PROGRAMACIÓN FUNCIONAL

Derivación de programas

Derivación de programas

- Derivación de programas
  - Transformación de programas
  - Síntesis de programas
- Ejemplos
  - transformación: fact, fib, sumsqr
  - síntesis: take&drop, lines
  - ejercicios: reverse, media, sumN
- Transformación de listas por comprensión

Derivación de programas

- Considere los siguientes números

  $x_1 = 1.4142 \qquad\qquad x_2 = 7071 / 5000$

  - ¿puede verificar si los números son iguales?
- Ahora considere este número y esta fórmula

  $x_1 = 1.4142 \qquad\qquad P(y) \equiv y^2 - 1.99996164 = 0$

  - ¿puede verificar si el número satisface la propiedad expresada por la fórmula?
- Si sólo tuviéramos x_2 ó $P(y)$,
  - ¿podríamos hallar el valor de x_1?

Derivación de programas

- Verificar
  - que dos números son iguales se corresponde con ver si dos programas son equivalentes
  - que un número es solución de una fórmula se corresponde con ver si un programa satisface una propiedad
- Sin embargo, es más natural pensar en calcular números que en verificarlos...
- ¿Y con programas?

Derivación de programas

- Dados dos programas,
  - podemos probar que son equivalentes
  - podemos medir la eficiencia de ambos
- Pero, ¿qué pasa si tenemos un único programa y queremos mejorarlo?...
- Podemos:
  - inventar otro programa, y ver si son equivalentes

  o bien,
  - ¡transformar el programa en otro equivalente!

Derivación de programas

- En una situación similar, dado un programa,
  - podemos probar propiedades sobre él
    - utilizando las ecuaciones del script
    - utilizando el principio de inducción
- Pero, ¿qué pasa si tenemos la propiedad pero no el programa?... Podemos:
  - inventar un programa, y ver si cumple la propiedad

  o bien,
  - ¡calcular un programa que la cumpla!

Derivación de programas

- Derivación de programas
  - metodología constructiva de diseño de programas
  - incluye transformación y síntesis de programas
- Transformación de programas
  - dado un programa, obtener, por medios puramente sintácticos, otro equivalente
  - quizás más eficiente, pero no necesariamente
- Síntesis de programas
  - dada una propiedad, obtener, por medios puramente sintácticos, un programa que la cumple

Transformación

- Transformación de programas
  - existen varios esquemas de transformación
  - una definición y clasificación rigurosas escapan al alcance del curso
  - veremos ejemplos que ilustran algunos esquemas relevantes:
    - recursión de cola
    - tupling
    - fusión

Transformación

- **Ejemplo 1**: recursión de cola

```
fact 0 = 1
fact n = n * fact (n-1)
```

- ¿Cuánta memoria usa?
  - Lineal, porque no puede hacer ninguna cuenta hasta el final.
- ¿Cómo transformarla a una recursiva de cola?
  - Recursión de cola es equivalente a iteración

Transformación

- **Ejemplo 1**: recursión de cola

```
fact 0 = 1
fact n = n * fact (n-1)
```

- ¿Cómo transformarla a una recursiva de cola?
- Usar la eureka: $\boxed{\text{ifact r n = r * fact n}}$

Transformación

- Técnica:
  - por casos
  - en el caso inductivo, intentar aplicar la eureka en las partes inductivas
- **Caso** n=0**)** ifact r 0 (por eureka) = r * fact 0
  (por (fact.1)) = r * 1
  (por aritmética) = r
- **Caso** n>0**)** ifact r n (por eureka) = r * fact n
  (por (fact.2)) = r * (n * fact (n-1))
  (por asoc. *) = (r * n) * fact (n-1)
  (por eureka) = ifact (r * n) (n-1)

Transformación

- **Ejemplo 1**: recursión de cola

```
fact 0 = 1
fact n = n * fact (n-1)
```

- ¿Cómo transformarla a una recursiva de cola?
- Usar la eureka: $\boxed{\text{ifact r n = r * fact n}}$
- Resultado:

```
ifact r 0 = r
ifact r n = ifact (r*n) (n-1)
fact n = ifact 1 n
```

Transformación

- **Ejemplo 2**: tupling

```
fib n | n==0 || n==1 = 1
fib n = fib (n-1) + fib (n-2)
```

- Es ineficiente
  - Su costo es exponencial
  - Calcula muchas veces lo mismo
- ¿Cómo la mejoramos?

Transformación

- **Ejemplo 2**: tupling

```
fib n | n==0 || n==1 = 1
fib n = fib (n-1) + fib (n-2)
```

- Es ineficiente, pues recalcula muchas veces
- ¿Cómo la mejoramos?
- Usar la eureka: `dfib n = (fib n, fib (n+1))`

Transformación

- **Caso** n=0**)**

  dfib 0 (por eureka) = (fib 0, fib 1)

  (por (fib.1)) = (1,1)

- **Caso** n>0**)**

  dfib n (por eureka) = (fib n, fib (n+1))

  (por (fib.2)) = (fib n, fib n + fib (n-1))

  (por let) = let (a,b) = (fib (n-1), fib n)
  in (b, b+a)

  (por eureka) = let (a,b) = dfib (n-1)
  in (b, b+a)

Transformación

- **Ejemplo 2**: tupling

```
fib n | n==0 || n==1 = 1
fib n = fib (n-1) + fib (n-2)
```

- ¿Cómo la mejoramos?
- Usar la eureka: `dfib n = (fib n, fib (n+1))`
- Resultado:

```
dfib 0 = (1,1)
dfib n = let (a,b) = dfib (n-1) in (b, b+a)
fib n = fst (dfib n)
```

Transformación

- **Ejemplo 3**: fusión

```
sumsqr ns = sum (map (^2) ns)
```

- Con orden aplicativo, utiliza una cantidad lineal de memoria extra
- ¿Cómo obtenemos una definición recursiva?

Transformación

- **Ejemplo 3**: fusión

  sumsqr ns = sum (map (^2) ns)

- Con orden aplicativo, utiliza una cantidad lineal de memoria extra
- ¿Cómo obtenemos una definición recursiva?
  - Usar la definición como eureka

    $\boxed{\text{sumsqr ns = sum (map (\^{}2) ns)}}$

Transformación

- **Caso** [ ]**)** sumsqr [ ]

(por eureka)	= sum (map (^2) [ ])
(por (sqrl))	= sum [ ]
(por (sum))	= 0

- **Caso** n:ns**)** sumsqr (n:ns)

(por eureka)	= sum (map (^2) (n:ns))
(por (sqrl))	= sum (n^2 : map (^2) ns)
(por (sum))	= n^2 + sum (map (^2) ns)
(por eureka)	= n^2 + sumsqr ns

Transformación

- **Ejemplo 3**: fusión

```
sumsqr ns = sum (map (^2) ns)
```

- ¿Cómo obtenemos una definición recursiva?
  - Usar la definición como eureka
- Resultado:
  - sumsqr [ ]   = 0
  - sumsqr (n:ns) = n^2 + sumsqr ns

Síntesis

- Síntesis de programas
  - en funcional está menos desarrollada que la transformación
  - no siempre es claro como especificar las propiedades
  - veremos ejemplos de síntesis, pero un tratamiento más riguroso escapa al alcance del curso

Síntesis: ejemplo 1

- **Ejemplo**: sintetizar una definición de lines
  - para todo xs finito, lines debe cumplir que

```
type Text = String
type Line = String

lines :: Text -> [ Line ]
lines (unlines xs) = xs                              (1)

unlines [line]     = line                            (2)
unlines (line:ls) = line ++ "\n" ++ unlines ls       (3)
```

- ¿Cómo obtener una definición recursiva?

Síntesis: ejemplo 1

- Buscamos un algoritmo recursivo, entonces
- planteamos ecuaciones recursivas para lines

```
lines [ ]      = a                    (4)
lines (c:cs)   = f c (lines cs)       (5)
 -- lines = foldr f a
```

  - ¡Observar que a y f son incógnitas!
  - Ahora procedemos por análisis de casos

Síntesis: ejemplo 1

- Analizamos el caso base de lines

  a
  = (por (4))
  lines [ ]
  = (por (2))
  lines (unlines [[ ]])
  = (por (1))
  [[ ]]

  - Tenemos así el valor de a

    a = [[ ]] (6)

Síntesis: ejemplo 1

- Para el caso inductivo, precisamos 2 subcasos
  - cuando c=='\n'
  - cuando c/='\n'

  - Analizamos cada uno por separado, obteniendo las ecuaciones que faltan.

Síntesis: ejemplo 1

- Caso inductivo, cuando c=='\n'

```
f '\n' ls
 =                                       (por (1))
f '\n' (lines (unlines ls))
 =                                       (por (5))
lines ('\n' : unlines ls)
 =                                       (por listas)
lines ([] ++ "\n" ++ unlines ls)
 =                                       (por (3))
lines (unlines ([] : ls))
 =                                       (por (1))
[] : ls
```

Síntesis: ejemplo 1

- Caso inductivo, cuando c/='\n' (ls es no vacía)

```
f c (linea:ls)
  =                                   (por (1))
f c (lines (unlines (linea:ls)))
  =                                   (por (5))
lines (c : unlines (linea:ls))
  =                                   (por (3))
lines (c : (linea ++ "\n" ++ unlines ls))
  =                                   (por listas)
lines ((c:linea) ++ "\n" ++ unlines ls)
  =                                   (por (3))
lines (unlines ((c:linea) : ls))
  =                                   (por (1))
(c:linea) : ls
```

Síntesis: ejemplo 1

- Juntando ambos casos, obtenemos

```
f '\n' ls = [] : ls                                  (7)
f c (line:ls) = (c:line) : ls                        (8)
```

  - y con (4) a (8) llegamos al resultado

```
lines [ ]       = a
lines (c:cs)  = f c (lines cs)
  where   a = [ [ ] ]
          f '\n' ls        = [] : ls
          f c (line:ls)  = (c:line) : ls
```

  - ¿Qué propiedades cumple lines?

Síntesis: ejemplo 2

- **Ejemplo**: sintetizar definiciones de f & g
- Especificación: para todo n≥0, y para todo xs finito, f y g deben cumplir que

  f n xs ++ g n xs = xs (1)

  length (f n xs) = n `min` length xs (2)
- ¿Cómo obtener definiciones recursivas?
- Analizamos los distintos casos inductivos de n, y usamos propiedades de las listas

Síntesis: ejemplo 2

- **Caso base**: n=0
  - Por (2), length (f 0 xs) = 0 `min` length xs
  - 0 `min` length xs = 0, pues length xs $\geq$ 0 para toda lista finita xs (probado en clase teórica)
  - Pero length ys = 0 si y sólo si ys = [ ] (probado en clase teórica)
  - Por lo tanto,

$$\boxed{f\ 0\ xs = [\ ]} \qquad (3)$$

Síntesis: ejemplo 2

- **Caso base (cont.)**: n=0
  - Por (1), f 0 xs ++ g 0 xs = xs
  - Por (3), f 0 xs ++ g 0 xs = [ ] ++ g 0 xs
  - Pero [ ] ++ ys = xs si y sólo si ys = xs (¡probarlo!)
  - Como [ ] ++ g 0 xs = xs entonces,

g 0 xs = xs	(4)

Síntesis: ejemplo 2

- **Caso inductivo**: n=m+1
  - Debemos analizar los casos inductivos de las listas
  - **Caso base**: xs = [ ]
    - Por (1), f (m+1) [ ] ++ g (m+1) [ ] = [ ]
    - Pero xs ++ ys = [ ] si y sólo si xs = ys = [ ] (¡probarlo!)
  - Por lo tanto,

    f n [ ] = [ ] (5)

    g n [ ] = [ ] (6)
  - Falta ver que (5) satisface (2) (si no, no hay solución)

Síntesis: ejemplo 2

- **Caso inductivo (cont.)**: n=m+1
  - **Caso inductivo**: xs = (y:ys)
    - Por (2),
      length (f (m+1) (y:ys)) = (m+1) \`min\` length (y:ys)
    - Usando la definición de length y aritmética,
      (m+1) \`min\` length (y:ys) = 1 + (m \`min\` length ys)
    - Usando (2) nuevamente,
      length (f m ys) = m \`min\` length ys
      y entonces (para cualquier elección de ??)
      length (?? : f m ys) = 1 + (m \`min\` length ys)
    - Por lo tanto, eligiendo y en lugar de ??,
      length (f (m+1) (y:ys)) = length (y : f m ys)

Síntesis: ejemplo 2

- **Caso inductivo (cont.)**: n=m+1
  - **Caso inductivo (cont.)**: xs = (y:ys)
    - Pero tanto f (m+1) (y:ys) como (y : f m ys) son segmentos iniciales de (y:ys) (por (1))
    - Y como dos segmentos iniciales de igual longitud de una misma lista son iguales (¡probarlo!)
    - entonces,

$$\boxed{f\ n\ (y{:}ys) = y : f\ (n-1)\ ys} \qquad (7)$$

Síntesis: ejemplo 2

- **Caso inductivo (cont.)**: n=m+1
  - **Caso inductivo (cont.)**: xs = (y:ys)
    - Reemplazando (7) en (1) tenemos que
      (y: f m ys) ++ g (m+1) (y:ys) = y:ys
    - Usando (++.2) y el hecho de que las listas son algebraicas, vemos que
      f m ys ++ g (m+1) (y:ys) = ys
    - Pero, por (1) sabemos que
      f m ys ++ g m ys = ys
    - entonces, como las listas son algebraicas,

$$\boxed{g\ n\ (y{:}ys) = g\ (n-1)\ ys} \qquad (8)$$

Síntesis: ejemplo 2

- Combinando los resultados (3), (4), (5), (6), (7) y (8), obtenemos las definiciones buscadas

```
f 0 xs = [ ]                          (3)
f n [ ] = [ ]                         (5)
f n (y:ys) = y : f (n-1) ys           (7)

g 0 xs = xs                           (4)
g n [ ] = [ ]                         (6)
g n (y:ys) = g (n-1) ys               (8)
```

- ¿Qué sabemos de estas funciones? ¿Qué propiedades cumplen?

Transformación

- **Ejercicio 1**: recursión de cola

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

- Es ineficiente
  - (++) es lineal en su primer argumento
  - entonces queda cuadrática
- ¿Cómo transformarla a una recursiva de cola?

Transformación

- **Ejercicio 1**: recursión de cola

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

- ¿Cómo transformarla a una recursiva de cola?
- Usar la eureka: $\boxed{\texttt{irev rs xs = reverse xs ++ rs}}$

Transformación

- **Caso** [ ]**)** irev rs [ ]

  (por eureka) = reverse [ ] ++ rs

  (por (reverse.1)) = [ ] ++ rs

  (por (++.1)) = rs

- **Caso** x:xs**)** irev rs (x:xs)

  (por eureka) = reverse (x:xs) ++ rs

  (por (reverse.2)) = (reverse xs ++ [x]) ++ rs

  (por asoc. (++)) = reverse xs ++ ([x] ++ rs)

  (por (++.1y2)) = reverse xs ++ (x:rs)

  (por eureka) = irev (x:rs) xs

Transformación

- **Ejercicio 1**: recursión de cola

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

- ¿Cómo transformarla a una recursiva de cola?
- Usar la eureka: `irev rs xs = reverse xs ++ rs`
- Resultado:

```
irev rs [ ] = rs
irev rs (x:xs) = irev (x:rs) xs

fastrev xs = irev [ ] xs
```

Transformación

- **Ejercicio 2**: tupling

```
media xs = sum xs / length xs
```

- Es ineficiente, pues recorre dos veces la lista
- ¿Cómo la mejoramos?

Transformación

- **Ejercicio 2**: tupling

    media xs = sum xs / length xs

- Es ineficiente, pues recorre dos veces la lista
- ¿Cómo la mejoramos?
- Usar la eureka: $\boxed{\text{st xs = (sum xs, length xs)}}$

Transformación

- **Caso** [ ]**)** st [ ]

  (por eureka) = (sum [ ], length [ ])

  (por (sum) y (length)) = (0, 0)

- **Caso** x:xs**)** st (x:xs)

  (por eureka) = (sum (x:xs), length (x:xs))

  (por (sum) y (length)) = (x + sum xs, 1 + length xs)

  (por let) = let (a,b) = (sum xs, length xs)
  in (x + a, 1 + b)

  (por eureka) = let (a,b) = st xs
  in (x + a, 1 + b)

Transformación

- **Ejercicio 2**: tupling

```
media xs = sum xs / length xs
```

- Es ineficiente, pues recorre dos veces la lista
- ¿Cómo la mejoramos?
- Usar la eureka: $\boxed{\texttt{st xs = (sum xs, length xs)}}$
- Resultado:

```
st [ ] = (0,0)
st (x:xs) = let (a,b) = st xs in (x+a, 1+b)
media xs = let (a,b) = st xs in a / b
```

Transformación

- **Ejercicio 3**: fusión

```
sumN n = sum (take n nats)
nats = iterate (+1) 0
```

- ¿Cómo obtener una versión sin partes infinitas?

Transformación

- **Ejercicio 3**: fusión

```
sumN n = sum (take n nats)
nats = iterate (+1) 0
```

- ¿Cómo obtener una versión sin partes infinitas?
- Usar la eureka:

$$\boxed{\text{sumMN i n = sum (take n (iterate (+1) i))}}$$

Transformación

- **Caso** n=0**)** sumMN i 0

(por eureka)	= sum (take 0 (iterate (+1) i))
(por (take.1))	= sum [ ]
(por (sum))	= 0

- **Caso** n>0**)** sumMN i n

(por eureka)	= sum (take n (iterate (+1) i))
(por (iterate))	= sum (take n (i : iterate (+1) (i+1)))
(por (take.3))	= sum (i : take (n-1) (iterate (+1) (i+1)))
(por (sum))	= i + sum (take (n-1) (iterate (+1) (i+1)))
(por eureka)	= i + sumMN (i+1) (n-1)

Transformación

- **Ejercicio 3**: fusión

```
sumN n = sum (take n nats)
nats = iterate (+1) 0
```

- ¿Cómo obtener una versión sin partes infinitas?
- Usar la eureka:

```
sumMN i n = sum (take n (iterate (+1) i))
```

- Resultado:

```
sumN n = sumMN n 0
sumMN i 0 = 0
sumMN i n = i + sumMN (i+1) (n-1)
```

Derivación con listas

- Las listas por comprensión se definieron de manera informal.
- ¿Podrá darse una derivación que transforme una lista por comprensión en una expresión que no utilice comprensiones?
- ¡Hay que utilizar funciones de alto orden!
  - map
  - filter
  - concat

Derivación con listas

- Reglas de derivación

  (1) $[\ f\ x \mid x \leftarrow xs\ ] = map\ f\ xs$

  (2) $[\ e \mid x \leftarrow xs,\ p\ x,\ q_3, \ldots, q_k]$
  $= [\ e \mid x \leftarrow filter\ p\ xs,\ q_3, \ldots, q_k\ ]$

  (3) $[\ e \mid x \leftarrow xs,\ y \leftarrow ys,\ q_3, \ldots, q_k]$
  $= concat\ [\ [\ e \mid y \leftarrow ys,\ q_3, \ldots, q_k\ ] \mid x \leftarrow xs\ ]$

  - Aplicar estas reglas hasta que no se pueda más
  - Puede ser necesario reescribir alguna expresión

Derivación con listas

- Ejemplo
  - [x^2 | x <- [1..6], even x]

    = -- (por (2))

    [x^2 | x <-filter even [1..6]]

    = -- (por secc.de operadores)

    [(^2) x | x <-filter even [1..6]]

    = -- (por (1))

    map (^2) (filter even [1..6])

Derivación con listas

- [(i,j) | i <- [1..3], j <-['a','b']]

```
  =                                                    (por (3))
    concat [ [ (i,j) | j <-[ 'a','b' ] ] | i <-[1..3] ]
  =                                                    (por pair)
    concat [ [ (pair i) j | j <-[ 'a','b' ] ] | i <-[1..3] ]
  =                                                    (por (1))
    concat [ map (pair i) [ 'a','b' ] | i <-[1..3] ]
  =                                                    (por let)
    let f i = map (pair i) [ 'a','b' ]
     in concat [ f i | i <-[1..3] ]
  =                                                    (por (1))
    let f i = map (pair i) [ 'a','b' ]
     in concat (map f [1..3])
```

Derivación con listas

- [(i,j) | i <- [1..3], even i, j <-[i+1..4], odd j]

```
(por (2)) = [ (i,j) | i <-filter even [1..3], j <-[ i+1..4 ], odd j ]
(por (3)) = concat [ [ (pair i) j | j <-[ i+1..4 ], odd j ]
                     | i <-filter even [1..3] ]
(por (2)) = concat [ [ (pair i) j | j <-filter odd [ i+1..4 ] ]
                     | i <-filter even [1..3] ]
(por (1)) = concat [ map (pair i) (filter odd [ i+1..4 ])
                     | i <-filter even [1..3] ]
(por (1)) = let f i = map (pair i) (filter odd [ i+1..4 ])
            in concat (map f (filter even [1..3]))
```

Resumen

- Es posible 'calcular' programas a partir de especificaciones, de manera sintáctica
- Es posible 'mejorar' un programa transformándolo de manera sintáctica
- La derivación
  - guía el proceso de construcción de programas
  - nos ahorra las demostraciones de corrección