QuickCheck: Custom Generators

QuickCheck allows us to write our own test-value generators
In particular we can define generators for user-defined algebraic datatypes.
QuickCheck also allows us to run tests with a different generator than the default (arbitrary in class Arbitrary)

QuickCheck Study: Binary Tree

We shall define a binary-tree datatype as an efficient representation of a set.
We shall define an element insertion function
We will define a lookup function.
We will then test a number of key properties.

Starting Out Program

Our program header:

```haskell
module Main where
import Test.QuickCheck
import Control.Monad
```

Why Control.Monad?
QuickCheck’s generators are monadic
We shall want to use some monad features not in the standard prelude.
Our Binary Tree

- We define a tree that can be empty, singleton or a branches of two subtrees around a branch-value.

```haskell
data BTree a = Null
  | Lf a
  | Br (BTree a) a (BTree a)
  deriving (Eq, Ord, Show)
```

- As usual, we intend that elements of the left-subtree be smaller than the branch-value, itself smaller than any element in the right-subtree.

- This property is an invariant of our binary tree.

Binary Tree Insertion

- If tree is empty, make a leaf:

  ```haskell
  btINS x Null = Lf x
  ```

- If tree is a leaf, make a branch with new value at root:

  ```haskell
  btINS x bt@(Lf y)
  | x < y = Br Null x bt
  | x > y = Br bt x Null
  | otherwise = bt
  ```

- Otherwise insert into appropriate sub-tree if not found at root:

  ```haskell
  btINS x bt@(Br left y right)
  | x < y = Br (btINS x left) y right
  | x > y = Br left y (btINS x right)
  | otherwise = bt
  ```

Binary Tree Lookup

- Nothing is in the null tree:

  ```haskell
  btLKP _ Null = False
  ```

- In a leaf if it matches its content:

  ```haskell
  btLKP x (Lf y) = x == y
  ```

- In a branch, if not the root value, look into appropriate sub-tree:

  ```haskell
  btLKP x (Br left y right)
  | x == y = True
  | x < y = btLKP x left
  | x > y = btLKP x right
  ```

Property 1 : Lookup after Insert

- A key property of any set is that a value is present after we have added it in:

  ```latex
  x \in S \cup \{x\}
  ```

- In terms of our operations:

  ```latex
  \text{lookup } x \text{ after } \text{insert } x \text{ is always } \text{true}
  ```

- As a QuickCheck property:

  ```haskell
  prop_lkp_ins bt x
  = btLKP x (btINS x bt)
  where types = (x::Int)
  ```

  (Here we fix our property to work with \texttt{BTree Int}, by using a \texttt{where}-clause that defines an unreferenced name \texttt{types}, but which itself references \texttt{x} with a type signature.)
Property 2: Lookup after different Insert

- The presence of a value is not affected by inserting a different one:
  \[ x \neq y \implies (x \in S \equiv x \in S \cup \{y\}) \]

- In QuickCheck speak:
  ```
  prop_lkp_other bt x y = x /= y
  ==> ( (btLKP x bt) == (btLKP x (btINS y bt)) )
  where types = (x::Int)
  ```

Property 3: Insert order immaterial

- The order in which elements are added does not matter:
  \[ z \in (S \cup \{y\}) \cup \{x\} \equiv z \in (S \cup \{x\}) \cup \{y\} \]

- In QuickCheck:
  ```
  prop_lkp_ins2 bt x y z
  = btLKP z (btINS x (btINS y bt))
  ==
  btLKP z (btINS y (btINS x bt))
  where types = (x::Int)
  ```

- We are indirectly testing the property:
  \[ (S \cup \{y\}) \cup \{x\} = (S \cup \{x\}) \cup \{y\} \]

- We don’t compare BTrees directly (why not?)
- Instead, we explore the observations we can make using the available operations (btINS and btLKP).

Testing BTrees

- We cannot yet test our propositions
- We need a generator (btree) of random binary trees
- How do we do this for a user-defined data-type?
  - Idea: write a generator for each variant: Null, Lf and Br
  - Use the `oneof` combinator to give us a random choice of these
    ```
    oneof :: [Gen a] -> Gen a
    ```

Generating Null

- A generator is complex
  ```
  newtype Gen a = Gen (Int -> StdGen -> a)
  ```
- However it is monadic, so we don’t need to know gory internal details
- We can simply use monadic `return` to produce a Null generator
  ```
  null = return Null
  ```
Generating Lf x

- To generate a leaf element, we need to generate an arbitrary element, and then wrap it with the leaf data constructor:
  ```haskell
do x <- arbitrary -- use generator for element type
    return (Lf x)
```
- Here we do a monadic action (`x <- arbitrary`) and then apply function (data-constructor) `Lf` to the result.
- We can do this as a one-liner using `liftM`
  ```haskell
  liftM :: (Monad m) => (a -> b) -> m a -> m b
  (Here `m` is `Gen` in our case)
  ```
- So the definition of leaf-generator `leaf`:
  ```haskell
  leaf = liftM Lf arbitrary
  (liftM is defined in library Control.Monad)
  ```

Generating Br left x right

- To generate a branch element we need two subtrees and an arbitrary element:
  ```haskell
do left <- btree
    x <- arbitrary
    right <- btree
    return (Br left x right)
```
- We can do this as a one-liner using `liftM3`
  ```haskell
  liftM3 :: (Monad m) => (a -> b -> c -> d) -> m a -> m b -> m c -> m d
  ```
- So the definition of branch-generator `branch`:
  ```haskell
  branch = liftM3 Br btree arbitrary btree
  ```

A BTree generator (attempt 1)

- We put all of this together as follows:
  ```haskell
dodgy_btree :: Gen (BTree Int)
dodgy_btree = oneof [ return Null
                        , liftM Lf arbitrary
                        , liftM3 Br dodgy_btree arbitrary
                        , dodgy_btree ]
```
- How do we test using this generator?
  ```haskell
  forAll :: (Show a, Testable b) =>
           Gen a -> (a -> b) -> Property
  ```

Tests using specific generators

- We can define tests that use a specific named generator and property, one terse, the other verbose:
  ```haskell
gtest gen prop = test $ forAll gen $ prop
vtest gen prop = verboseCheck $ forAll gen $ prop
```
A Dodgy BTree generator

- Why do we call this generator *dodgy_btree*?
  - The reason is that this generator is not guaranteed to terminate
  - It needs to generate null or leaves for every branch to terminate, but it is choosing at random!

Improved BTree generator

- We use the *sized* combinator, that passes a number to its generator argument.
  ```haskell```
  btree :: Gen (BTree Int)
  btree = sized btree'
  ```haskell```
- So now, we define `btree'` which takes an extra `Int` argument
  ```haskell```
  btree' :: Int -> Gen (BTree Int)
  btree' n = ... 
  ```haskell```
- The test functions pass gradually increasing values of this integer parameter to a `sized` generator as testing proceeds
  - This is an example of complex “plumbing” in the testing infrastructure that is hidden inside the `Gen` monad.
  - QuickCheck would be much harder to use without monads.

A Sized BTree generator (I)

- If the size is zero, we generate an empty tree
  ```haskell```
  btree' 0 = return Null
  ```haskell```
- If the size is one, we generate either `Null` or a random leaf.
  ```haskell```
  btree' 1 = oneof [ return Null, liftM Lf arbitrary ]
  ```haskell```

A Sized BTree generator (II)

- If the size is larger than 1, then
  - One time in 8 we generate null
  - One time in 4 we generate a leaf
  - 5 times out of 8 we generate a branch, with subtrees of half the size.
  ```haskell```
  btree' n | n>0
  = frequency [ (1,return Null)
    , (2,liftM Lf arbitrary)
    , (5,liftM3 Br subt arbitrary subt)
  ]
  where subt = btree' ( n 'div' 2)
  ```haskell```
- We use `frequency`, which is `oneof` extended with explicit probability information
  ```haskell```
  frequency :: [(Int,Gen a)] -> Gen a
  ```haskell```
The Invariant?

- What about the ordering invariant on binary trees?
- We disregarded it completely?
- Yet, tests still worked !
- Is it necessary ?
- How do we test for it ?

Defining the Invariant (I)

- We define the minimum element in a tree:
  - $btMIN\ Null = maxBound$
  - $btMIN\ (Lf\ x) = x$
  - $btMIN\ (Br\ left\ x\ right) = minimum\ [btMIN\ left,\ x,\ btMIN\ right]$
- And the maximum:
  - $btMAX\ Null = minBound$
  - $btMAX\ (Lf\ x) = x$
  - $btMAX\ (Br\ left\ x\ right) = maximum\ [btMAX\ left,\ x,\ btMAX\ right]$

Defining the Invariant (II)

- Null trees and leaves satisfy the invariant, so we just need to specify the case for a branch:
  - $invBTree\ (Br\ left\ x\ right)$
    - $= invBTree\ left$
    - $&& btMAX\ left < x \&& x < btMIN\ right$
    - $&& invBTree\ right$
  - $invBTree\ _ = True$

Testing Invariant, and Invariant Preservation

- We check the invariant against arbitrary trees — it should fail
  - prop_inv bt =
    - invBTree (bt :: BTree Int)
- We check that insert preserves the invariant
  - prop_ins_inv bt x
    - $= invBTree\ bt \Rightarrow invBTree\ (btINS x\ bt)$
    - where types $= (x::Int)$
- This test is not adequate !
  - only small trivial trees pass the invariant check
We shall generate trees that are ordered.
We shall:
1. create a random list
2. sort it
3. remove duplicates
4. turn this into a tree

```haskell
ordbtree :: (Ord a, Arbitrary a) => Gen (BTree a)
ordbtree = do alist <- arbitrary
              mkrandtree (nub (sort alist))
```

Empty lists produce empty trees
```
mkrandtree [] = return Null
```
Singleton lists produce leaves
```
mkrandtree [x] = return (Lf x)
```

For other lists we
1. select element at random
2. recursively turn elements before into a tree
3. recursively turn elements after into a tree
4. bind it all into a branch

```
mkrandtree xs
  = do splitix <- choose (0, length xs - 1)
       let (llist, (x: rlist)) = splitAt splitix xs
       ltree <- mkrandtree llist
       rtree <- mkrandtree rlist
       return (Br ltree x rtree)
```

We define a test that explicitly invokes our generator to check its output does satisfy the invariant
```
prop_ordtree_good = forAll ordbtree
  (invBTree :: BTree Int -> Bool)
```
We use our generator to check the lookup-insert property

```haskell
prop_ginv_lkp_ins x
  = forAll ordbtree lkp_ins
  where
    lkp_ins bt = btLKP x (btINS x bt)
    types = (x :: Int)
```