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

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

  - ¿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)

- ¿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

- 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

  ```
  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))`
- Resultado:

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

(por eureka)	= sum (sqrl [ ])
(por (sqrl))	= sum [ ]
(por (sum))	= 0

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

(por eureka)	= sum (sqrl (n:ns))
(por (sqrl))	= sum (n^2 : sqrl ns)
(por (sum))	= n^2 + sum (sqrl 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≥0, y para todo xs finito, take y drop deben cumplir que

  take n xs ++ drop n xs = xs (1)

  length (take 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)

- **Caso base**: n=0
  - Por (2), length (take 0 xs) = 0 `min` length xs
  - 0 `min` length xs = 0, pues length xs ≥ 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,

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

Síntesis (ejemplo)

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

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

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),
      length (take (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 (take m ys) = m `min` length ys
      y entonces
      length (y : take m ys) = 1 + (m `min` length ys)
    - Por lo tanto,
      length (take (m+1) (y:ys)) = length (y : take m ys)

Síntesis (ejemplo)

- **Caso inductivo (cont.)**: n=m+1
  - **Caso inductivo (cont.)**: xs = (y:ys)
    - Pero tanto take (m+1) (y:ys) como (y : take 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,

take n (y:ys) = y : take (n-1) ys	(7)

Síntesis (ejemplo)

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

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

Síntesis (ejemplo)

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

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

drop 0 xs = xs                               (4)
drop n [ ] = [ ]                             (6)
drop n (y:ys) = drop (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]
```

- ¿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

- **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 2**: tupling

```
media xs = sum xs / length xs
sum = foldr (+) 0; length = foldr (\x n -> 1+n) 0
```

- Es ineficiente, pues recorre dos veces la lista
- ¿Cómo la mejoramos?
- Usar la eureka: `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

- **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

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

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

```
sumN' n i = sum (take n (iterate (+1) i))
```

- Resultado:

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

Transformación

- **Caso** n=0**)** sumN' 0 i

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

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

(por (sum)) = 0

- **Caso** n>0**)** 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 + 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) $[\ f\ x \mid x \leftarrow xs\ ] = map\ f\ xs$

  (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