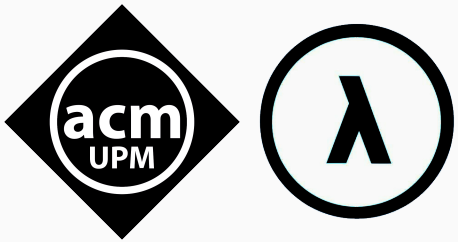

Property-Based Testing

Luis Eduardo Bueso de Barrio
lbueso@acm.org

14 de febrero de 2018

ACM PLIG

Índice

Introducción

El testing es el método de verificación de la calidad de software más usado.

Problema: consume gran cantidad de recursos (hasta el 50 % del coste del software).

¿Qué es el Property-Based Testing?

Definición

El *Property-Based Testing* es una metodología de testing que a través de la especificación de propiedades, el análisis del flujo de datos del programa y la ejecución de tests busca su corrección y completitud.

Responde a las siguientes preguntas:

- ¿Hasta dónde se puede llegar mediante el testing?
- ¿Qué datos se deben usar en los tests?
- ¿Cuándo se ha hecho suficiente testing?
- ¿Cómo se determina si un test es exitoso o fallido?

¿Qué ha conseguido hasta ahora?

El *Property-Based Testing* ha evolucionado mucho desde su concepción, y ahora es una metodología muy madura, que reduce en gran medida el precio de la generación de tests.

Ahora mismo existen gran cantidad de herramientas que utilizan esta metodología:

- Haskell *QuickCheck*.
- Erlang *QuickCheck*.
- Erlang *PropEr*.
- Scala *ScalaCheck*.
- Python *Hypothesis*.
- Java 8 *QuickTheories*.
- JavaScript *JSVerify*.

¿Qué ha conseguido hasta ahora?

El *Property-Based Testing* ha evolucionado mucho desde su concepción, y ahora es una metodología muy madura, que reduce en gran medida el precio de la generación de tests.

Ahora mismo existen gran cantidad de herramientas que utilizan esta metodología:

- **Haskell QuickCheck**.
- Erlang *QuickCheck*.
- Erlang *PropEr*.
- Scala *ScalaCheck*.
- Python *Hypothesis*.
- Java 8 *QuickTheories*.
- JavaScript *JSVerify*.

QuickCheck

¿Qué es QuickCheck?

QuickCheck es una herramienta de creación y testeo de propiedades sobre los programas (*Property-Based Testing*).

Permite la descripción de propiedades mediante funciones en Haskell que se prueban sobre una entrada aleatoria.

Permite la creación de generadores propios.

QuickCheck es un *DSL* (*Domain-Specific Language*) de especificación de tests.

Herramienta muy ligera, la implementación original tenía sobre 300 líneas de código.

Propiedades

Ejemplo:

```
        reverse [x] = [x]
    reverse (xs++ys) = reverse ys ++ reverse xs
reverse (reverse xs) = xs
```

Propiedades:

```
prop_RevUnit x = reverse [x] == [x]

prop_RevApp xs ys =
  reverse (xs++ys) == reverse ys ++ reverse xs

prop_RevRev xs =
  reverse (reverse xs) == xs
```

```
*Main Test.QuickCheck> quickCheck prop_RevUnit
+++ OK, passed 100 tests.
*Main Test.QuickCheck> quickCheck prop_RevApp
+++ OK, passed 100 tests.
*Main Test.QuickCheck> quickCheck prop_RevRev
+++ OK, passed 100 tests.
*Main Test.QuickCheck>
```

```
*Main Test.QuickCheck> verboseCheck prop_RevApp
Passed:
[]
[]

Passed:
[()]
[(),(),(),(),(),()]

Passed:
[(),(),(),(),()]
[(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),()]

...

+++ OK, passed 100 tests.
*Main Test.QuickCheck>
```

```
*Main Test.QuickCheck>
    verboseCheck (prop_RevApp :: [Int] -> [Int] -> Bool)
Passed:
[]
[]

Passed:
[-2,-5,-3,-1,-5,4]
[-2,-5,-3,-5,3]

Passed:
[13,-15,15,4,-12,1,5,3,2,-7,-14,-13,1,-12]
[-5,8,2,1,-3,2,13,-2,-10]

...

+++ OK, passed 100 tests.
*Main Test.QuickCheck>
```

Propiedades de funciones

Propiedad commutativa y asociativa del operador de composición

```
(f === g) x = f x == g x

prop_CompAssoc :: (Int -> Int) -> (Int -> Int) ->
  (Int -> Int) -> Int -> Bool
prop_CompAssoc f g h = (f . (g . h)) === ((f . g) . h)

prop_CompCommut :: (Int -> Int) -> (Int -> Int) ->
  Int -> Bool
prop_CompCommut f g = (f . g) === (g . f)

-- Necesario para que en caso de error no se lance una
-- excepción, ya que las funciones no se pueden imprimir
instance Show (a -> b) where show _ = "<<function>>"
```

```
*Main Test.QuickCheck> quickCheck prop_CompAssoc
+++ OK, passed 100 tests.
*Main Test.QuickCheck> quickCheck prop_CompCommut
*** Failed! Falsifiable (after 5 tests and 2 shrinks):
<<function>>
<<function>>
0
*Main Test.QuickCheck>
```

Condicionales

Operación de implicación (==>) para condicionales

Ejemplo:

```
-- La propiedad x <= y ==> max x y == y
prop_MaxLe :: Int -> Int -> Property
prop_MaxLe x y = x <= y ==> max x y == y

-- Propiedad que comprueba que insertar deja
-- la lista ordenada
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  isSorted xs ==> isSorted (insert x xs)
```

```
*Main Test.QuickCheck> quickCheck prop_MaxLe
+++ OK, passed 100 tests.
*Main Test.QuickCheck> quickCheck prop_Insert
*** Gave up! Passed only 88 tests.
*Main Test.QuickCheck>
```

Monitorización

También podemos monitorizar datos de los tests:

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  isSorted xs ==>
    classify (null xs) "trivial" $
      isSorted (insert x xs)

prop_Insert' :: Int -> [Int] -> Property
prop_Insert' x xs =
  isSorted xs ==>
    collect (length xs) $
      isSorted (insert x xs)
```

```
*Main> quickCheck prop_Insert
*** Gave up! Passed only 70 tests (34% trivial).
*Main> quickCheck prop_Insert'
*** Gave up! Passed only 80 tests:
40% 0
35% 1
21% 2
 2% 3
 1% 4
*Main>
```

Estructuras Infinitas

Hay una sutil diferencia entre estas dos propiedades:

```
-- Esta propiedad intenta comparar 2 listas infinitas
-- generadas por cycle por tanto no acaba
prop_DoubleCycle :: [Int] -> Property
prop_DoubleCycle xs =
  not (null xs) ==>
    cycle xs == cycle (xs ++ xs)

-- En cambio aquí suponemos que si los n primeros elementos
-- de las listas son iguales la propiedad es correcta
prop_DoubleCycle' :: [Int] -> Int -> Property
prop_DoubleCycle' xs n =
  not (null xs) && n >= 0 ==>
    take n (cycle xs) == take n (cycle (xs ++ xs))
```

```
*Main> quickCheck prop_DoubleCycle
(0 tests; 11 discarded)


----------------------------------------------------

*Main> quickCheck prop_DoubleCycle'
+++ OK, passed 100 tests.
*Main>
```

Generadores

Además de los generadores propios de QuickCheck se pueden implementar generadores propios.

Para implementar generadores se utiliza la *typeclass*:

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

Ejemplo:

```
data Colour = Red | Blue | Green deriving Show

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

```
*Main> sample (arbitrary :: Gen Colour)
Red
Red
Blue
Blue
Red
Blue
Green
Blue
Green
Red
Red
*Main>
```

Generadores

Podemos mejorar nuestros generadores controlando, por ejemplo, la frecuencia con la que se generan determinados valores.

```
data Colour = Red | Blue | Green deriving Show

instance Arbitrary Colour where
  arbitrary = frequency
    [ (1, return Red  ),
      (4, return Blue ),
      (2, return Green)]
```

```
*Main> sample (arbitrary :: Gen Colour)
Red
Blue
Blue
Blue
Red
Green
Blue
Red
Blue
Blue
Blue
*Main>
```

Modificadores

Podemos mejorar nuestros generadores aplicándoles modificadores, que aportan ciertas propiedades:

```
-- modificador que genera listas ordenadas
newtype OrderedList a

-- modificador que genera listas no vacías
newtype NonEmptyList a

-- modificador que genera listas infinitas
newtype InfiniteList a

-- modificador que genera números distintos a 0
newtype NonZero a
```

Para buscar más modificadores consultar *Test.QuickCheck.Modifiers*

```
*Main> sample (arbitrary :: Gen (OrderedList Int))
Ordered {getOrdered = []}
Ordered {getOrdered = [-2,2]}
Ordered {getOrdered = [-3,2]}
Ordered {getOrdered = [-5,-3,-1,2,4,4]}
Ordered {getOrdered = [-2,1,2,4]}
Ordered {getOrdered = [-10,-7,-7,-6,-4,-3,0,1,4,7]}
Ordered {getOrdered = [-4,-1,5,8,9]}
Ordered {getOrdered = [-9,9,13]}
Ordered {getOrdered = [-12,-10,-5,5,14,16]}
Ordered {getOrdered = []}
Ordered {getOrdered = [-20,-17,-15,-11,-10,-10,-2,1,1,3,5,7,7,
   11,15,19,19]}
*Main>
```

Modificadores

Con modificadores podemos conseguir que siempre se generen datos correctos para propiedades que requieren condiciones.

Ejemplo:

```
prop_Insert :: Int -> OrderedList Int -> Property
prop_Insert x xs =
    classify (null ys) "trivial" $
      isSorted (insert x ys)
  where ys = getOrdered xs

prop_Insert' :: Int -> OrderedList Int -> Property
prop_Insert' x xs =
    collect (length ys) $
      isSorted (insert x ys)
  where ys = getOrdered xs
```

```
*Main> quickCheck prop_Insert
+++ OK, passed 100 tests (4% trivial).
*Main> quickCheck prop_Insert'
+++ OK, passed 100 tests:
 7% 0
 5% 5
 5% 3
 4% 8
 4% 37
 4% 23
 4% 11
 3% 52
 3% 4

...

*Main>
```

Conclusiones y observaciones

Problemas:

- Definir propiedades en muchos casos, no es una tarea trivial.
- Los *side-effects* pueden ser difíciles de verificar.

A pesar de estos problemas, definir propiedades, nos aporta una gran recompensa.

Los lenguajes funcionales puros son más fáciles de testear.

Algunos ejemplos reales de uso de *QuickCheck*:

- Unificación.
- Propiedades en circuitos con *Lava*.
- *Pretty Printing*.
- Edison.

Referencias

Claessen, K., and Hughes, J.

Quickcheck: A lightweight tool for random testing of haskell programs.

SIGPLAN Not. 46, 4 (May 2011), 53–64.

Fink, G., and Bishop, M.

Property-based testing: A new approach to testing for assurance.

SIGSOFT Softw. Eng. Notes 22, 4 (July 1997), 74–80.

Hughes, J.

Quickcheck testing for fun and profit.

In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages* (Berlin, Heidelberg, 2007), PADL'07, Springer-Verlag, pp. 1–32.

¿Preguntas?