Programación Funcional Avanzada

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

Correctitud y Testing

Correctitud

- programa cumple con su especificación

Testing

- permite ganar confianza en la correctitud de un programa
- mostrar que los casos comunes trabajan correctamente
- mostrar que los casos de borde trabajan correctamente
- NO prueba la ausencia de bugs

Objetivos

Razonamiento Ecuacional

- entender como probar propiedades simples usando razonamiento ecuacional

QuickCheck

- entender como definir propiedades en QuickCheck y como usar QuickCheck
- entender como trabaja QuickCheck

Razonamiento Ecuacional

Transparencia Referencial

"Iguales pueden ser sustituídos por iguales"

Una expresión con un valor en un contexto puede ser sustituída por cualquier otra, con el mismo valor en el mismo contexto, sin afectar el significado del programa.

$$e_1 == e_2 \implies \mathcal{C}[e_1] == \mathcal{C}[e_2]$$

Asignación y Transparencia Referencial

Lenguajes con características imperativas **no** tienen transparencia referencial. Ese es el caso de lenguajes funcionales de la familia ML (Standard ML, OCaml):

$$
\begin{array}{l}
\text{let val } x \quad = \text{ref } 0 \\
\quad \text{fun } f\ n = (x := !x + n; !x) \\
\text{in } f\ 1 + f\ 2
\end{array}
$$

La expresión evalúa a 4.

El valor de $f\ 1$ es 1. Si sustituimos $f\ 1$ por 1,

$$
\begin{array}{l}
\text{let val } x \quad = \text{ref } 0 \\
\quad \text{fun } f\ n = (x := !x + n; !x) \\
\text{in } 1 + f\ 2
\end{array}
$$

resulta que ahora la expresión evalúa a 3.

Asignación y Transparencia Referencial (2)

En este otro ejemplo,

$$
\begin{array}{l}
\text{let val } x \quad = \text{ref } 0 \\
\quad \text{fun } f\ n = (x := !x + n; !x) \\
\text{in } \ f\ 1 + f\ 1
\end{array}
$$

tampoco podemos reemplazar ambas ocurrencias de $f\ 1$ por una variable:

$$
\begin{array}{l}
\text{let val } x \quad = \text{ref } 0 \\
\quad \text{fun } f\ n = (x := !x + n; !x) \\
\quad \text{val } r \quad = f\ 1 \\
\text{in } r + r
\end{array}
$$

Transparencia Referencial en Haskell

Haskell tiene transparencia referencial

Los efectos laterales son visibles

```
do
  x ← newIORef 0
  let f n = do modifyIORef x (+n); readIORef x
  r ← f 1
  s ← f 2
  return (r + s)
```

El tipo de f en Standard ML es $Int \to Int$

El tipo de f en Haskell es $Int \to IO\ Int$

Transparencia Referencial en Haskell (2)

Gracias a la transparencia referencial las definiciones de funciones nos dan reglas para razonar sobre los programas Haskell

Las propiedades sobre tipos de datos se pueden probar usando:

- tipos no recursivos $\rightarrow$ pruebas por casos
- tipos recursivos $\rightarrow$ inducción

$$\text{data } [a] = [] \mid a : [a]$$

Para probar $\forall\ (xs :: [a]).P\ xs$, se prueba:

- $P\ []$
- $\forall\ (x :: a)\ (xs :: [a]).P\ xs \rightarrow P\ (x : xs)$

Ejemplo de razonamiento ecuacional

$$
\begin{aligned}
&\mathit{isort}\ [\,] = [\,] \\
&\mathit{isort}\ (x : xs) = \mathit{insert}\ x\ (\mathit{isort}\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{insert}\ x\ [\,] = [x] \\
&\mathit{insert}\ x\ (y : ys) \mid x \leqslant y \qquad = x : y : ys \\
&\qquad\qquad\qquad\qquad\ \mid \mathit{otherwise} = y : \mathit{insert}\ x\ ys
\end{aligned}
$$

Teorema: Ordenamiento preserva el largo

$$\forall\ (xs :: [a]).\mathit{length}\ (\mathit{isort}\ xs) \equiv \mathit{length}\ xs$$

Lema:

$$\forall\ (x :: a)\ (ys :: [a]).\mathit{length}\ (\mathit{insert}\ x\ ys) \equiv 1 + \mathit{length}\ ys$$

Prueba del Lema

$$\forall\ (x :: a)\ (ys :: [a]).length\ (insert\ x\ ys) \equiv 1 + length\ ys$$

Prueba por inducción en la lista

Caso $ys \equiv [\,]$:

$$length\ (insert\ x\ [\,])$$

$\equiv$ {definición de *insert*}

$$length\ [x]$$

$\equiv$ {definición de *length*}

$$1 + length\ [\,]$$

Prueba del Lema (2)

Caso $ys \equiv y : ys'$, caso $x \leqslant y$:

$$length\ (insert\ x\ (y : ys'))$$

$\equiv$ {definición de *insert*}

$$length\ (x : y : ys')$$

$\equiv$ {definición de *length*}

$$1 + length\ (y : ys')$$

Prueba del Lema (3)

Caso $ys \equiv y : ys'$, caso $x > y$:

$$\begin{aligned}
& \mathit{length}\ (\mathit{insert}\ x\ (y : ys')) \\
\equiv\ & \{\text{definición de } \mathit{insert}\} \\
& \mathit{length}\ (y : \mathit{insert}\ x\ ys') \\
\equiv\ & \{\text{definición de } \mathit{length}\} \\
& 1 + \mathit{length}\ (\mathit{insert}\ x\ ys') \\
\equiv\ & \{\text{hipótesis}\} \\
& 1 + (1 + \mathit{length}\ ys') \\
\equiv\ & \{\text{definición de } \mathit{length}\} \\
& 1 + (\mathit{length}\ (y : ys'))
\end{aligned}$$

Prueba del Teorema

$$\boxed{\forall\,(xs :: [a]).length\;(isort\;xs) \equiv length\;xs}$$

Prueba por inducción en la lista

Caso $xs \equiv [\,]$:

$$length\;(isort\;[\,])$$

$\equiv$ {definición de *isort*}

$$length\;[\,]$$

Prueba del Teorema (2)

Caso $xs \equiv x : xs'$:

$$\mathit{length}\ (\mathit{isort}\ (x : xs'))$$

$\equiv$ {definición de *isort*}

$$\mathit{length}\ (\mathit{insert}\ x\ (\mathit{isort}\ xs'))$$

$\equiv$ {Lema}

$$1 + \mathit{length}\ (\mathit{isort}\ xs')$$

$\equiv$ {Hipótesis}

$$1 + \mathit{length}\ xs'$$

$\equiv$ {definición de length}

$$\mathit{length}\ (x : xs')$$

Algunas conclusiones

- Razonamiento ecuacional es una forma elegante de probar propiedades de un programa
- Puede ser usado para establecer una relación entre un programa Haskell "obviamente correcto" (la *especificación*) y uno eficiente (la *implementación*)
- Usualmente las pruebas son muy largas
- Hay que tener cuidado con casos especiales (lazyness):
  - valores indefinidos
  - valores infinitos
- No es viable probar propiedades de cada programa Haskell

Otros métodos de pruebas

Sistemas de tipos

- ej: Theorems for free! [P.Wadler'89]

Asistentes de pruebas

- ej: Coq, Agda, Idris

QuickCheck

QuickCheck

Es una biblioteca Haskell desarrollada por Koen Claessen y John Hughes

Permite definir propiedades

Generación automática de datos de prueba aleatorios en base a los tipos de datos

Extensible por el usuario

Reduce los casos de prueba que fallan

Usando QuickCheck

Para usar QuickCheck:

```
import Test.QuickCheck
```

La interfaz más sencilla es:

```
quickCheck :: Testable prop => prop -> IO ()
```

```
class Testable prop where
  property :: prop -> Property
instance Testable Bool
instance (Arbitrary a, Show a, Testable prop)
         => Testable (a -> prop)
```

Clases e instancias

Las clases declaran predicados sobre los tipos

class *Testable prop* where
  property :: *prop* $\rightarrow$ *Property*

Un tipo puede ser *Testable* o no

Si un predicado se cumple para un tipo, entonces los métodos de esa clase son soportados por el tipo

Para cada tipo *prop* tal que *Testable prop*, hay un método *property* :: *prop* $\rightarrow$ *Property*

Fuera de la declaración de la clase, el tipo es:

property :: *Testable prop* $\Rightarrow$ *prop* $\rightarrow$ *Property*

Clases e instancias (2)

Las instancias declaran cuales tipos pertenecen al predicado

$$\begin{aligned}&\text{instance } \mathit{Testable\ Bool}\\&\text{instance } (\mathit{Arbitrary\ a}, \mathit{Show\ a}, \mathit{Testable\ prop})\\&\qquad\Rightarrow \mathit{Testable}\ (a \rightarrow \mathit{prop})\end{aligned}$$

Los *Bool* están en *Testable*
Las funciones $a \rightarrow prop$ están en *Testable* si *prop* pertenece a *Testable* y *a* pertenece a *Arbitrary* y *Show*

Las declaraciones de instancias deben proveer implementaciones de los métodos de la clase, como prueba de que el predicado se cumple para el tipo.

Otras funciones que usan los métodos de una clase heredan sus restricciones

$$\mathit{quickCheck} :: \mathit{Testable\ prop} \Rightarrow \mathit{prop} \rightarrow \mathit{IO}\ ()$$

Propiedades "Nularias"

instance *Testable Bool*

sortAscending :: *Bool*
sortAscending = *sort* $[2, 1] \equiv [1, 2]$

sortDescending :: *Bool*
sortDescending = *sort* $[2, 1] \equiv [2, 1]$

Ejecutando QuickCheck:

```
$ quickCheck sortAscending
+++ OK, passed 1 test.
$ quickCheck sortDescending
*** Failed! Falsifiable (after 1 test):
```

Propiedades “Nularias” (2)

Son propiedades estáticas

QuickCheck puede ser utilizado para tests unitarios

Por defecto verifica 100 casos (configurable)

Propiedades Funcionales

$$\text{instance } (\mathit{Arbitrary}\ a, \mathit{Show}\ a, \mathit{Testable}\ \mathit{prop}) \Rightarrow \mathit{Testable}\ (a \to \mathit{prop})$$

$$\mathit{sortPreservesLength} :: ([\mathit{Int}] \to [\mathit{Int}]) \to [\mathit{Int}] \to \mathit{Bool}$$
$$\mathit{sortPreservesLength}\ \mathit{fsort}\ xs = \mathit{length}\ (\mathit{fsort}\ xs) \equiv \mathit{length}\ xs$$

```
$ quickCheck (sortPreservesLength isort)
+++ OK, passed 100 tests.
```

Propiedades parametrizadas
QuickCheck genera listas de enteros automáticamente

Una función de ordenamiento incorrecta

import *Data.Set*
setSort = *toList* ∘ *fromList*

```
$ quickCheck (sortPreservesLength setSort)
*** Failed! Falsifiable (after 6 tests and 2 shrinks):
[1, 1]
```

La función *setSort* elimina elementos duplicados, por lo que una lista con elementos duplicados causa una falla

QuickCheck muestra evidencia del fallo, intentando presentar casos mínimos

Especificación completa del ordenamiento

Propiedad 1: Una lista ordenada debe estar ordenada

$$
\begin{aligned}
&\mathit{sortOrders} :: [\mathit{Int}] \to \mathit{Bool} \\
&\mathit{sortOrders}\ xs = \mathit{ordered}\ (\mathit{sort}\ xs) \\
&\mathit{ordered} :: \mathit{Ord}\ a \Rightarrow [a] \to \mathit{Bool} \\
&\mathit{ordered}\ [\,] \quad = \mathit{True} \\
&\mathit{ordered}\ [x] = \mathit{True} \\
&\mathit{ordered}\ (x : y : ys) = x \leqslant y \ \wedge\ \mathit{ordered}\ (y : ys)
\end{aligned}
$$

Especificación completa del ordenamiento (2)

Propiedad 2: Una lista ordenada debe tener los mismos elementos que la lista original

$$sortPreservesElements :: [Int] \to Bool$$
$$sortPreservesElements\ xs = sameElements\ xs\ (sort\ xs)$$

$$sameElements :: Eq\ a \Rightarrow [a] \to [a] \to Bool$$
$$sameElements\ xs\ ys = null\ (xs \setminus\setminus ys)\ \wedge\ null\ (ys \setminus\setminus xs)$$

Información sobre los casos de prueba

$$collect :: (Testable\ prop, Show\ a) \Rightarrow a \rightarrow prop \rightarrow Property$$

La función *collect* reune estadísticas sobre los casos de prueba. Esa información se muestra cuando se pasa un test.

```
$ let p = sortPreservesLength isort
$ quickCheck (\xs -> collect (null xs) (p xs))
+++ OK, passed 100 tests:
92% False
8% True
```

Información sobre los casos de prueba (2)

```
$ quickCheck (\xs -> collect (length xs `div` 10) (p xs))
+++ OK, passed 100 tests:
31% 0
24% 1
16% 2
9% 4
9% 3
4% 8
4% 6
2% 5
1% 7
```

Información sobre los casos de prueba (3)

En un caso extremo se pueden mostrar todos los datos testeados:

```
$ quickCheck (\xs -> collect xs (p xs))
+++ OK, passed 100 tests:
6% [ ]
1% [9, 4,-6, 7]
1% [9,-1, 0,-22, 25, 32, 32, 0, 9, ...
...
```

Especificando los tipos

¿Todas las listas son ordenadas?

```
$ quickCheck ordered
+++ OK, passed 100 tests.
```

Usar type signatures para estar seguros de que se usan tipos más prudentes

```
$ quickCheck (ordered :: [Int] -> Bool)
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
[0,-1]
```

Implicancia

Dada la siguiente implementación de *replicate*

$$
\begin{aligned}
&\mathit{replicate'} :: \mathit{Int} \to a \to [a] \\
&\mathit{replicate'}\ 0\ \_ = [\,] \\
&\mathit{replicate'}\ n\ x = x : \mathit{replicate'}\ (n-1)\ x
\end{aligned}
$$

Si definimos la propiedad:

$$
\begin{aligned}
&\mathit{repRev\_bad} :: \mathit{Eq}\ a \Rightarrow \mathit{Int} \to a \to \mathit{Bool} \\
&\mathit{repRev\_bad}\ i\ a = \mathit{replicate'}\ i\ a \equiv \mathit{reverse}\ (\mathit{replicate'}\ i\ a)
\end{aligned}
$$

```
$ quickCheck repRev_bad
```

No termina!

Implicancia (2)

Se puede resolver usando el operador implica de QuickCheck

$$(==>) :: (\mathit{Testable}\ \mathit{prop}) \Rightarrow \mathit{Bool} \rightarrow \mathit{prop} \rightarrow \mathit{Property}$$
instance *Testable Property*

El tipo *Property* permite no sólo codificar *True* y *False*, sino que también permite rechazar casos de prueba

$$\mathit{repRev} :: \mathit{Eq}\ a \Rightarrow \mathit{Int} \rightarrow a \rightarrow \mathit{Property}$$
$$\mathit{repRev}\ i\ a = i > 0 ==> \mathit{replicate'}\ i\ a \equiv \mathit{reverse}\ (\mathit{replicate'}\ i\ a)$$

```
$ quickCheck repRev
+++ OK, passed 100 tests; 141 discarded.
```

Implicancia (3)

La función *insert* preserva una lista ordenada:

$$iPO :: Int \to [Int] \to Property$$
$$iPO\ x\ xs = ordered\ xs ==> ordered\ (insert\ x\ xs)$$

```
$ quickCheck iPO
*** Gave up! Passed only 72 tests; 1000 discarded tests.

$ quickCheck (\x xs -> collect (length xs) (iPO x xs))
*** Gave up! Passed only 77 tests; 1000 discarded tests:
38% 0
27% 1
21% 2
12% 3
 3% 4
```

La precondición es muy fuerte!

Configuración de QuickCheck

$$quickCheckWith :: Testable\ prop \Rightarrow Args \rightarrow prop \rightarrow IO\ ()$$

```
data Args
   = Args { replay :: Maybe (QCGen, Int)
          , maxSuccess :: Int
          , maxDiscardRatio :: Int
          , maxSize :: Int
          , chatty :: Bool
          , maxShrinks :: Int }
```

Al incrementar el número de intentos puede funcionar la implicancia

Una mejor solución es usar un generador a medida

Generadores

QuickCheck incluye *generadores básicos* para tipos pre-definidos y *combinadores* para construir nuevos generadores.

Mónada *Gen a*: generadores de valores de tipo *a*.

Combina un generador pseudo-aleatorio y una medida de tamaño para estructuras de datos.

Arbitrary

En la clase *Arbitrary* se define el generador y el método para "achicar" valores

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a → [a]
```

Se provee de instancias para la mayoría de los tipos en base: (), Bool, Int, Integer, Float, Double, Char, [a], Maybe a, etc.

Combinadores para generadores

$$
\begin{aligned}
&\mathit{choose} && :: \mathit{Random}\ a \Rightarrow (a, a) \to \mathit{Gen}\ a \\
&\mathit{oneof} && :: [\mathit{Gen}\ a] \to \mathit{Gen}\ a \\
&\mathit{frequency} && :: [(\mathit{Int}, \mathit{Gen}\ a)] \to \mathit{Gen}\ a \\
&\mathit{elements} && :: [a] \to \mathit{Gen}\ a \\
&\mathit{shuffle} && :: [a] \to \mathit{Gen}\ [a] \\
&\mathit{suchThat} && :: \mathit{Gen}\ a \to (a \to \mathit{Bool}) \to \mathit{Gen}\ a \\
&\mathit{resize} && :: \mathit{Int} \to \mathit{Gen}\ a \to \mathit{Gen}\ a \\
&\mathit{sized} && :: (\mathit{Int} \to \mathit{Gen}\ a) \to \mathit{Gen}\ a
\end{aligned}
$$

Generadores simples

```
instance Arbitrary Bool where
  arbitrary = choose (False, True)
```

```
data Dir = Norte | Sur | Este | Oeste
instance Arbitrary Dir where
  arbitrary = elements [Norte, Sur, Este, Oeste]
```

```
instance (Arbitrary a, Arbitrary b) ⇒ Arbitrary (a, b) where
  arbitrary = do x ← arbitrary
                 y ← arbitrary
                 return (x, y)
```

Generando números

Una posibilidad:

instance *Arbitrary Int* where
  arbitrary = *choose* $(-20, 20)$

Mejor:

instance *Arbitrary Int* where
  arbitrary = *sized* $(\lambda n \rightarrow$ *choose* $(-n, n))$

QuickCheck incrementa el tamaño automáticamente de forma gradual, hasta llegar al máximo valor configurado

Generando listas

Generadores:

$$
\begin{aligned}
&\mathit{listOf} && :: \mathit{Gen}\ a \to \mathit{Gen}\ [a] && \text{-- largo aleatorio}\\
&\mathit{listOf1} && :: \mathit{Gen}\ a \to \mathit{Gen}\ [a] && \text{-- no vacias, largo aleatorio}\\
&\mathit{vectorOf} && :: \mathit{Int} \to \mathit{Gen}\ a \to \mathit{Gen}\ [a] && \text{-- largo dado}\\
&\mathit{orderedList} && :: (\mathit{Ord}\ a, \mathit{Arbitrary}\ a) \Rightarrow \mathit{Gen}\ [a] && \text{-- lista ordenada}
\end{aligned}
$$

Generando listas definidas por el usuario

$$\text{data } List\ a = Nil \mid Cons\ a\ (List\ a)$$

Una posibilidad:

$$\begin{aligned}
&genCons :: Arbitrary\ a \Rightarrow Gen\ (List\ a)\\
&genCons = \text{do } v \leftarrow arbitrary\\
&\qquad\qquad\quad\ \ l \leftarrow arbitrary\\
&\qquad\qquad\quad\ \ return\ (Cons\ v\ l)\\
&genNil \quad = return\ Nil\\
&\text{instance } Arbitrary\ a \Rightarrow Arbitrary\ (List\ a) \text{ where}\\
&\quad arbitrary = oneof\ [genCons, genNil]
\end{aligned}$$

Largo promedio es uno!

Generando listas definidas por el usuario (2)

```
instance Arbitrary a ⇒ Arbitrary (List a) where
  arbitrary = sized genSizedList

genSizedList :: Arbitrary a ⇒ Int → Gen (List a)
genSizedList 0 = return Nil
genSizedList n = do v ← arbitrary
                    l ← genSizedList (n − 1)
                    return (Cons v l)
```

Generando árboles

Una posibilidad:

$$\text{data } \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\begin{aligned}&\text{instance } \mathit{Arbitrary}\ a \Rightarrow \mathit{Arbitrary}\ (\mathit{Tree}\ a) \text{ where}\\ &\quad \mathit{arbitrary} = \mathit{frequency}\ [(1, \mathit{liftM}\ \ \mathit{Leaf}\ \ \mathit{arbitrary})\\ &\qquad\qquad\qquad\qquad , (2, \mathit{liftM2}\ \mathit{Node}\ \mathit{arbitrary}\ \mathit{arbitrary})]\end{aligned}$$

donde:

$$\begin{aligned}\mathit{liftM}\ \ &:: (a \to b) \to \mathit{Gen}\ a \to \mathit{Gen}\ b\\ \mathit{liftM2} &:: (a \to b \to c) \to \mathit{Gen}\ a \to \mathit{Gen}\ b \to \mathit{Gen}\ c\end{aligned}$$

Puede llevar tiempo infinito!

Generando árboles (2)

```
instance Arbitrary a ⇒ Arbitrary (Tree a) where
  arbitrary = sized arbitraryTree

arbitraryTree :: Arbitrary a ⇒ Int → Gen (Tree a)
arbitraryTree 0 = liftM Leaf arbitrary
arbitraryTree n = frequency [(1, liftM  Leaf arbitrary)
                            ,(4, liftM2 Node t t)]
  where t = arbitraryTree (n `div` 2)
```

Reducción de casos

El otro método en *Arbitrary* es

$$shrink :: (Arbitrary\ a) \Rightarrow a \rightarrow [a]$$

- Mapea cada valor con un cierto número de valores estructuralmente menores

```
$ shrink 2
[0,1]
$ shrink [1,2]
[[],[2],[1],[0,2],[1,0],[1,1]]
```

- Ante un caso fallido, se aplica *shrink* hasta que no se pueda obtener un caso fallido menor