

A decorative vertical bar on the left side of the slide, composed of two parallel gray lines. On the right side, three large, light gray triangles are arranged vertically, pointing downwards.

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
 - ◆ ejercicios: reverse, media, sumN
- ◆ Transformación de listas por comprensión

Derivación de programas

- ◆ Considere los siguientes números

$$x_1 = 1.4142$$

$$x_2 = 7071 / 5000$$

- ◆ ¿puede verificar si los números son iguales?

- ◆ Ahora considere este número y esta fórmula

$$x_1 = 1.4142$$

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

- ◆ ¿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)

- ◆ ¿Cómo transformarla a una recursiva de cola?

- ◆ Usar la eureka: `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

◆ 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 2: tupling**

$\text{fib } n \mid n==0 \parallel n==1 = 1$

$\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$

- ◆ Es ineficiente, pues recalcula muchas veces

- ◆ ¿Cómo la mejoramos?

- ◆ Usar la eureka: $\text{dfib } n = (\text{fib } n, \text{fib } (n+1))$

- ◆ Resultado:

$\text{dfib } 0 = (1, 1)$

$\text{dfib } n = \text{let } (a, b) = \text{dfib } (n-1) \text{ in } (b, b+a)$

$\text{fib } n = \text{fst } (\text{dfib } n)$

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 3: fusión**

`sumsqr ns = sum (sqrl ns)`

`sum = foldr (+) 0`

`sqrl = map (^2)`

- ◆ ¿Cómo obtenemos una definición recursiva de `sumsqr`? Por análisis de casos

- ◆ **Resultado:**

`sumsqr [] = 0`

`sumsqr (n:ns) = n^2 + sumsqr ns`

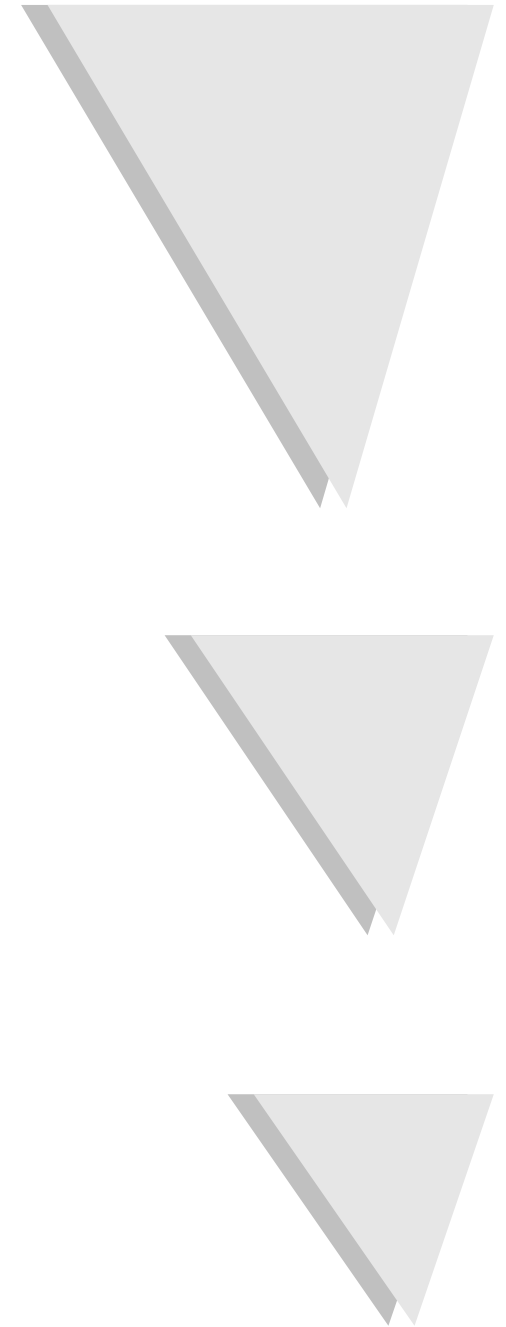
Transformación

◆ Caso []

(por eureka)	sumsqr []
(por (sqr))	= sum (sqr [])
(por (sum))	= 0

◆ Caso n:ns

	sumsqr (n:ns)
(por eureka)	= sum (sqr (n:ns))
(por (sqr))	= sum (n^2 : sqr ns)
(por (sum))	= n^2 + sum (sqr ns)
(por eureka)	= 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 un ejemplo de síntesis, pero un tratamiento más riguroso escapa al alcance del curso

Síntesis (ejemplo)

- ◆ **Ejemplo:** sintetizar definiciones de take&drop
- ◆ Especificación: para todo $n \geq 0$, y para todo xs finito, take y drop deben cumplir que
$$\text{take } n \text{ } xs ++ \text{drop } n \text{ } xs = xs \quad (1)$$
$$\text{length } (\text{take } n \text{ } xs) = n \text{ `min` length } xs \quad (2)$$
- ◆ ¿Cómo obtener definiciones recursivas?
- ◆ Analizamos los distintos casos inductivos de n , y usamos propiedades de las listas

Síntesis (ejemplo)

❖ Caso base: $n=0$

- ❖ Por (2), $\text{length}(\text{take } 0 \text{ xs}) = 0 \text{ `min` length xs}$
- ❖ $0 \text{ `min` length xs} = 0$, pues $\text{length xs} \geq 0$ para toda lista finita xs (probado en clase teórica)
- ❖ Pero $\text{length ys} = 0$ si y sólo si $\text{ys} = []$ (probado en clase teórica)
- ❖ Por lo tanto,

$\text{take } 0 \text{ xs} = []$

(3)

Síntesis (ejemplo)

♦ Caso base (cont.): $n=0$

- ♦ Por (1), $\text{take } 0 \text{ xs} ++ \text{drop } 0 \text{ xs} = \text{xs}$
- ♦ Por (3), $\text{take } 0 \text{ xs} ++ \text{drop } 0 \text{ xs} = [] ++ \text{drop } 0 \text{ xs}$
- ♦ Pero $[] ++ \text{ys} = \text{xs}$ si y sólo si $\text{ys} = \text{xs}$ (¡probarlo!)
- ♦ Como $[] ++ \text{drop } 0 \text{ xs} = \text{xs}$ entonces,

$$\text{drop } 0 \text{ xs} = \text{xs}$$

(4)

Síntesis (ejemplo)

- ◆ **Caso inductivo:** $n=m+1$

- ◆ Debemos analizar los casos inductivos de las listas

- ◆ **Caso base:** $xs = []$

- ◆ Por (1), $take\ (m+1)\ [] ++ drop\ (m+1)\ [] = []$

- ◆ Pero $xs ++ ys = []$ si y sólo si $xs = ys = []$
(¡probarlo!)

- ◆ Por lo tanto,

$take\ n\ [] = []$

(5)

$drop\ n\ [] = []$

(6)

Síntesis (ejemplo)

- ◆ **Caso inductivo (cont.): $n=m+1$**

- ◆ **Caso inductivo: $xs = (y:ys)$**

- ◆ Por (2),

$$\text{length}(\text{take}(m+1)(y:ys)) = (m+1) \text{ `min` } \text{length}(y:ys)$$

- ◆ Usando la definición de `length` y aritmética,

$$(m+1) \text{ `min` } \text{length}(y:ys) = 1 + (m \text{ `min` } \text{length } ys)$$

- ◆ Usando (2) nuevamente,

$$\text{length}(\text{take } m \text{ } ys) = m \text{ `min` } \text{length } ys$$

y entonces

$$\text{length}(y : \text{take } m \text{ } ys) = 1 + (m \text{ `min` } \text{length } ys)$$

- ◆ Por lo tanto,

$$\text{length}(\text{take}(m+1)(y:ys)) = \text{length}(y : \text{take } m \text{ } ys)$$

Síntesis (ejemplo)

♦ **Caso inductivo (cont.): $n=m+1$**

♦ **Caso inductivo (cont.): $xs = (y:ys)$**

- ♦ Pero tanto $\text{take } (m+1) (y:ys)$ como $(y : \text{take } m \text{ } ys)$ son segmentos iniciales de $(y:ys)$ (por (1))
- ♦ Y como dos segmentos iniciales de igual longitud de una misma lista son iguales (¡probarlo!)
- ♦ entonces,

$$\text{take } n (y:ys) = y : \text{take } (n-1) \text{ } ys$$

(7)

Síntesis (ejemplo)

◆ **Caso inductivo (cont.): $n=m+1$**

◆ **Caso inductivo (cont.): $xs = (y:ys)$**

- ◆ Reemplazando (7) en (1) tenemos que
 $(y: \text{take } m \text{ } ys) ++ \text{drop } (m+1) (y:ys) = y:ys$
- ◆ Usando $(++ .2)$ y el hecho de que las listas son algebraicas, vemos que
 $\text{take } m \text{ } ys ++ \text{drop } (m+1) (y:ys) = ys$
- ◆ Pero, por (1) sabemos que
 $\text{take } m \text{ } ys ++ \text{drop } m \text{ } ys = ys$
- ◆ entonces, como las listas son algebraicas,

$$\text{drop } n (y:ys) = \text{drop } (n-1) \text{ } ys$$

(8)

Síntesis (ejemplo)

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

$\text{take } 0 \text{ xs} = []$ (3)

$\text{take } n [] = []$ (5)

$\text{take } n (y:ys) = y : \text{take } (n-1) \text{ ys}$ (7)

$\text{drop } 0 \text{ xs} = \text{xs}$ (4)

$\text{drop } n [] = []$ (6)

$\text{drop } n (y:ys) = \text{drop } (n-1) \text{ ys}$ (8)

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

Transformación

- ◆ **Ejercicio 1:** recursión de cola

$\text{reverse } [] = []$

$\text{reverse } (x:xs) = \text{reverse } xs ++ [x]$

- ◆ ¿Cómo transformarla a una recursiva de cola?

- ◆ Usar la eureka: $\text{irev } rs \ xs = \text{reverse } xs ++ rs$

- ◆ Resultado:

$\text{irev } rs \ [] = rs$

$\text{irev } rs \ (x:xs) = \text{irev } (x:rs) \ xs$

$\text{fastrev } xs = \text{irev } [] \ xs$

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

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

Transformación

- ◆ **Ejercicio 2: tupling**

$\text{media xs} = \text{sum xs} / \text{length xs}$

$\text{sum} = \text{foldr } (+) \ 0; \text{length} = \text{foldr } (\backslash x \ n \rightarrow 1+n) \ 0$

- ◆ Es ineficiente, pues recorre dos veces la lista

- ◆ ¿Cómo la mejoramos?

- ◆ Usar la eureka: $\text{st xs} = (\text{sum xs}, \text{length xs})$

- ◆ Resultado:

$\text{st } [] = (0,0)$

$\text{st } (x:xs) = \text{let } (a,b) = \text{st xs in } (x+a, 1+b)$

$\text{media xs} = \text{let } (a,b) = \text{st xs in } a / b$

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 3: fusión

$\text{sumN } n = \text{sum } (\text{take } n \text{ nats})$

$\text{nats} = \text{iterate } (+1) 0$

◆ ¿Cómo obtenemos una versión de sumN que no tenga partes infinitas? Usar la eureka:

$$\text{sumN}' n i = \text{sum } (\text{take } n (\text{iterate } (+1) i))$$

◆ Resultado:

$\text{sumN } n = \text{sumN}' n 0$

$\text{sumN}' 0 i = i$

$\text{sumN}' n i = i + \text{sumN}' (n-1) (i+1)$

Transformación

➤ Caso $n=0$) $\text{sumN}'\ 0\ i$

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

(por (take.1)) = sum []

(por (sum)) = 0

➤ Caso $n>0$) $\text{sumN}'\ n\ i$

(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 + $\text{sumN}'\ (n-1)\ (i+1)$

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

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

$$(2) [e \mid x \leftarrow xs, p\ x, q_3, \dots, q_k] \\ = [e \mid x \leftarrow \text{filter } p\ xs, q_3, \dots, q_k]$$

$$(3) [e \mid x \leftarrow xs, y \leftarrow ys, q_3, \dots, q_k] \\ = \text{concat } [[e \mid y \leftarrow ys, q_3, \dots, q_k] \mid x \leftarrow xs]$$

- ◆ 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']]`

=

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

=

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

=

`concat [map (pair i) ['a','b'] | i <- [1..3]]`

=

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

=

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

(por (3))

(por pair)

(por (1))

(por let)

(por (1))

Derivación con listas

◆ $[(i,j) \mid i \leftarrow [1..3], \text{even } i, j \leftarrow [i+1..4], \text{odd } j]$

(por (2)) = $[(i,j) \mid i \leftarrow \text{filter even } [1..3], j \leftarrow [i+1..4], \text{odd } j]$

(por (3)) = $\text{concat } [[(\text{pair } i) j \mid j \leftarrow [i+1..4], \text{odd } j] \mid i \leftarrow \text{filter even } [1..3]]$

(por (2)) = $\text{concat } [[(\text{pair } i) j \mid j \leftarrow \text{filter odd } [i+1..4]] \mid i \leftarrow \text{filter even } [1..3]]$

(por (1)) = $\text{concat } [\text{map } (\text{pair } i) (\text{filter odd } [i+1..4]) \mid i \leftarrow \text{filter even } [1..3]]$

(por (1)) = $\text{let } f \ i = \text{map } (\text{pair } i) (\text{filter odd } [i+1..4])$
in $\text{concat } (\text{map } f (\text{filter even } [1..3]))$

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