Formal Methods for Software Development

Till Mossakowski, Lutz Schröder

01.11.2004

Deficiencies of HUnit

- user has to supply test cases
- degree of coverage is unclear
- systematically writing tests can be tedious
- specification language limited to simple Boolean expressions

QuickCheck

- test cases are generated automatically, using random testing
- test cases are checked against specifications
- specifications also document the program
- test data generation language
- leightweight approach
- see `http://www.cs.chalmers.se/~rjmh/QuickCheck/`

QuickCheck Specification Language

- basic conditions: Boolean Haskell expressions
- conditinal properties
- universal quantification (over all finite, total datatype elements)

Example Specifications

```
prop_RevUnit x = reverse [x] == [x]
  where types = x::Int

prop_RevApp xs ys =
  reverse (xs++ys) ==
    reverse ys ++ reverse xs
  where types = (xs::[Int], ys::[Int])

prop_RevRev xs = reverse (reverse xs) == xs
  where types = xs::[Int]
```

Properties of Functions

```
prop_ComposeAssoc f g h x =
  ((f . g) . h) x == (f . (g . h)) x
  where types = [f, g, h] :: [Int->Int]
```

Conditional Laws

<condition> ==> <property>

Conditional Laws – Example: the Program

```
ordered xs =
   and (zipWith (<=) xs (drop 1 xs))

insert(x,[]) = [x]
insert(x,y:l) = if x <= y then x:y:l
                          else y:insert(x,l)
```

Conditional Laws – Example: the Specification

```
prop_Insert x xs =
  ordered xs ==> ordered (insert (x,xs))
  where types = x::Int

prop_Insert' = \(x::Int) -> \ xs ->
   (ordered xs ==> ordered (insert (x,xs)) )
```

Counting Trivial Cases

<condition> ‘trivial‘ <property>

```
prop_Insert2 x xs =
  ordered xs ==>
    null xs ‘trivial‘ ordered (insert (x,xs))
  where types = x::Int
```

Classifying Test Cases

```
  classify <condition> <string>$ <property>
```

```
prop_Insert3 x xs =
  ordered xs ==>
    classify (ordered (x:xs)) "at-head"$
    classify (ordered (xs++[x])) "at-tail"$
    ordered (insert (x,xs))
  where types = x::Int
```

Collecting Data Values

```
collect <expression>$ <property>
```

```
prop_Insert4 x xs =
        ordered xs ==>
            collect (length xs)$
            ordered (insert (x,xs))
  where types = x::Int
```

Quantified Properties

```
forAll <generator> $ \<pattern> -> <property>
```

```
prop_Insert5 x =
  forAll orderedList $ \xs ->
     ordered (insert (x,xs))
  where types = x::Int
```

`orderedList` is a test data generator.

Test Data Generators: The Type Gen

- Test data is produced by test data generators.
- QuickCheck defines default generators for most types.
- User-supplied generators: with `forAll`, and for any new types.
- Generators have types of the form `Gen a`, where `Gen` is a monad

The Class `Arbitrary`

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

The class method `arbitrary` is the default generator for type `a`.

Generators are built up on top of the function

```
choose :: Random a => (a, a) -> Gen a
```

which makes a random choice of a value from an interval, with a uniform distribution.

Example: Random Choice Between Elements of a List

```
oneof :: [Gen a] -> Gen a
oneof m =
 do xs <- sequence m
    i<-choose (0,length xs-1)
    return (xs!!i)
```

Generation of Ordered Lists

```
orderedList :: Gen [Int]
orderedList = orderedListAux 0
orderedListAux :: Int -> Gen [Int]
orderedListAux n = frequency
  [(1, return []),
   (4, do i <- arbitrary
          let j = abs i
          xs <- orderedListAux (n+j)
          return ((n+j):xs))]
```

Generators for User-Defined Types

```
data Colour = Red | Blue | Green

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

Generators for Lists

```
instance Arbitrary a => Arbitray [a] where
 arbitrary = oneof
  [return [], liftM2 (:) arbitrary arbitrary]

liftM2  :: (Monad m) =>
    (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
liftM2 f m1 m2 = do
   x1 <- m1
   x2 <- m2
   return (f x1 x2)
```

Better Distribution of Lists

```
instance Arbitrary a => Arbitray [a] where
 arbitrary = frequency
  [(1, return []),
   (4, liftM2 (:) arbitrary arbitrary) ]
```

Generation of Trees

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
instance Arbitrary a => Arbitray (Tree a)
 where
 arbitrary = frequency
  [(1, liftM Leaf arbitrary),
   (2, liftM2 Branch arbitrary arbitrary) ]
```

Problem: this definition only has a 50% chance of terminating!

Generation of Trees: Size Function

```
sized :: (Int -> Gen a) -> Gen a
instance Arbitrary a => Arbitrary (Tree a)
 where
 arbitrary = sized arbTree
arbTree 0 = liftM Leaf arbitrary
arbTree n = frequency
  [(1, liftM Leaf arbitrary),
   (4, liftM2 Branch subtree subtree)]
  where subtree = arbTree (n `div` 2)
```

Limits of QuickCheck

- Undefined values (can be handled via catching exceptions)
- Infinite datastructures (can be handled via finite parts)
- Monadic values: see next lecture
- No test case coverage analysis (but user can supply new test generators)
- Only a fragment of higher-order logic is covered