Tema 7: Razonamiento sobre programas

Informática (2009–10)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 7: Razonamiento sobre programas

Tema 7: Razonamiento sobre programas

Cálculo con longitud

- Programa:

```
longitud []     = 0                  -- longitud.1
longitud (_:xs) = 1 + longitud xs    -- longitud.2
```

- Propiedad: `longitud [2,3,1] = 3`
- Demostración:

```
  longitud [2,3,1]
= 1 + longitud [2,3]                 [por longitud.2]
= 1 + (1 + longitud [3])             [por longitud.2]
= 1 + (1 + (1 + longitud []))        [por longitud.2]
= 1 + (1 + (1 + 0)                   [por longitud.1]
= 3
```

Propiedad de intercambia

- Programa:

```
intercambia :: (a,b) -> (b,a)
intercambia (x,y) = (y,x)          -- intercambia
```

- Propiedad: `intercambia (intercambia (x,y)) = (x,y)`.
- Demostración:

```
  intercambia (intercambia (x,y))
= intercambia (y,x)                    [por intercambia]
= (x,y)                                [por intercambia]
```

Propiedad de intercambia

Comprobación con QuickCheck

- Propiedad:

```
prop_intercambia :: Eq a => a -> a -> Bool
prop_intercambia x y =
    intercambia (intercambia (x,y)) == (x,y)
```

- Comprobación:

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

Inversa de listas unitarias

- Inversa de una lista:

```
inversa :: [a] -> [a]
inversa []     = []                    -- inversa.1
inversa (x:xs) = inversa xs ++ [x]     -- inversa.2
```

- Prop.: `inversa [x] = [x]`

```
  inversa [x]
= inversa (x:[])          [notación de lista]
= (inversa []) ++ [x]     [inversa.2]
= [] ++ [x]               [inversa.1]
= [x]                     [def. de ++]
```

Inversa de listas unitarias

Comprobación con QuickCheck

- Propiedad:

```
prop_inversa_unitaria :: (Eq a) => a -> Bool
prop_inversa_unitaria x =
    inversa [x] == [x]
```

- Comprobación:

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

Razonamiento ecuacional con análisis de casos

- Negación lógica:

Prelude

```
not :: Bool -> Bool
not False = True
not True  = False
```

- Prop.: `not (not x) = x`
- Demostración por casos:
  - Caso 1: `x = True`:

```
  not (not True)   = not False   [not.2]
                   = True        [not.1]
```

  - Caso 2: `x = False`:

```
  not (not False)   = not True   [not.1]
                    = False      [not.2]
```

Razonamiento ecuacional con análisis de casos

Comprobación con QuickCheck

- Propiedad:

```
prop_doble_negacion :: Bool -> Bool
prop_doble_negacion x =
  not (not x) == x
```

- Comprobación:

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

Tema 7: Razonamiento sobre programas

Esquema de inducción sobre los números naturales

Para demostrar que todos los números naturales tienen una propiedad `P` basta probar:

1. Caso base `n=0`:
   `P(0)`.
2. Caso inductivo `n=(m+1)`:
   Suponiendo `P(m)` demostrar `P(m+1)`.

En el caso inductivo, la propiedad `P(n)` se llama la hipótesis de inducción.

Ejemplo de inducción sobre los naturales: Propiedad

- `(replicate n x)` es la lista formda por `n` elementos iguales a `x`. Por ejemplo,

```
replicate 3 5  ⇝  [5,5,5]
```

Prelude

```
replicate :: Int -> a -> [a]
replicate 0     _ = []
replicate (n+1) x = x : replicate n x
```

- Prop.: `length (replicate n xs) = n`

Ejemplo de inducción sobre los naturales: Demostración

- Caso base (`n=0`):

```
length (replicate 0 xs)
= length []                          [por replicate.1]
= 0                                  [por def. length]
```

- Caso inductivo (`n=m+1`):

```
length (replicate (m+1) xs)
= length (x:(replicate m xs))        [por replicate.2]
= 1 + length (replicate m xs)        [por def. length]
= 1 + m                              [por hip. ind.]
= m + 1                              [por conmutativa de +]
```

Ejemplo de inducción sobre los naturales: Verificación

Verificación con QuickCheck:

- Especificación de la propiedad:

```
prop_length_replicate :: Int -> Int -> Bool
prop_length_replicate n xs =
    length (replicate m xs) == m
    where m = abs n
```

- Comprobación de la propiedad:

```
*Main> quickCheck prop_length_replicate
OK, passed 100 tests.
```

Tema 7: Razonamiento sobre programas

1. Razonamiento ecuacional

2. Razonamiento por inducción sobre los naturales

3. Razonamiento por inducción sobre listas
   - Esquema de inducción sobre listas
   - Asociatividad de ++
   - [] es la identidad para ++ por la derecha
   - Relación entre `length` y ++
   - Relación entre `take` y `drop`
   - La concatenación de listas vacías es vacía

4. Equivalencia de funciones

Esquema de inducción sobre listas

Para demostrar que todas las listas finitas tienen una propiedad `P` basta probar:

1. Caso base `xs=[]`:
   `P([])`.
2. Caso inductivo `xs=(y:ys)`:
   Suponiendo `P(ys)` demostrar `P(y:ys)`.

En el caso inductivo, la propiedad `P(ys)` se llama la hipótesis de inducción.

Asociatividad de ++

- Programa:

Prelude
```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys                -- ++.1
(x:xs) ++ ys = x : (xs ++ ys) -- ++.2
```

- Propiedad: `xs++(ys++zs)=(xs++ys)++zs`
- Comprobación con QuickCheck:

```
prop_asociativa_conc :: [Int] -> [Int] -> [Int] -> Bool
prop_asociativa_conc xs ys zs =
    xs++(ys++zs)==(xs++ys)++zs
```

```
Main> quickCheck prop_asociatividad_conc
OK, passed 100 tests.
```

Asociatividad de ++

- Demostración por inducción en `xs`:
  - Caso base `xs=[]`: Reduciendo el lado izquierdo

    ```
    xs++(ys++zs)
    = []++(ys++zs)          [por hipótesis]
    = ys++zs                [por ++.1]
    ```

    y reduciendo el lado derecho

    ```
    (xs++ys)++zs
    = ([]++ys)++zs          [por hipótesis]
    = ys++zs                [por ++.1]
    ```

    Luego, `xs++(ys++zs)=(xs++ys)++zs`

Asociatividad de ++

- Demostración por inducción en `xs`:
  - Caso inductivo `xs=a:as`: Suponiendo la hipótesis de inducción

    `as++(ys++zs)=(as++ys)++zs`

    hay que demostrar que

    `(a:as)++(ys++zs)=((a:as)++ys)++zs`

    `(a:as)++(ys++zs)`

    = `a:(as++(ys++zs))` [por ++.2]

    = `a:((as++ys)++zs)` [por hip. ind.]

    = `(a:(as++ys))++zs` [por ++.2]

    = `((a:as)++ys)++zs` [por ++.2]

[] es la identidad para ++ por la derecha

- Propiedad: `xs++[]=xs`
- Comprobación con QuickCheck:

```
prop_identidad_concatenacion :: [Int] -> Bool
prop_identidad_concatenacion xs = xs++[] == xs
```

```
Main> quickCheck prop_identidad_concatenacion
OK, passed 100 tests.
```

[] es la identidad para ++ por la derecha

- Demostración por inducción en `xs`:
  - Caso base `xs=[]`:

    ```
    xs++[]
    = []++[]
    = []                    [por ++.1]
    ```

  - Caso inductivo `xs=(a:as)`: Suponiendo la hipótesis de inducción

    ```
    as++[]=as
    ```

    hay que demostrar que

    ```
    (a:as)++[]=(a:as)
    (a:as)++[]
    = a:(as++[])            [por ++.2]
    = a:as                  [por hip. ind.]
    ```

Relación entre `length` y `++`

- Programas:

```
length :: [a] -> Int
length []     = 0                    -- length.1
length (x:xs) = 1 + n_length xs      -- length.2

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys                    -- ++.1
(x:xs) ++ ys = x : (xs ++ ys)        -- ++.2
```

- Propiedad: `length(xs++ys)=(length xs)+(length ys)`

Relación entre `length` y `++`

- Comprobación con QuickCheck:

```
prop_length_append :: [Int] -> [Int] -> Bool
prop_length_append xs ys =
   length(xs++ys)==(length xs)+(length ys)
```

```
Main> quickCheck prop_length_append
OK, passed 100 tests.
```

- Demostración por inducción en `xs`:
  - Caso base `xs=[]`:

    `length([]++ys)`
    = `length ys` [por `++.1`]
    = `0+(length ys)` [por aritmética]
    = `(length [])+(length ys)` [por `length.1`]

Relación entre `length` y `++`

- Demostración por inducción en `xs`:
  - Caso inductivo `xs=(a:as)`: Suponiendo la hipótesis de inducción

    ```
    length(as++ys) = (length as)+(length ys)
    ```

    hay que demostrar que

    ```
    length((a:as)++ys) = (length (a:as))+(length ys)
    ```

```
length((a:as)++ys)
= length(a:(as++ys))                    [por ++.2]
= 1 + length(as++ys)                    [por length.2]
= 1 + ((length as) + (length ys))       [por hip. ind.]
= (1 + (length as)) + (length ys)       [por aritmética]
= (length (a:as)) + (length ys)         [por length.2]
```

Relación entre take y drop

- Programas:

```
take :: Int -> [a] -> [a]
take 0 _        = []                    -- take.1
take _ []       = []                    -- take.2
take n (x:xs)   = x : take (n-1) xs     -- take.3

drop :: Int -> [a] -> [a]
drop 0 xs       = xs                    -- drop.1
drop _ []       = []                    -- drop,2
drop n (_:xs)   = drop (n-1) xs         -- drop.3

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys                       -- ++.1
(x:xs) ++ ys = x : (xs ++ ys)           -- ++.2
```

Relación entre take y drop

- Propiedad: `take n xs ++ drop n xs = xs`
- Comprobación con QuickCheck:

```
prop_take_drop :: Int -> [Int] -> Property
prop_take_drop n xs =
    n >= 0 ==> take n xs ++ drop n xs == xs
```

```
Main> quickCheck prop_take_drop
OK, passed 100 tests.
```

Relación entre `take` y `drop`

- Demostración por inducción en `n`:
  - Caso base `n=0`:

    ```
    take 0 xs ++ drop 0 xs
    = [] ++ xs                    [por take.1 y drop.1]
    = xs                          [por ++.1]
    ```

  - Caso inductivo `n=m+1`: Suponiendo la hipótesis de inducción 1

    $(\forall xs :: [a])$`take m xs ++ drop m xs = xs`

    hay que demostrar que

    $(\forall xs :: [a])$`take (m+1) xs ++ drop (m+1) xs = xs`

Relación entre take y drop

Lo demostraremos por inducción en `xs`:

- Caso base `xs=[]`:

```
take (m+1) [] ++ drop (m+1) []
= [] ++ []                                  [por take.2 y drop.2]
= []                                        [por ++.1]
```

- Caso inductivo `xs=(a:as)`: Suponiendo la hip. de inducción 2

```
take (m+1) as ++ drop (m+1) as = as
```

hay que demostrar que

```
take (m+1) (a:as) ++ drop (m+1) (a:as) = (a:as)
```

```
take (m+1) (a:as) ++ drop (m+1) (a:as)
= (a:(take m as)) ++ (drop m as)            [take.3 y drop.3]
= (a:((take m as) ++ (drop m as))           [por ++.2]
= a:as                                      [por hip. de ind. 1]
```

La concatenación de listas vacías es vacía

- Programas:

Prelude

```
null :: [a] -> Bool
null []           = True          -- null.1
null (_:_)        = False         -- null.2

(++) :: [a] -> [a] -> [a]
[]     ++ ys      = ys            -- (++).1
(x:xs) ++ ys      = x : (xs ++ ys)  -- (++).2
```

- Propiedad: `null xs = null (xs ++ xs)`.

La concatenación de listas vacías es vacía

- Demostración por inducción en `xs`:
  - Caso 1: `xs = []`: Reduciendo el lado izquierdo

```
  null xs
= null []          [por hipótesis]
= True             [por null.1]
```

y reduciendo el lado derecho

```
  null (xs ++ xs)
= null ([] ++ [])          [por hipótesis]
= null []                  [por (++).1]
= True                     [por null.1]
```

Luego, `null xs = null (xs ++ xs)`.

La concatenación de listas vacías es vacía

- Demostración por inducción en `xs`:
  - Caso `xs = (y:ys)`: Reduciendo el lado izquierdo

```
null xs
= null (y:ys)          [por hipótesis]
= False                [por null.2
```

y reduciendo el lado derecho

```
null (xs ++ xs)
= null ((y:ys) ++ (y:ys))          [por hipótesis]
= null (y:(ys ++ (y:ys))           [por (++).2]
= False                            [por null.2
```

Luego, `null xs = null (xs ++ xs)`.

Tema 7: Razonamiento sobre programas

1. Razonamiento ecuacional
2. Razonamiento por inducción sobre los naturales
3. Razonamiento por inducción sobre listas
4. Equivalencia de funciones

Equivalencia de funciones

- Programas:

```
inversa1, inversa2 :: [a] -> [a]
inversa1 []     = []
inversa1 (x:xs) = inversa1 xs ++ [x]

inversa2 xs = inversa2Aux xs []
    where inversa2Aux []     ys = ys
          inversa2Aux (x:xs) ys = inversa2Aux xs (x:ys)
```

- Propiedad: `inversa1 xs = inversa2 xs`
- Comprobación con QuickCheck:

```
prop_equiv_inversa :: [Int] -> Bool
prop_equiv_inversa xs = inversa1 xs == inversa2 xs
```

Equivalencia de funciones

- Demostración: Es consecuencia del siguiente lema:

```
inversa1 xs ++ ys = inversa2Aux xs ys
```

En efecto,

```
inversa1 xs
= inversa1 xs ++ []          [por identidad de ++]
= inversa2Aux xs ++ []       [por el lema]
= inversa2 xs                [por el inversa2.1]
```

Equivalencia de funciones

- Demostración del lema: Por inducción en `xs`:
  - Caso base `xs=[]`:

```
inversa1 [] ++ ys
= [] ++ ys                          [por inversa1.1]
= ys                                [por ++.1]
= inversa2Aux [] ++ ys              [por inversa2Aux.1]
```

  - Caso inductivo `xs=(a:as)`: La hipótesis de inducción es

$$(\forall ys :: [a])\texttt{inversa1 as ++ ys = inversa2Aux as ys}$$

Por tanto,

```
inversa1 (a:as) ++ ys
= (inversa1 as ++ [a]) ++ ys          [por inversa1.2]
= (inversa1 as) ++ ([a] ++ ys)        [por asociativa de ++]
= (inversa1 as) ++ (a:ys)             [por ley unitaria]
= (inversa2Aux as (a:ys)              [por hip. de inducción]
= (inversa2Aux (a:as) ys              [por inversa2Aux.2]
```

Bibliografía

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 13: Reasoning about programs.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 6: Programación con listas.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 8: Reasoning about programs.
6. E.P. Wentworth (1994) *Introduction to Funcional Programming*.