Análisis, validación y depuración de software

Práctica 2

QuickCheck

Alicia Villanueva

Curso 2017/2018

Introducción a QuickCheck

Índice

1. Introducción

QuickCheck es una herramienta para probar programas Haskell de forma automática. El programador debe proporcionar una serie de *especificaciones* del programa en forma de propiedades que las funciones deben satisfacer. A partir de estas especificaciones, QuickCheck prueba que las propiedades se satisfacen para un número determinado de casos de prueba generados de forma aleatoria. Es pues un ejemplo de herramienta de generación de casos de prueba basado en el método aleatorio.

Las especificaciones que queremos comprobar deben definirse en sintaxis Haskell, aunque en realidad se usan una serie de operadores definidos específicamente en la biblioteca QuickCheck. Estos operadores permiten en primer lugar definir propiedades, pero también nos permiten observar las características de los casos de prueba generados de forma que el programador puede evaluar la calidad de la prueba. Asimismo, existen operadores para definir generadores de casos de pruebas de forma *manual*, es decir, ideados por el programador. De esta forma se puede evitar la generación de demasiados casos de prueba *inútiles*.

1.1. Objetivo de la práctica

El objetivo de la práctica consiste en que los alumnos sean capaces de manejar la biblioteca QuickCheck para especificar propiedades y probarlas en los programas implementados.

1.2. Material adicional

A continuación damos una lista de material bibliográfico que puede servir de apoyo.

- *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, artículo de Koen Claessen y John Hughes, publicado en ICFP 2000.
- *Testing Monadic Code with QuickCheck*, artículo de Koen Claessen y John Hughes, publicado en el Workshop de Haskell en 2002.
- Manual en red:
  `http://www.cs.chalmers.se/~rjmh/QuickCheck/manual.html`
- Historia sobre la utilidad del *software testing*:
  `http://www.ellium.com/~thor/icfp2001/k5-icfp.html`
- Real World Haskell by Bryan O'Sullivan, Don Stewart, and John Goerzen. Chapter 11. Testing and quality assurance
  `http://book.realworldhaskell.org/read/testing-and-quality-assurance.html`

2. Trabajo a realizar y evaluación

1. Lee el boletín y haz los ejercicios propuestos. Cuando se planteen preguntas, respóndelas en el mismo fichero, como comentarios.

2. Guarda tus ejercicios en el disco local para tenerlos disponibles en la próxima sesión.

Debers realizar una prueba individual de PoliformaT durante los últimos quince minutos de las dos sesiones de laboratorio correspondientes a esta práctica. La prueba versará sobre los conceptos trabajados en este boletín.

3. QuickCheck

Como ya hemos dicho, QuickCheck es una herramienta cuyo objetivo es el de facilitar la tarea del programador a la hora de formular y probar las propiedades que debe satisfacer un programa. Las propiedades se describen mediante funciones Haskell y pueden ser probadas de forma *automática* ante casos de prueba generados de forma aleatoria.

La prueba de programas es uno de los métodos más usados para garantizar una determinada calidad del *software*. Se trata de una tarea bastante pesada que consume muchos recursos y tiempo. Aunque pueda creerse que la programación declarativa en general, y la funcional en particular, no necesitan la prueba de programas, en la práctica el *software testing* es fundamental y los programadores lo realizan cuando programan en lenguajes funcionales igual que lo harían si programaran en cualquier lenguaje imperativo.

Hemos elegido profundizar en la prueba de programas automática en el marco funcional porque nos permite mantenernos en un alto nivel de especificación, sin tener que preocuparnos de detalles de implementación. En otras asignaturas se ha profundizado en el diseño de casos de prueba para el *software*, aquí no vamos a diseñar, sino a usar una herramienta que diseña por nosotros (en este caso mediante el método aleatorio).

Con respecto al marco utilizado, las funciones puras son generalmente más sencillas de probar que las funciones que contienen *side-effects*. Si tenemos en cuenta que en un programa funcional normalmente casi todas las funciones son puras, encontramos una razón por la que probar programas funcionales de forma automática permite que nos centremos en los aspectos de validación y no de programación. Aun así, las funciones no puras también pueden probarse automáticamente aunque con un poco más de dificultad.

En cualquier método de prueba es necesario un mecanismo con el que se pueda decidir cuándo un caso de prueba/ejecución satisface la propiedad y cuándo no. En QuickCheck se elige una aproximación en la que el programador da una especificación formal usando un lenguaje especial definido para

este propósito. Estas especificaciones se integrarán en el mismo módulo donde se especifique el programa de forma que constituirán una documentación acerca de los requisitos que deberían cumplir las funciones.

QuickCheck fue diseñado para ser un método ágil. De hecho la implementación consta de unas 300 líneas de un módulo funcional puro integrado en Haskell. En la práctica, este módulo es usado desde el intérprete ghci. También puede usarse un *script* para invocarlo.

4. Definición de propiedades en QuickCheck

La sintaxis básica de definición de propiedades es la siguiente:

```
property ::= boolExp
           | \x -> property
           | boolExp ==> property
           | forAll set $ \x -> property
           | classify expr property
           | collect expr property

test     ::= quickCheck property
```

Iremos viendo el significado de la sintaxis en este boletín. De momento diremos que boolExp hace referencia a cualquier expresión de tipo Bool y que property hace referencia a un nuevo tipo de datos definido en QuickCheck muy similar al Bool. Insistimos en que las propiedades se escribirán junto con el código de nuestros programas.

Empezaremos mostrando un ejemplo muy sencillo. La función (predefinida) reverse invierte una lista dada. Esta función estándar satisface varias propiedades como:

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

Es decir, la lista inversa de la lista que tiene un único elemento es la misma lista; La lista inversa de la concatenación de dos listas es equivalente a hacer la inversa de la segunda lista y concatenarla con la inversa de la primera:

```
reverse ([1,2]++[3,4]) = reverse [3,4] ++ reverse [1,2] = [4,3,2,1]
```

Estas leyes o propiedades se satisfacen sólo para valores finitos y totales. De momento trabajaremos con estas restricciones.

Para poder probar estas propiedades, hay que expresarlas como funciones de Haskell con la sintaxis introducida arriba. El nombre de las funciones deberá empezar por prop_. Así pues para la primera propiedad descrita tendríamos:

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

que devolverá True si la propiedad se satisface. Nótese que la sintaxis es completamente compatible con la sintaxis de cualquier función Haskell: nombre de la función, parámetros de entrada y tras el signo de la igualdad el resultado de la función, que en este caso debe ser un booleano.

Ejercicio 1 *Escribe (de momento a lápiz) la segunda y tercera propiedad mencionadas arriba:*

prop_RevApp

prop_RevRev

A continuación vamos a ejecutar nuestra prueba. Las propiedades deben estar integradas en el módulo funcional que queremos probar. En nuestro caso, la función reverse viene ya definida en el Prelude, así que lo único que tendremos que especificar en nuestro programa serán las propiedades.

> ATENCIÓN: Nuestro módulo (o programa) deberá importar el módulo Test.QuickCheck.

Ejercicio 2 *Escribir las tres propiedades anteriores en un fichero que importe el módulo de QuickCheck*

Para ejecutar la prueba tendremos que cargar las funciones y propiedades en el intérprete. Para ello, en primer lugar debéremos invocar al intérprete:

```
$ ghci
```

Una vez lanzado el intérprete, tendremos que cargar nuestro programa con la opción :load o :l. Si no ocurren errores, ahora podremos ejecutar lo siguiente:

```
Main> quickCheck prop_RevApp
```

El resultado debería ser un mensaje indicando que el programa ha pasado la prueba.

La función quickCheck toma como parámetro una propiedad y ejecuta un número determinado de pruebas aleatorias. Por defecto dicho número es 100. Si todas las pruebas son satisfactorias, el resultado será OK, en caso contrario el sistema nos dará un contraejemplo, es decir, el caso de prueba que ha fallado. Hay que tener un poco de cuidado con los tipos de datos de las propiedades. La función reverse está sobrecargada, de forma que admite cualquier tipo de elemento en una lista. Cuando quickCheck se encuentra con una propiedad polimórfica, genera casos de prueba del tipo especial Unit de Haskell, que el tipo más sencillo del lenguaje. En concreto, los valores tomados son siempre (). Esto puede dar lugar a errores en las verificaciones de las propiedades.

Ejercicio 3 *Ejecuta nuevamente la prueba, pero en lugar de usar* quickCheck *emplearemos* verboseCheck*, que además de generar automáticamente los casos de prueba y tratar de verificar la propiedad, te muestra los casos de prueba generados en cada verificación. ¿Qué puedes concluir?.*

```
Main> verboseCheck prop_RevApp
```

Podemos resolver los posibles problemas simplemente especificando para qué tipo de dato queremos que se verifique la propiedad, por ejemplo en este caso podemos añadir la especificación de tipos de la propiedad como sigue:

```
prop_RevApp :: [Int] -> [Int] -> Bool
```

También podemos definir los tipos mediante la sentencia where types = (xs::[Int]), por ejemplo.

Ejercicio 4 *Añadir los tipos a la propiedad y volver a intentar probar la propiedad. Utilizar para ello tanto* quickCheck *como* verboseCheck*.*

Ejercicio 5 *Modifica la propiedad* prop_RevApp *del ejercicio 4 (con tipos) para hacerla incorrecta, intercambiando las variables* xs *e* ys *de un solo lado de la igualdad:*

```
prop_RevApp :: [Int]->[Int] -> Bool
prop_RevApp xs ys = reverse(xs++ys)==reverse(xs)++reverse(ys)
```

A continuación ejecuta la prueba y observa el resultado. ¿Qué representan los valores que aparecen en pantalla? Observa la salida con verboseCheck*, ¿qué crees que ha ocurrido?*

Elimina la especificación de tipos y vuelve a repetir la prueba. ¿Qué ocurre ahora? ¿Puedes dar una explicación? Ayúdate utilizando verboseCheck *si es necesario.*

4.1. Especificación de propiedades condicionales

Podemos definir propiedades regidas por una condición. La idea intuitiva es la de validar que el programa satisface una determinada propiedad siempre y cuando se cumpla una condición.

Por ejemplo, `max :: Int -> Int -> Int` es una función definida en el Prelude que devuelve el máximo de dos números. La siguiente propiedad

$$x <= y \Longrightarrow max\ x\ y == y$$

especifica que si x es menor o igual que y, entonces el máximo entre x e y es y. En QuickCheck podemos definirla de la siguiente forma:

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

Ejercicio 6 *Escribe la propiedad* MaxLe *en un módulo y ejecuta el test de la misma.*

Antes hemos dicho que las funciones que representan las propiedades devolverían True o False. Sin embargo, podemos observar que en el ejemplo anterior hemos definido la salida de la función de tipo Property. Esto es debido a que cuando definimos propiedades condicionales, la semántica del resultado es ligeramente distinta. El algoritmo, en vez de probar la propiedad para 100 casos de prueba aleatorios, lo que hace es ejecutar la propiedad para 100 casos de prueba *que satisfagan la propiedad*, es decir, casos de prueba *válidos*. Esto quiere decir que puede darse el caso en el que la propiedad no se satisfaga, casos que no cumplían la condición. Por este motivo se introduce el nuevo tipo de datos Property.

Ejercicio 7 *Dadas las siguientes funciones* miInsert *(que inserta un elemento en una lista ordenada) y* miOrdered *(que comprueba si una lista está ordenada):*

```
miInsert x [] = [x]
miInsert x (y:ys) | x < y = x : y : ys
                  | otherwise = y : (miInsert x ys)

miOrdered [] = True
miOrdered [x] = True
miOrdered (x:y:ys) = x <= y && miOrdered (y:ys)
```

Define una propiedad con el siguiente tipo

```
prop_Ordered :: Int -> [Int] -> Property
```

que comprobe que al insertar un elemento en una lista ordenada, se obtiene como resultado una lista ordenada.

Es posible que la salida al ejecutar la prueba del ejercicio anterior haya sido algo parecido a:

```
*** Gave up! Passed only 77 tests.
```

Esto ocurre cuando, tras un número determinado de intentos, QuickCheck no ha sido capaz de generar 100 casos de prueba que satisfagan la condición de la propiedad. Por lo tanto, debemos tener en cuenta que no se han ejecutado 100 casos de prueba, sino sólo 77 (en el ejemplo dado). Queda bajo nuestro criterio considerar suficiente o no este número de casos para concluir que la prueba ha sido pasada satisfactoriamente.

4.2. Tratando con estructuras infinitas

Hasta ahora hemos visto ejemplos de propiedades y programas que manejan estructuras de datos finitas. En Haskell tenemos la función cycle que toma como entrada una lista *no vacía* y devuelve una lista donde se repite el contenido de la lista de entrada infinitamente. En esta sección vamos a ver cómo tratar con propiedades que afectan a listas infinitas como la que genera la función cycle.

Empecemos por intentar probar la siguiente propiedad, donde decimos que el resultado de aplicar la función cycle a una lista determinada es el mismo que el de aplicarla a la concatenación de la lista de entrada consigo misma (una vez).

```
prop_DoubleCycle :: [Int] -> Property

prop_DoubleCycle xs = not (null xs) ==> cycle xs == cycle (xs ++ xs)
```

Si intentáis ejecutar esta prueba, el programa no terminará debido a que la comparación == entre listas infinitas no termina. Sin embargo, podemos usar la propiedad matemática que nos dice que dos listas infinitas serán iguales si todos sus prefijos finitos lo son:

```
prop_DoubleCycle :: [Int] -> Int -> Property

prop_DoubleCycle xs n = not (null xs) && n >= 0 ==> take n (cycle xs) ==
take n (cycle (xs ++ xs))
```

Esta prueba generará datos para xs y para n (los datos de entrada) y comprobará el resultado. Obviamente no se van a probar todos los casos posibles ya que al ser una lista infinita esto sería imposible.

Ejercicio 8 *Ejecuta la prueba descrita justo arriba y observa el resultado. ¿Qué conclusiones puedes sacar?. Puedes ayudarte observando la salida con* `verboseCheck`.

5. Mejorando la generación de casos de prueba

Como ya hemos mencionado, a veces, al intentar probar una propiedad condicional, podemos obtener el siguiente mensaje de salida:

```
*** Gave up! Passed only 64 tests.
```

Esto quiere decir que tras 1000 intentos (valor por defecto) de generar datos de entrada para los casos de prueba, no se ha conseguido encontrar 100 casos válidos (es decir, que satisfacieran la condición). En el caso concreto del mensaje anterior se habrían encontrado sólo 64 valores para los que la condición se cumple, y por tanto para los que se ejecuta la prueba. Es decisión del programador considerar si 64 casos de prueba son suficientes para

dar el programa por satisfactorio o bien es conveniente refinar la generación para obtener una prueba más sólida. Veremos en la siguiente sección cómo es posible refinar la generación de casos de prueba para evitar este problema.

Lo primero que vamos a ver es cómo podemos monitorizar los casos de prueba generados aleatoriamente por el sistema. Es decir, conocer algo más sobre sus características, lo que nos permitirá decidir si la prueba la damos por buena o necesitamos trabajarla más. Para ello podemos usar operadores definidos específicamente para este propósito como classify y collect (ver la sintaxis en la página 4).

Ejercicio 9 *Modifica la propiedad definida en el Ejercicio 7 (prop_Ordered) introduciendo la siguiente sentencia tras la condición:*

```
... ==> classify (null xs) "trivial" $ ...
```

Ejecuta la prueba y observa el resultado. ¿Crees que puedes dar la prueba como satisfactoria?. Ten en cuenta que null x devuelve cierto cuando la lista x es vacía.

El operador classify debe colocarse justo antes de la definición que se quiere probar y después de haber seleccionado/filtrado/generado los datos para los que se quiere ejecutar la prueba.

El operador classify nos permite identificar casos de prueba que cumplan algún requisito (en este caso que la lista sea vacía) y los etiqueta con un nombre. classify no afecta en absoluto a la semántica de la propiedad o del programa.

Otro operador interesante es collect que genera un histograma a partir de un factor determinado.

Ejercicio 10 *Sustituye la sentencia* classify *del ejercicio anterior por lo siguiente:*

```
... ==> collect (length xs) $ ...
```

Ejecuta la prueba y observa el resultado. ¿Qué conclusiones podemos sacar?. ¿Se te ocurre algún motivo para que tengamos esta distribución de casos de prueba?.

En la siguiente sección veremos cómo mejorar la calidad de la prueba realizada refinando el método de generación de casos que, hasta el momento, ha sido completamente aleatorio ya que se encargaba QuickCheck de generar datos para los parámetros de entrada de las propiedades.

6. Generadores de datos

Como ya hemos dicho, QuickCheck genera los casos de prueba de forma aleatoria. Para ello tiene definidos una serie de generadores, pero también

podemos definir nosotros unos generadores específicos y forzar a QuickCheck a usarlos para generar los datos.

Dependiendo del tipo de datos de cada función, la forma de definir un generador de datos cambia. QuickCheck usa el mecanismo de sobrecarga de Haskell para definir el generador de datos por defecto para cada tipo. Para poder generar elementos de un tipo determinado de forma arbitraria, el tipo de datos debe ser una instancia de la clase Arbitrary, lo que recordemos quiere decir que debe tener definidas una serie de operaciones. La definición de la clase es la siguiente:

```
class Arbitrary a where
    arbitrary :: Gen a
    coarbitrary :: a -> Gen b -> Gen b
```

siendo Gen un tipo abstracto que representa un generador de datos para el tipo a. La función coarbitrary sirve cuando queremos definir generadores para funciones, no sólo para datos. De momento no trabajaremos con esta segunda función ya que no es necesario para generar datos.

Existen generadores definidos ya en QuickCheck para los tipos Bool, Int, Integer, Float, Double, pares, triples, listas y funciones, lo que nos ha permitido trabajar con los ejemplos presentados hasta el momento. Para definir nuevos generadores, más adecuados para ciertas funciones, el programador deberá declarar una instancia de la clase Arbitrary.

Existen también algunas funciones que nos ayudan a generar datos. Podemos usar por ejemplo el operador primitivo de generación de datos que elegirá un número aleatorio dentro de un intervalo. Usando el resultado de esta función podremos definir otros. A continuación mostramos el perfil de la función que, como ves, devuelve un dato de tipo Gen Int.

```
choose :: (Int, Int) -> Gen Int
```

Para definir generadores más complejos necesitamos otros operadores. Para ello tenemos que basarnos en la clase Monad de Haskell. Recordemos que las mónadas eran un recurso que nos permitían definir acciones que no entraban en el marco de la programación funcional pura. En particular vimos ya en la primera práctica que el operador return construye un generador constante (*pone el nombre de la mónada antes del dato, en este caso el nombre de la mónada es Gen*)

```
return :: a -> Gen a
```

El operador >>= genera un dato a y se lo pasa al segundo argumento para generar un b. Puede verse como un *pipe*.

```
(>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

Como ejemplo vamos a ver cómo se define un generador para enteros y para pares. La sintaxis de la clase es la mostrada en el cuadro.

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

Definición de una instancia de la clase:

```
instance Arbitrary Int where
    arbitrary = choose (-10, 20)
```

Definición de una instancia de la clase para trabajar con tuplas. Se exige que los dos componentes de la tupla sean de la clase Arbitrary, y esto lo especificamos en la parte inicial del perfil de la instancia.

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
    arbitrary = liftM2 (,) arbitrary arbitrary
```

Este código es equivalente al siguiente, el cual quizás sea más fácil de entender al principio:

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
        arbitrary = do a <- arbitrary
                       b <- arbitrary
                       return (a,b)
```

Recordemos que la expresión do es una forma de expresar acciones secuenciales en el programa.

Vamos a ver un ejemplo más. El siguiente código (mostramos sólo la cabecera de la declaración de la instancia) generaría un método por defecto de generación mediante la recursión sobre el tipo:

```
instance Arbitrary a => Arbitrary [a] where ...
```

6.1. Generadores de tipos definidos por el usuario

Cuando el programador define una estructura de datos nueva, tendrá que proporcionar también un generador de datos para dicha estructura. Por fortuna, QuickCheck proporciona una serie de operadores que permiten hacer esta tarea de forma muy sencilla. Uno de estos operadores es oneof, que simplemente elige uno de los posibles valores dados en una lista con una distribución uniforme.

Imaginemos que tenemos un tipo de datos Colour:

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

Podemos definir el siguiente generador:

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

Otro ejemplo, esta vez para generar listas:

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

En este último ejemplo hemos usado la función liftM2 para combinar una cabeza y cola arbitrarias usando el operador :. Este generador en realidad no es demasiado apropiado, ya que genera listas de una longitud media de un elemento. Podemos mejorar la generación usando el operador frequency, que nos permite especificar la frecuencia con la que cada posible alternativa es elegida. De esta forma, el ejemplo anterior quedaría:

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

Esta solución tiene algún problema cuando, por ejemplo, tenemos estructuras de datos como árboles, ya que si asignamos una frecuencia demasiado baja a la elección de hojas o ramas, la probabilidad de que la estructura sea terminante es muy baja. La solución a este problema pasa por usar una noción de tamaño de los datos generados implícita en ellos. Este parámetro *size* permitirá tanto limitar los casos de datos infinitos, como asegurar que vamos a generar datos lo suficientemente complejos o grandes como para detectar errores.

Podemos usar el tamaño de los datos usando la función sized:

```
sized :: (Int -> Gen a) -> Gen a
```

Por ejemplo, sized g llama a g pasándole como parámetro el tamaño actual. Si quisiéramos generar números naturales comprendidos en el rango [0,size], pondríamos:

```
sized $ \n -> choose (0, n)
```

Existe una función que modifica el valor del parámetro *size*:

```
resize :: Int -> Gen a -> Gen a
```

Por ejemplo, resize n g invoca al generador g con un parámetro de tamaño n. Vamos a mostrar un último ejemplo: generar una matriz aleatoria. Para ello un tamaño adecuado para las pruebas podría ser la raíz cuadrada del tamaño original:

```
matrix = sized $ \n -> resize (round (sqrt n)) arbitrary
```

La sintaxis básica para la definición de generadores es la siguiente:

```
gen  ::=  return expr
      |   do { x <- gen}* gen
      |   arbitrary
      |   oneof [gen*]
      |   frequency [(int,gen)*]
```

6.2. Definición de generadores para estructuras recursivas

Usando la noción de tamaño descrita arriba podemos definir generadores adecuados para tipos de datos recursivos. Supongamos que tenemos un tipo de datos que representa un árbol:

```
data Tree = Leaf Int | Branch Tree Tree
```

Podríamos definir el generador como:

```
tree = oneof [ liftM Leaf arbitrary, liftM2 Branch tree tree ]
```

Pero este generador podría definir un árbol infinito, o datos demasiado grandes para la prueba. El generador se puede refinar usando el parámetro del tamaño explicado en la sección anterior:

```
instance Arbitrary Tree where
    arbitrary = sized tree' where
      tree' 0 = liftM Leaf arbitrary
      tree' n | n>0 = oneof [ liftM arbitrary, liftM2 subtree subtree ] where
        subtree = tree' (n 'div' 2)
```

6.3. Otros operadores

Supongamos que tenemos un generador de datos `g` para el tipo `t`, podemos usarlo en combinación con una serie de operadores que facilitan la tarea al programador:

- `two g` genera un par de elementos de tipo `t`
- `three g` genera un trío de elementos de tipo `t`
- `four g` genera cuatro elementos de tipo `t`
- `vector n g` genera una lista de `n` elementos de tipo `t`

Mencionaremos un último operador: dada una lista `xs`, `elements xs` genera un elemento arbitrario de `xs`.

6.4. Ejemplo concreto

Vamos a ver a continuación un ejemplo concreto de definición de generador para el programa `miInsert`. Este generador solucionará los problemas de distribución que habíamos detectado mediante el uso de `collect` ya que generará listas ordenadas:

La nueva propiedad que queremos verificar es la siguiente:

```
prop_miInsert :: Int -> Property

prop_miInsert x = forAll miOrderedList $ \xs -> miOrdered (miInsert x xs)
```

Atención al tipo de la propiedad, ya que como esta vez pretendemos generar la lista usando el generador, sólo tomará un parámetro de entrada, el elemento que se quiere insertar. orderedList es un generador de datos definido como sigue:

```
miOrderedList :: (Ord a, Arbitrary a) => Gen [a]

miOrderedList = oneof [ return [],
                        do
                        xs <- miOrderedList
                        n <- arbitrary
                        return ((case xs of
                                    [] -> n
                                    x:_ -> n `min` x):xs)
                        ]
```

Ejercicio 11 *Introducir el ejemplo descrito en esta última sección y ejecutarlo.*

Añadir a la condición lo necesario para observar la distribución de los casos de prueba en cuanto a la longitud de la cadena generada por orderedList. ¿Qué conclusiones podemos sacar?.

Existe otra forma de definir el generador miOrderedList:

```
miOrderedList2 :: (Num a, Arbitrary a) => Gen [a]

miOrderedList2 = do
                    n <- arbitrary
                    listFrom n where
                            listFrom n = frequency [(1, return []),
                                        (4, do
                                                m <- arbitrary
                                                ns <- listFrom (n+abs m)
                                                return (n:ns))]
```

Ejercicio 12 *Introducir esta segunda implementación en el programa y comparar los resultados con los obtenidos en el ejercicio anterior.*

Ejercicio 13 *Definir una función de generación de datos orderedList usando la función sort. Analizar de nuevo el algoritmo miInsert y comparar los resultados obtenidos con esta nueva versión de orderedList con las dos versiones anteriores.*

7. Caso de estudio: La cola

Una cola contiene una secuencia de valores. Se pueden añadir elementos en la cola de la cola y quitar elementos del principio de la cola. Tradicionalmente, toda implementación de una cola contiene las siguientes funciones:

```
type Queue a = [a]

empty :: Queue a
isEmpty :: Queue a -> Bool
add :: a -> Queue a -> Queue a
front :: Queue a -> a
remove :: Queue a -> Queue a
```

Ejercicio 14 *Escribe la especificación de estas funciones de la forma más intuitiva posible y da una implementación (escribe el cuerpo de las funciones).*

Ejercicio 15 *Prueba que la implementación es correcta, para ello debes escribir y ejecutar las propiedades que deben satisfacerse para cada función.*

Es posible implementar una cola de una forma más eficiente. Podemos dividir una cola en dos listas, de forma que la primera mitad de la cola la almacenaremos en la primera lista, mientras que en la segunda almacenaremos la inversa del resto de la cola. Esto nos permite poder quitar elementos de la cola de forma rápida, pero también podemos añadir elementos a la cola de forma rápida, cosa que no era posible con la implementación anterior.

Algunas propiedades de esta implementación son que la primera lista nunca puede ser la lista vacía si la segunda lista contiene algún elemento. Así pues, al quitar un elemento de la cola será necesario comprobar si la primera lista es la lista vacía y en tal caso habrá que pasar el contenido de la segunda lista a la primera.

Ejercicio 16 *Implementa la versión optimizada de la cola sabiendo que ahora la representaremos con el tipo* `type QueueF a = ([a],[a])`*. Se deben implementar las funciones* `empty`, `isEmpty`, `add`, `front` *y* `remove`.

Ejercicio 17 *Prueba que la implementación es correcta. Para ello se podrán probar las mismas propiedades consideradas en el ejercicio 15.*

Supongamos que queremos usar la primera implementación de las colas como especificación de referencia y la segunda como implementación funcional. En este caso querríamos probar que las dos implementaciones se comportan de forma idéntica ante idénticas entradas. Pero los tipos que representan las colas son distintos, por lo que antes que nada se debe establecer la relación entre una cola representada como una lista y una cola representada por las dos listas de la segunda implementación.

Ejercicio 18 *Define una función* trans *que convierta una cola de tipo* QueueF *en una cola de tipo* Queue*.*

Ejercicio 19 *Define una función* igual *cuyo tipo sea* igual :: Queue Int -> QueueF Int -> Bool *que devuelva cierto si las dos colas son iguales y falso en cualquier otro caso.*

Ejercicio 20 *Define las propiedades necesarias que comprueben que las funciones de ambas implementaciones dan los mismos resultados. Se tendrá que definir una propiedad por función y se podrá usar (o no, dependerá de la función) la función* igual*.*