Definición de funciones

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Objetivos

Objetivos

El objetivo de este tema es presentar las herramientas que existen en Haskell para la definición de funciones:

- el ajuste de patrones (*pattern matching*)
- expresiones condicionales
- ecuaciones con guardas
- ámbitos y definiciones locales
- expresiones lambda

Algunas se han visto de forma colateral en otros temas y otras son completamente nuevas.

Ajuste de patrones

Ajuste de patrones o ***pattern matching***:
Es el proceso de reconocer en un dato una estructura o pauta determinada, o detectar que no encaja dentro de esa estructura.

El ajuste de patrones se utiliza en informática por ejemplo a través de la idea de las expresiones regulares (*regex*).

El ajuste de patrones en Haskell permite, además de reconocer si un dato tiene una estructura determinada, asociar identificadores a cada parte de la estructura que se ha reconocido.
A ese proceso se le denomina **unificación**.

Ajuste de patrones

En Haskell (y en Matemáticas) una **función** puede definirse mediante varias **ecuaciones**.

A la hora de evaluar una función sobre un parámetro el ajuste de patrones se puede utilizar para determinar la ecuación a utilizar:

```haskell
fact :: Integer -> Integer

-- Ecuacion cuando el parametro tiene el valor 0
fact 0 = 1
-- Ecuacion para otros valores del parametro
fact n = n * fact (n-1)
```

Ajuste de patrones

Proceso:

- se comprueban los patrones en el orden dado en el programa haste que uno unifique
- en una misma ecuación, se intentan unificar los patrones de los argumentos de izquierda a derecha
- los argumentos no se evalúan completamente, sólo lo suficiente para ver si encajan o fallan el patrón
- en cuanto un argumento encaja, se pasa el siguiente argumento
- en cuanto un argumento falla, se pasa a la siguiente ecuación

El patrón más sencillo es un valor constante.

```haskell
not :: Bool -> Bool

not False = True
not True  = False
```

El símbolo `'_'` (comodín) representa (se *unifica*) con cualquier valor.

```haskell
not :: Bool -> Bool

not False = True
not _     = False

empty :: [a] -> Bool

empty [] = True
empty _  = False
```

El patrón más sencillo es un valor constante.

```
1 (&&) :: Bool -> Bool -> Bool
2 (&&) False _ = False
3 (&&) True  x = x
4
5 (||) :: Bool -> Bool -> Bool
6 True  || _ = True
7 False || x = x
```

El orden importa $\Rightarrow$ evaluación cortocircuitada.

Los patrones definidos para listas son los siguientes:

- `[]`: lista vacía
- `[x], [x,y], [x,y,z] ...`: se unifica con listas con un número determinado de elementos
- `(x:xs), (x:y:xs), (x:y:z:xs) ...` se unifica con listas que tengan **al menos** uno, dos, tres, ... elementos (`x,y,z`), más una cola (`xs`, puede que vacía).

Cada nombre simbólico del patrón queda ligado a la parte de la estructura de datos que le corresponde.

```
1 f (x:xs) = ....
2
3 f [1,2,3]
4   x  -> 1
5   xs -> [2,3]
```

```haskell
ones :: [Int]
ones = 1:ones

empty :: [a] -> Bool
empty []    = True
empty (_:_) = False
```

¿ Se puede evaluar esto ?

```
> empty ones
```

¿ Se puede evaluar esto ?

```
> empty ones
-- Intentamos aplicar las ecuaciones de empty
-- 'ones' no encaja con ninguna
-- Por la defincion de ones
empty 1:ones
-- Por la defincion de empty
empty (_:_)
-- Por la segunda ecuacion de empty
False
```

Los patrones definidos para tuplas son los siguientes:

- `(x)`, `(x,y)`, `(x,y,z)` `...`, se unifica con tuplas con un número determinado de elementos

Cada nombre simbólico del patrón queda ligado al elemento de la estructura de datos que le corresponde.

```haskell
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Se utilizan las propias parabras clave:

```haskell
data Color = Red | Orange | Yellow | Green | Blue | Indigo | Violet

isSemaphore :: Color -> Bool
isSemaphore Red    = True
isSemaphore Orange = True
isSemaphore Green  = True
isSemaphore _      = False
```

En los tipos union, las propias palabras clave pueden venir acompañadas de valores.

```
1  data Temperature = Farenheit Float | Celsius Float
2
3  switchUnits :: Temperature -> Temperature
4  switchUnits (Farenheit f) = Celsius ((f-32.0)*(5.0/9.0))
5  switchUnits (Celsius c)   = Farenheit (c*(9.0/5.0) + 32.0)
6
7  toCelsius :: Temperature -> Temperature
8  toCelsius (Farenheit f) = Celsius ((f-32.0)*(5.0/9.0))
9  toCelsius t             = t
```

En los tipos union, las propias palabras clave pueden venir acompañadas de valores.

```
data ExtendedFloat = Number Float | Infinity | Indeterminate
```

```
divide :: ExtendedFloat -> ExtendedFloat -> ExtendedFloat

divide Indeterminate _             = Indeterminate
divide _             Indeterminate = Indeterminate
divide Infinity      Infinity      = Indeterminate
divide (Number 0.0)  (Number 0.0)  = Indeterminate
divide _             (Number 0.0)  = Infinity
divide (Number _)    Infinity      = Number 0.0
divide (Number a)    (Number b)    = Number (a/b)
```

Pueden ser mucho más complicados (por ejemplo, tipos recursivos...)

```haskell
data Tree a = Null | Branch a (Tree a) (Tree a)

leftmost :: Tree a -> a
leftmost (Branch a Null _) = a
leftmost (Branch a t    _) = leftmost t
leftmost Null              = error "Arbol vacio"
```

Diferentes patrones para diferentes tipos de datos pueden combinarse entre sí:

```
1 f :: [(Integer,ExtendedFloat)] -> Float
2 f [] = ...
3 f (x:xs) = ..
4 f ((1,_):_) = ...
5 f (_:(_,Indeterminate):xs) = ...
```

Expresiones

Expresiones Condicionales

Las expersiones condicionales permiten seleccionar entre distintas ecuaciones para la evaluación de una expresión.

Las más importantes son **if** y **case**.

if

Selecciona entre dos opciones.

```
1 max x y = if x>y then x
2                  else y
3
4 abs x = if x>=0 then  x
5                 else -x
```

La condición debe evaluarse a un valor booleano.

case

Selecciona entre varias opciones.

La opción elegida se determina mediante ajuste de patrones.

La versión más simple se basa en usar patrones numéricos constantes, como otros lenguajes...

```
1 even x = case (x mod 2) of
2            0 -> True
3            1 -> False
```

case

...pero pueden utilizar cualquier tipo de datos...

```
1 conocido n = case n of
2                "Pedro" -> True
3                "Juan"  -> True
4                _       -> False
```

case

...e incluso patrones.

```
1 single x = case x of
2                []     -> False
3                [_]    -> True
4                (_:_)  -> False
```

El ámbito de cualquier definición en Haskell es global desde el momento en que se construye (declara).

A veces nos interesará definir elementos con visibilidad solamente local para la evaluación de una expresión o una definición.

Eso se consigue mediante las construcciones **let** y **where**.

let

Permite acotar un conjunto de definiciones para que solo sean visibles en una determinada expresión.

```
1  roots :: Float -> Float -> Float -> (Float,Float)
2
3  roots a b c = let
4                  disc  = b**2 - 4*a*c
5                  rdisc = sqrt disc
6                  denom = 2*a
7                in
8                  ( (-b-rdisc)/denom, (-b+rdisc)/denom )
```

where

Se puede utilizar **let** en cualquier punto del código.
Para el caso de definciones de funciones o comandos **do**, existe una sintaxis alternativa algo más clara, la construcción **where**:

```
1 roots :: Float -> Float -> Float -> (Float,Float)
2
3 roots a b c = ( (-b-rdisc)/denom, (-b+rdisc)/denom )
4               where
5                 disc  = b**2 - 4*a*c
6                 rdisc = sqrt disc
7                 denom = 2*a
```

Ecuaciones con Guardas

Ecuaciones con Guardas

La definición matemática de funciones se basa muchas veces en imponer condiciones a los argumentos para definir una ecuación determinada para la función en ese caso.

El ajuste de patrones nos permite imponer condiciones en la estructura de los datos (lista no vacía...) o incluso en su valor concreto (patrones constantes).
Pero no nos permiten imponer condiciones como 'que sea mayor que 5'.
Para esa tarea Haskell ofrece las *definiciones a trozos* o *ecuaciones con guardas*.

Ecuaciones con Guardas

```
1 abs x
2   | x >= 0  =  x
3   | x < 0   = -x
```

Para asegurarnos de que todos los casos posibles para el argumento están cubiertos podemos usar **otherwise**:

```
1 abs x
2   | x >= 0      =  x
3   | otherwise   = -x
```

Ecuaciones con Guardas

```
sign x
  | x > 0   =  1
  | x < 0   = -1
  | x == 0  =  0
```

O bien:

```
sign x
  | x > 0     =  1
  | x < 0     = -1
  | otherwise =  0
```

Ecuaciones con Guardas

Se pueden combinar con definiciones locales:

```
1  lendiff as bs
2    | la > lb   = la - lb
3    | lb > la   = lb - la
4    | otherwise = 0
5    where
6      la = length as
7      lb = length bs
8
9  main = do
10   print $ lendiff "hola"  "adios"
11   print $ lendiff "adios" "hola"
```

Funciones de orden superior

Funciones de orden superior

El uso de **funciones de orden superior** o **formas funcionales** require la definición de funciones, muchas veces de un sólo uso, y su paso de parámetros a otras funciones.

Para simplificar la escritura del código en esos casos, existen algunas herramientas fundamentales:

- las **expresiones-λ**
- el **currying** y las **secciones**
- la **composición** de funciones

Las expresiones-λ nos permiten escribir valores de tipo función sin asignarles un nombre; son lo que en otros lenguajes se denominan *funciones anónimas*:

```
\x      -> 2*x
\x y    -> x+y
\x y z -> x*y*z
```

Pueden tener cualquier número de parámetros.
Son especialmente útiles cuando trabajamos con funciones de orden superior.

```
> map (\x -> x+1) [3,4,5]
[4,5,6]
```

Currying: Todas las funciones reciben un solo argumento y devuelven otra función dependiente de un argumento menos (aunque se interpreten de forma tradicional como si tuvieran varios parámetros).
Esto permite lo que se denomina *aplicación parcial*.

```
1 div:: Int -> Int -> Int
2
3 > div 11 2
4 5
```

En realidad se ejecuta como:

```
1 div:: Int -> Int -> Int
2 -- div 11 :: Int -> Int
3 > (div 11) 2
4 5
```

Se puede utilizar la aplicación parcial con funciones de orden superior:

```
> map (\x -> div 11 x) [1,2,3,4]
[11,5,3,2]

> map (div 11) [1,2,3,4]
[11,5,3,2]
```

Haskell tiene una sintaxis particular para la aplicación parcial de **operadores infijos**.
Se especifica tan sólo uno de los argumentos del operador y el resultado es una función sobre el otro operando, que se introduce en la posición libre.

```
1 (2 ^) = \x -> 2 ^ x
2 (^ 2) = \x -> x ^ 2
```

Es bastante conveniente en muchos casos:

```
1 (1+) -- Incrementar
2 (*2) -- Duplicar
```

También se pueden utilizar con funciones de orden superior

```
> map (1+) [1,2,3,4,5]
[2,3,4,5,6]

> map (2^) [1,2,3,4,5]
[2,4,8,16,32]

> map (^2) [1,2,3,4,5]
[1,4,9,16,25]
```

Mini–problema:

```
> map (\x -> mod 3 x) [1,2,3,4]
[0,1,0,3]
> map (mod 3) [1,2,3,4]
[0,1,0,3]

> map (\x -> mod x 3) [1,2,3,4]
[1,2,0,1]
> -- ??? Sin expresion lambda ???
```

El operador `.` es el operador de composición de funciones.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
-- Posible interpretación
(.)       f           g        x      = f (g x)
```

```
> f :: Int -> Int
> f n = n + 1
> g :: Int -> Int
> g n = 2*n - 1
> h = f . g  -- h is the composition of f and g
> f (g 3)
6
> h 3
6
```

El operador `.` es el operador de composición de funciones.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
(+1) :: Int -> Int
odd  :: Int -> True

nextIsOdd = odd . (+1)
```

Es muy útil combinado con otros mecanismos de generar funciones anónimas.

```
> cesar d = map ( chr
                .(+ (ord 'a'))
                .(`mod` 26)
                .(+ d)
                .(subtract (ord 'a'))
                .ord )

> cesar 7 "haskell"
"ohzrlss"
```

Definición de funciones

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20