Elements are weighted equally.

```haskell
data Prof = Steve | Stephanie | Benjamin
instance Arbitrary Prof where
    arbitrary = oneof
    [return Steve, return Stephanie, return Benjamin]
```

We can also define `Arbitrary [a]` using `oneof`:

```haskell
instance Arbitrary a ⇒ Arbitrary [a] where
    arbitrary = oneof
    [return [], liftM2 (:) arbitrary arbitrary]
```
Our previous instantiation of `Arbitrary [a]` created empty lists half the time. To fix this we use

\[\text{frequency} :: [(\text{Int}, \text{Gen } a)] \rightarrow \text{Gen } a:\]

\[
\text{instance } \text{Arbitrary } a \Rightarrow \text{Arbitrary } [a] \text{ where}
\]
\[
\text{arbitrary} = \text{frequency}
\]
\[
[(1, \text{return } []),
(4, \text{liftM2 } (: \text{ arbitrary arbitrary})] \]


Trees generator

Definition

We can also instantiate a tree generator:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Arbitrary a ⇒
    Arbitrary Tree a where
    arbitrary = frequency
        [(1, LiftM Leaf arbitrary),
        (2, LiftM2 Branch arbitrary arbitrary)]
```

What's wrong with this definition?
Trees generator

Definition

We can also instantiate a tree generator:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Arbitrary a ⇒
    Arbitrary Tree a where
    arbitrary = frequency
    [(1, LiftM Leaf arbitrary),
    (2, LiftM2 Branch arbitrary arbitrary)]
```

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 definition becomes

$$\text{newtype } Gen\ a = Gen\ (\text{Int} \rightarrow Rand \rightarrow a)$$

and is used with a new combinator:

$$\text{sized} :: (\text{Int} \rightarrow Gen\ a) \rightarrow Gen\ a$$
Tree generator

Definition

The following tree definition will produce a tree with no more elements than the parameter to \textit{arbTree}. Note that this parameter is passed in by \textit{sized} and is a global constant.

\begin{verbatim}
data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Arbitrary a \Rightarrow
    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)))]
\end{verbatim}
**Definition**

Back to our problem:

\[
\text{insBad } a \ [\] = [\ a] \\
\text{insBad } a \ y \\
\quad | \ (\text{length } y) > 4 = y \ ++ \ [\ a] \\
\quad | \ \text{otherwise} = \text{insert } a \ y
\]

\[prop\text{InsertBad1} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Property}\]

\[prop\text{InsertBad1} \ x \ xs = \]

\[\text{ordered } xs \implies \text{ordered } (\text{insBad } x \ xs)\]

**Output**

Test\> \text{quickCheck} \ prop\text{InsertBad1}

OK, passed 100 tests.
Now we can define `orderedList` generator:

```haskell
orderedList = do
    a ← frequency [(1, return []), (7, liftM2 (:) arbitrary arbitrary)]
    return (sort a)
```
Definition

And finally fail the example!

\[
\text{propInsertBad4} :: \text{Int} \to \text{Property} \\
\text{propInsertBad4} \ x = \\
\text{forAll orderedList } \lambda \text{xs} \Rightarrow \text{ordered (insBad x xs)}
\]

Output

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

**Definition**

Infinite structures will cause infinite loops:

\[
\text{propDoubleCycle1} :: [\text{Int}] \rightarrow \text{Property} \\
\text{propDoubleCycle1} \; \text{xs} = \\
\neg (\text{null} \; \text{xs}) \Rightarrow \\
\text{cycle} \; \text{xs} \equiv \text{cycle} \; (\text{xs} \; \| \; \text{xs})
\]
Infinite Structures

Definition

However we can control them up to any finite size:

\[\text{propDoubleCycle2} :: [\text{Int}] \to \text{Int} \to \text{Property}\]

\[\text{propDoubleCycle2} \; \text{xs} \; n =\]

\[\neg (\text{null} \; \text{xs}) \land n \geq 0 \implies\]

\[\text{take} \; n \; (\text{cycle} \; \text{xs}) \equiv \text{take} \; n \; (\text{cycle} \; (\text{xs} \; \cdot \; \cdot \text{xs}))\]
Functions

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

\[ propFunc1 :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Bool} \]
\[ propFunc1 \ f \ x = (f \circ (+2)) \ x \equiv (f \circ (*2)) \ x \]

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 $\text{Gen} (a \rightarrow b)$ is $\text{Int} \rightarrow \text{Rand} \rightarrow a \rightarrow b$
  - This is equivalent to $a \rightarrow \text{Int} \rightarrow \text{Rand} \rightarrow b$
  - And $a \rightarrow \text{Gen} b$
- 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 $\text{variant :: Int} \rightarrow \text{Gen} a \rightarrow \text{Gen} a$.
  - We can certainly make specific types into Ints:

\[ \text{coarbitrary } b = \begin{cases} \text{variant } 1 & \text{if } b \\ \text{variant } 0 & \text{else} \end{cases} \]
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 $\text{variant} :: Int \rightarrow Gen \ a \rightarrow Gen \ a$.
- We can certainly make specific types into Ints:

```haskell
coarbitrary \ b = if \ b
  then \ variant \ 1
  else \ variant \ 0
```
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:

  ```haskell
  coarbitrary b = if b
                  then variant 1
                  else variant 0
  ```
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:

```plaintext
coarbitrary b = if b
    then variant 1
    else variant 0
```
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:

\[
\text{coarbitrary } b = \begin{cases} 
\text{variant 1} & \text{if } b \\
\text{variant 0} & \text{else}
\end{cases}
\]
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:

\[
\text{coarbitrary } b = \text{if } b \\
\text{then } variant 1 \\
\text{else } variant 0
\]
We need a functional dependency between input and output, or we can get wrong results.

Type of $\text{Gen} \ (a \rightarrow b)$ is $\text{Int} \rightarrow \text{Rand} \rightarrow a \rightarrow b$

- This is equivalent to $a \rightarrow \text{Int} \rightarrow \text{Rand} \rightarrow b$
- And $a \rightarrow \text{Gen} \ b$

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 $\text{variant} :: \text{Int} \rightarrow \text{Gen} \ a \rightarrow \text{Gen} \ a$.
- We can certainly make specific types into Ints:

```latex
coarbitrary \ b = \text{if} \ b \\
\text{then} \ \text{variant} \ 1 \\
\text{else} \ \text{variant} \ 0
```
In Haskell, the right way to generalize this is with a type class.

```haskell
class Coarbitrary a where
  coarbitrary :: a → Gen b → Gen b
```

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

```haskell
instance (Coarbitrary a, Arbitrary b) ⇒
  Arbitrary (a → b) where
  arbitrary =
    promote (λa → coarbitrary a arbitrary)
```
In Haskell, the right way to generalize this is with a type class.

```haskell
class Coarbitrary a where
  coarbitrary :: a -> Gen b -> Gen b
```

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

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

Definition

\[\text{variant} :: \text{Int} \to \text{Gen } a \to \text{Gen } a\]
\[\text{variant } v \ (\text{Gen } m) = \]
\[\text{Gen } (\lambda n \ r \to m \ n \ (\text{rands } r \cdot!! \ (v + 1)))\]
\[\text{where}\]
\[\text{rands } r0 = r1 : \text{rands } r2 \text{ where } (r1, r2) = \text{split } r0\]
\[\text{promote} :: (a \to \text{Gen } b) \to \text{Gen } (a \to b)\]
\[\text{promote } f = \]
\[\text{Gen } (\lambda n \ r \to \lambda a \to \text{let } \text{Gen } m = f \ a \text{ in } m \ n \ r)\]
instance Coarbitrary Bool where
    coarbitrary b =
        if b then variant 0 else variant 1
instance Coarbitrary Int where
    coarbitrary n =
        variant (if n \geq 0 then 2 * n else 2 * (\neg n) + 1)
instance Coarbitrary Char where
    coarbitrary c = variant (ord c)
Functions

Definition

And back to the example:

\[ propFunc1 :: (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Bool} \]
\[ propFunc1 f x = (f \circ (+2)) \equiv (f \circ (*2)) \]

Output

*Test> quickCheck propFunc1
Falsifiable, after 0 tests:
*function*
-3
newtype Property = Prop (Gen Result)
class Testable a where
    property :: a -> Property
instance Testable Bool where
    property b = Prop (return $ resultBool b)
instance Testable Property where
    property prop = prop
instance (Arbitrary a, Show a, Testable b) =>
    Testable (a -> b) where
    property f = 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 $\Rightarrow$ are defined on Properties, 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 \( \Rightarrow \) are defined on \( Properties \), we need to redefine them on a monad transformer \( PropertyM \)
- QuickCheck2 provides support for that
Testing Monads

Outline

- It is impossible to random test \textit{IO} monad
- \textit{ST} monad can be tested by randomly generating lists of actions
- It is not too comfortable
  
  However since functions like \texttt{==>} are defined on \textit{Properties}, we need to redefine them on a monad transformer \textit{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 \( Properties \), we need to redefine them on a monad transformer \( PropertyM \)
- QuickCheck2 provides support for that
Testing Monads

Outline

- It is impossible to random test \textit{IO} monad
- \textit{ST} monad can be tested by randomly generating lists of actions
- It is not too comfortable
- However since functions like \texttt{===>} are defined on \textit{Properties}, we need to redefine them on a monad transformer \textit{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 the `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 the `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