Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2023

Departamento de Computación - FCEyN - UBA

Polimorfismo, Currificación y Recursión sobre enteros

Polimorfismo

- Se llama polimorfismo a una función que puede aplicarse a distintos tipos de datos (sin redefinirla).
- se usa cuando el comportamiento de la función no depende paramétricamente del tipo de sus argumentos
- lo vimos en el lenguaje de especificación con las funciones genéricas.
- En Haskell los polimorfismos se escriben usando variables de tipo y conviven con el tipado fuerte.
- Ejemplo de una función polimórfica: la función identidad.

Variables de tipos

¿Qué tipo tienen las siguientes funciones?

```
identidad x = x

primero x y = x

segundo x y = y

constante5 x y z = 5
```

Variables de tipo

- Son parámetros que se escriben en la signatura usando variables minúsculas
- En lugar de valores, denotan tipos
- Cuando se invoca la función se usa como argumento el tipo del valor

Variables de tipo (cont.)

Funciones con variables de tipo

```
identidad :: t → t
identidad x = x

primero :: tx → ty → tx
primero x y = x

segundo :: tx → ty → ty
segundo x y = y

constante5 :: tx → ty → tz → Int
constante5 x y z = 5

mismoTipo :: t → t → Bool
mismoTipo x y = True
```

Si dos argumentos deben tener el mismo tipo, se debe usar la misma variable de tipo

- Luego, `primero True 5 :: Bool`, pero `mismoTipo 1 True 0` no tipa

Clases de tipos

¿Qué tipo tienen las siguientes funciones?

```
triple x = 3*x

maximo x y | x >= y = x
           | otherwise = y

distintos x y = x /= y
```

Clases de tipos

- Conjunto de tipos a los que se le pueden aplicar ciertas funciones
- Un tipo puede pertenecer a distintas clases

  Los `Float` son números (`Num`), con orden (`Ord`), de punto flotante (`Floating`), etc.

En este curso

- No vamos a evaluar el uso de clases de tipos, pero ...
- ...saber la mecánica permite comprender los mensajes del compilador de Haskell (GHCi)

Clases de tipos (cont)

Clase de tipos

- Conjunto de tipos de datos a los que se les puede aplicar un conjunto de funciones

Algunas clases:

1. `Integral` := ({ `Int`, `Integer`,∘.. },{ `mod`, `div`,∘.. })
2. `Fractional` := ({ `Float`, `Double`,∘.. },{ `(/)`,∘.. })
3. `Floating` := ({ `Float`, `Double`,∘.. },{ `sqrt`, `sin`, `cos`, `tan`,∘.. })
4. `Num` := ({ `Int`, `Integer`, `Float`, `Double`,∘.. },{ `(+)`, `(*)`, `abs`,∘.. })
5. `Ord` := ({`Bool`, `Int`, `Integer`, `Float`, `Double`,∘.. },{ `(<=)`, `compare` })
6. `Eq` := ({ `Bool`, `Int`, `Integer`, `Float`, `Double`,∘.. },{ `(==)`, `(/=)` })

Clases de tipos (cont)

Las clases de tipos se describen como restricciones sobre variables de tipos

```
triple :: (Num t) => t -> t
triple x = 3*x

maximo :: (Ord t) => t -> t -> t
maximo x y | x >= y = x
           | otherwise = y

distintos :: (Eq t) => t -> t -> Bool
distintos x y = x /= y

-- Cantidad de raices de la ecuación: ax^2 + bx + c
cantidadDeSoluciones :: (Num t, Ord t) => t -> t -> t-> Int
cantidadDeSoluciones a b c | discriminante > 0 = 2
                           | discriminante == 0 = 1
                           | otherwise = 0
                           where discriminante = b^2 - 4*a*c

pepe :: (Floating t, Eq t, Num u, Eq u) => t -> t -> u -> Bool
pepe x y z = sqrt (x + y) == x && 3*z == 0
```

`(Floating t, Eq t, Num u, Eq u) =>`∘.. significa que:

- la variable `t` tiene que ser de un tipo que pertenezca a `Floating` y `Eq`
- la variable `u` tiene que ser de un tipo que pertenezca a `Num` y `Eq`

Ejercitación conjunta

Averiguar el tipo asignado por Haskell a las siguientes funciones

```
f1 x y z = x**y + z <= x+y**z

f2 x y = (sqrt x) / (sqrt y)

f3 x y = div (sqrt x) (sqrt y)

f4 x y z | x == y = z
         | x ** y == y = x
         | otherwise = y

f5 x y z | x == y = z
         | x ** y == y = z
         | otherwise = z
```

¿Qué error ocurre cuándo ejecutamos `f4 5 5 True`? ¿Tiene sentido?

¿Y si ejecutamos `f5 5 5 True`? ¿Qué cambió?

Nueva familia de tipos: Tuplas

Tuplas

- Dados tipos $A_1, \ldots, A_k$, el tipo *k*-upla $(A_1, \ldots, A_k)$ es el conjunto de las *k*-uplas $(v_1, \ldots, v_k)$ donde v_i es de tipo A_i

```
(1, 2)           :: (Int, Int)
(1.1, 3.2, 5.0)  :: (Float, Float, Float)
(True, (1, 2)) :: (Bool, (Int, Int))
(True, 1, 2)   :: (Bool, Int, Int)
```

- En Haskell hay infinitos tipos de tuplas

Funciones de acceso a los valores de un par en `Prelude`

- `fst :: (a, b) →a` Ejemplo: `fst (1 +4, 2)` ⇝ 5
- `snd :: (a, b) →b` Ejemplo: `snd (1, (2, 3))` ⇝ `(2, 3)`

Ejemplo: suma de vectores en $\mathbb{R}^2$

```
suma :: (Float, Float) → (Float, Float) → (Float, Float)
suma v w = ((fst v) + (fst w), (snd v) + (snd w))
```

Podemos usar *pattern matching* para acceder a los valores de una tupla

```
suma (vx, vy) (wx, wy) = (vx + wx, vy + wy)
```


Pattern matching sobre tuplas

Podemos usar *pattern matching* sobre constructores de tuplas y números

```
esOrigen :: (Float,Float) → Bool
esOrigen (0, 0) = True
esOrigen (_, _) = False

angulo0 :: (Float, Float) → Bool
angulo0 (_, 0) = True
angulo0 (_, _) = False

{-
No podemos usar dos veces la misma variable
angulo45 :: (Float, Float) → Bool
angulo45 (x,x) = True
angulo45 (_,_) = False
-}
angulo45 :: (Float, Float) → Bool
angulo45 (x,y) = x == y

patternMatching :: (Float, (Bool, Int), (Bool, (Int, Float))) → (Float, (Int, Float))
patternMatching (f1, (True, _), (_, (0, f2))) = (f1, (1, f2))
patternMatching (_ , _          , (_, (_, f ))) = (f,  (0, f))
```


Parámetros vs. tuplas

¿Conviene tener dos parámetros escalares o un parámetro dupla?

```
suma :: (Float, Float) → (Float, Float) → (Float, Float)
suma (vx, vy) (wx, wy) = (vx + wx, vy + wy)

-- normaVectorial2 x y es la norma de (x,y)
normaVectorial2 ::  Float → Float → Float
normaVectorial2 x y = sqrt (x^2 + y^2)

-- normaVectorial1 (x,y) es la norma de (x,y)
normaVectorial1 ::  (Float, Float) → Float
normaVectorial1 (x,y) = sqrt (x^2 + y^2)

norma1Suma :: (Float, Float) → (Float, Float) → Float
norma1Suma v1 v2 = normaVectorial1 (suma v1 v2)

norma2Suma :: (Float, Float) → (Float, Float) → Float
norma2Suma v1 v2 = normaVectorial2 (fst s) (snd s)
    where s = suma v1 v2
```


Currificación

- Diferencia entre `promedio1` y `promedio2`
  - `promedio1 :: (Float, Float) -> Float`
    `promedio1 (x,y) = (x+y)/2`
  - `promedio2 :: Float -> Float -> Float`
    `promedio2 x y = (x+y)/2`
- solo cambia el tipo de datos de la función
  - `promedio1` recibe un solo parámetro (una dupla)
  - `promedio2` recibe dos `Float` separados por un espacio
  - para declararla, separamos los tipos de los parámetros con una flecha
  - tiene motivos teóricos y prácticos (que no veremos ahora)
- la notación se llama currificación en honor al matemático Haskell B. Curry
- para nosotros, alcanza con ver que evita el uso de varios signos de puntuación (comas y paréntesis)
  - `promedio1 (promedio1 (2, 3), promedio1 (1, 2))`
  - `promedio2 (promedio2 2 3) (promedio2 1 2)`

Funciones binarias: notación prefija vs. infija

Funciones binarias

- Notación prefija: función antes de los argumentos (e.g., `suma x y`)
- Notación infija: función entre argumentos (e.g. `x +y`, `5 * 3`, etc)
- La notación infija se permite para funciones cuyos nombres son operadores
- El nombre real de una función definido por un operador • es (•)
- Se puede usar el nombre real con notación prefija, e.g. `(+) 2 3`
- Haskell permite definir nuevas funciones con símbolos, e.g., `(*+)` (no hacerlo!)
- Una función binaria `f` puede ser usada de forma infija escribiendo `‘f‘`

Ejemplos:

```
(≥) :: Ord a ⇒ a → a → Bool
(≥) 5 3 —evalua a True
(==) :: Eq a ⇒ a → a → Bool
(==) 3 4 —evalua a False
(^) :: (Num a, Int b) ⇒ a → b → a
(^) 2 5 —evalua 32.0
mod :: (Integral a) ⇒ a → a → a
5 ‘mod‘ 3 —evalua 2
div :: (Integral a) ⇒ a → a → a
5 ‘div‘ 3 —evalua 1
```

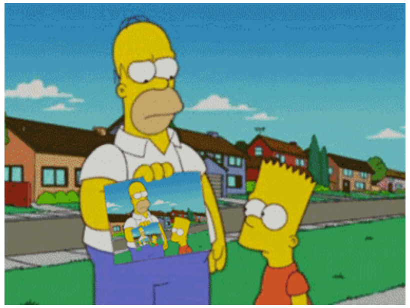

Recursión

- Hasta ahora, especificamos funciones que consistían en "expresiones sencillas".
- ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

$$n! = \prod_{k=1}^{n} k \qquad\qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

```
factorial :: Int → Int
factorial n
  | n == 0 = 1
  | n > 0 = n * factorial (n−1)
```



Recursión y reducción

¿Podemos definirla usando `otherwise`?

```
factorial :: Int → Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n−1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int → Int
factorial 0 = 1
factorial n = n * factorial (n−1)
```

¿Cómo reduce la expresión `factorial 3`?

```
factorial 3 ⇝ 3 * factorial 2 ⇝ 3 * 2 * factorial 1 ⇝
⇝ 6 * factorial 1 ⇝ 6 * 1 * factorial 0 ⇝ 6 * factorial 0 ⇝
⇝ 6 * 1 ⇝ 6
```


Asegurarse de llegar a un caso base

Veamos este programa recursivo para determinar si un entero positivo es par:

```
esPar :: Int → Bool
esPar n | n==0 = True
| otherwise = esPar (n−2)
```

¿Qué problema tiene esta función?

¿Cómo se arregla?

```
esPar :: Int → Bool
esPar n | n==0 = True
| n==1 = False
| otherwise = esPar (n−2)
```

```
esPar :: Int → Bool
esPar n | n==0 = True
| otherwise = not (esPar (n−1))
```

¿Cómo pensar recursivamente?

- Si queremos definir una función recursiva, por ejemplo `factorial`,
  - en el paso recursivo, suponiendo que tenemos el resultado para el caso anterior, ¿qué falta para poder obtener el resultado que quiero? En este caso, suponemos ya calculado `factorial (n-1)` y lo combinamos multiplicándolo por `n` para lograr obtener `factorial n`.
  - además, identificamos el o los casos base. En el ejemplo de `factorial`, definimos como casos base la función sobre 0: `factorial n | n == 0 = 1`
- Propiedades de una definición recursiva:
  - las llamadas recursivas tienen que *"acercarse"* a un caso base.
  - tiene que tener uno o más casos base que dependerán del tipo de llamado recursivo. Un caso base, es aquella expresión que no tiene paso recursivo.

¿Cómo pensar recursivamente?

- Casos bases: identificar el o los casos bases.
- Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

Otro Ejemplo:

```
sumaLosPrimerosNImpares :: Integer → Integer
sumaLosPrimerosNImpares n
 | n == 1 = 1
 | n > 1 =∘ .. sumaLosPrimerosNImpares (n−1) ∘ ..
```

- Verificar que (n==1) es el caso base, está bien definido y no hay otros.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos!

Con el paso anterior resuelto: ¿Qué falta para que el nuevo paso esté resuelto?

```
 | n > 1 = n_esimoImpar + sumaLosPrimerosNImpares (n−1)
```

Cambiamos el problema: ahora sólo falta definir `n_esimoImpar`.

```
 | n > 1 = n_esimoImpar + sumaLosPrimerosNImpares (n−1)
 where n_esimoImpar = 2*n − 1
```


Inducción vs. Recursión

- Probar por inducción $P(n) : \sum_{i=1}^{n}(2i-1) = n^2$
- Vale para $n = 1 : \sum_{i=1}^{1}(2i-1) = 1^2$
- Supongo que vale $P(n)$, quiero probar $P(n+1)$
- ¿Qué relación hay entre $\sum_{i=1}^{n}(2i-1)$ y $\sum_{i=1}^{n+1}(2i-1)$?

$$\sum_{i=1}^{n+1}(2i-1) = \left(\sum_{i=1}^{n}(2i-1)\right) + 2n+1$$

- Uso la Hipótesis Inductiva $P(n)$:

$$\sum_{i=1}^{n+1}(2i-1) = n^2 + 2n + 1 = (n+1)^2$$

- ¡¿Pero cómo?! ¡¿Estoy usando lo que quiero probar?!
- Ah, claro... vale $P(1)$ y $P(n) => P(n+1)$, entonces ¡vale para todo n!

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base en Haskell: `f 1 =1`
- Supongo que ya sé calcular $f(n-1)$, quiero calcular $f(n)$
- ¿Qué relación hay entre $\sum_{i=1}^{n-1}(2i-1)$ y $\sum_{i=1}^{n}(2i-1)$?

$$\sum_{i=1}^{n}(2i-1) = \left(\sum_{i=1}^{n-1}(2i-1)\right) + 2n - 1$$

- Uso la función que sé calcular: $f(n) = f(n-1) + 2n - 1$

  En Haskell: `f n =f (n−1) +2*n −1`

- ¡¿Pero cómo?! ¡¿Estoy usando la función que quiero definir?!
- Ah, claro... está definido $f(1)$ y con $f(n-1)$ sé obtener $f(n)$, entonces ¡puedo calcular f para todo n!

Generalización de funciones

¿Una fácil?.. o no tanto

- Implementar una función `sumaDivisores :: Integer →Integer` que calcule la suma de los divisores de un número entero positivo.

problema *sumaDivisores*$(n : \mathbb{Z}) : \mathbb{Z}$ {
  requiere: $\{n > 0\}$
  asegura: $\{res = \sum_{i=1}^{n}$ if $(n \bmod i = 0)$ then i else 0 fi$\}$
}

Pregunta clave: ¿alcanza con hacer recursión sobre n?

No hay ninguna relación sencilla entre `sumaDivisores n` y `sumaDivisores (n−k)` (para ningún k particular).

¿Qué sucede si definimos primero una funcion **más general** que devuelve la suma de los divisores de un número hasta cierto punto?

```
sumaDivisoresHasta :: Integer → Integer → Integer
```

Ahora **sí** existe una relación sencilla entre `sumaDivisoresHasta n k` y `sumaDivisoresHasta n (k−1)`. ¿Por qué?

Generalización de funciones

Veamos cómo sería la especificación:

problema *sumaDivisoresHasta*$(n : \mathbb{Z}, k : \mathbb{Z}) : \mathbb{Z}$ {
  requiere: $\{(n > 0) \wedge (k > 0)\}$
  asegura: $\{res = \sum_{i=1}^{k}$ if $(n \bmod i = 0)$ then i else 0 fi$\}$
}

Ahora podemos definir esta función en Haskell recursivamente

```
sumaDivisoresHasta :: Integer → Integer → Integer
sumaDivisoresHasta n 1 = 1
sumaDivisoresHasta n i | (mod n i == 0) = i +          sumaDivisoresHasta n (i-1)
                       | otherwise = sumaDivisoresHasta n (i-1)
```

¿Y por último, cómo definimos `SumaDivisores` utilizando lo anterior?

```
sumaDivisores :: Integer → Integer
sumaDivisores n = sumaDivisoresHasta n n
```

Entonces, `SumaDivisores`, ¿es una función recursiva?

Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} i^j$$

Veamos primero la especificación:

problema *sumatoriaDoble*$(n : \mathbb{Z}, m : \mathbb{Z}) : \mathbb{Z}$ {
  requiere: $\{(n > 0) \wedge (m > 0)\}$
  asegura: $\{res = \sum_{i=1}^{n} \sum_{j=1}^{m} i^j\}$
}

Pregunta clave: ¿alcanza con hacer recursión sobre n?

¿Qué sucede si definimos primero una funcion **más específica** que devuelve la sumatoria interna?

```
sumatoriaInterna :: Integer → Integer → Integer
```

Ahora parece más sencillo definir `sumatoriaDoble n m` utilizando `sumatoriaInterna n m`. ¿Cómo lo hacemos?

Recursión en más de un parámetro

Veamos cómo sería la especificación:

problema *sumatoriaInterna*$(n : \mathbb{Z}, m : \mathbb{Z}) : \mathbb{Z}$ {
  requiere: $\{(n > 0) \wedge (m > 0)\}$
  asegura: $\{res = \sum_{j=1}^{m} n^j\}$
}

Ahora podemos definir esta función en Haskell recursivamente

```
sumatoriaInterna :: Integer → Integer → Integer
sumatoriaInterna _ 0 = 0
sumatoriaInterna n j = n^j + sumatoriaInterna n (j-1)
```

¿Y por último, cómo definimos `sumatoriaDoble` utilizando lo anterior?

```
sumatoriaDoble :: Integer → Integer → Integer
sumatoriaDoble 0 _ = 0
sumatoriaDoble n m = sumatoriaDoble (n-1) m + sumatoriaInterna n m
```

Entonces, `sumatoriaDoble`, ¿cuántas recursiones involucra?

Práctica 3: Ejercicio 6

Especificar e implementar la función `sumaDigitos :: Integer →Integer` que calcula la suma de dígitos de un número natural. Para esta función pueden utilizar `div` y `mod`.

Práctica 3: Ejercicio 7

Implementar la función `todosDigitosIguales :: Integer →Bool` que determina si todos los dígitos de un número natural son iguales, es decir:

problema *todosDigitosIguales*($n : \mathbb{Z}$) : *bool*{
  requiere: $\{(n > 0)\}$
  asegura: {*res* ↔ todos los dígitos de *n* son iguales }
}