

Funciones definidas sobre conjuntos inductivos

Definición de funciones

Ahora veremos cómo definir una función cuyo dominio es un conjunto que está definido inductivamente. Lo introduciremos mediante un ejemplo:

Se quiere definir la función **factorial**, que dado un natural devuelve el factorial de dicho natural. Lo definiremos para el conjunto inductivo de los naturales. Entonces la debemos definir para todos los posibles elementos del conjunto \mathbb{N} , hay dos tipos, los que se construyen con la regla i) y los que se construyen con la regla ii). Tenemos que definir la función factorial para cada una de las reglas.

Dominio y codominio de la función:

`factorial: N → N`

Definición inductiva de N:

1. $0 \in \mathbb{N}$
2. $(S n) \in \mathbb{N}$ si $n \in \mathbb{N}$

Una definición inductiva para el tipo de los naturales en Haskell

```
data Nat = Zero | Succ (Nat) deriving Show
```

Con ésta línea estaremos definiendo en Haskell el conjunto inductivo \mathbb{N} , que es el conjunto de los naturales.

Definición de la función factorial:

Para la cláusula base:

1. `(factorial 0) = 1`

Esto se deduce de la definición de factorial.

Para la cláusula inductiva:

2. `(factorial (S n)) = (S n) * (factorial n)`

En este paso hemos usado la definición del factorial de un número, y también el razonamiento inductivo, que intuitivamente consiste en preguntarnos, ¿cómo hacemos para definir el factorial de un número natural cualquiera que no es 0, teniendo como elementos al propio número, a su predecesor y a la propia función factorial que estamos definiendo?

¿Cómo calcula nuestra función el factorial de (S (S (S (S (S 0)))) o 5?

Utilizaremos la representación 5, para facilitar la notación:

`(factorial 5)`, como 5 es de la forma `(S x)` se calcula con la segunda cláusula

`(factorial 5) = 5*(factorial 4)`, a su vez

`(factorial 4) = 4*(factorial 3)`

`(factorial 3) = 3*(factorial 2)`

`(factorial 2) = 2*(factorial 1)`

`(factorial 1) = 1*(factorial 0)`

Al calcular de `(factorial 0)` la única opción que queda es elegir la primera cláusula, entonces `(factorial 0) = 1` y aquí no hay más cálculos para hacer, ya que no se está invocando a la función factorial, sino que se está devolviendo un valor, el 1.

Ahora estamos en condiciones de responder a todos los cálculos intermedios que quedaron "esperando", ya que:

`factorial(0) = 1`, entonces

`factorial(1) = 1*1 = 1`, entonces

`factorial(2) = 2*1 = 2`, entonces

`factorial(3) = 3*2 = 6`, entonces

`factorial(4) = 4*6 = 24`, entonces

`factorial(5) = 5*24 = 120`

Hemos completado el cálculo de `factorial(5)`

A esta forma de definir funciones utilizando el hecho de que el dominio es un conjunto definido inductivamente se le denomina definición por *Análisis de Casos*, si además en la definición se usa la misma función que se está definiendo aplicada a un argumento "menor", se denomina definición por *Recursión*.

Análisis de casos y Recursión

Una función cuyo dominio es un conjunto definido inductivamente se puede definir utilizando el método de análisis de casos, también denominado *concordancia de patrones*.

Definir por casos o concordancia de patrones una función consiste en definirla para cada una de las cláusulas de la definición inductiva de su dominio.

Podemos decir que estamos utilizando *recursión* en la definición de una función si además de estar definida por casos, utilizamos la misma función que estamos definiendo pero aplicada a un elemento construido previamente que el elemento inductivo ('menor' desde el punto de vista de su estructura).

Ejemplo:

$(\text{factorial } (S\ n)) = (S\ n) * (\text{factorial } n)$

En el ejemplo en la definición de $(\text{factorial } (S\ n))$ se utiliza la propia función factorial, pero aplicada a n , que es un elemento "menor" que $(S\ n)$ desde el punto de vista de su construcción.

Las definiciones de funciones por recurrencia son correctas si las llamadas recursivas se realizan sobre elementos construidos previamente, o dicho de otra forma "menores" desde el punto de vista de la definición inductiva.

Definir la función factorial en Haskell, utilizar el conjunto Nat, del cual se da la definición anteriormente.

Ejemplo de una función definida utilizando Nat en Haskell, función máximo:

maximo:: Nat -> Nat -> Nat

maximo Zero x = x

maximo x Zero = x

maximo (Succ n) (Succ m) = (Succ (maximo n m))

Ejercicios:

1. Definir por concordancia de patrones la función suma: $N \rightarrow N \rightarrow N$, que dados dos naturales devuelve su suma.
2. Definir el operador lógico or como una función que recibe dos elementos del conjunto bool y devuelve un elemento de bool.
3. Definir la relación = entre naturales como una función que dados dos naturales devuelve un elemento de bool. Indicar previamente su tipo.
4. Definir la relación < entre naturales como una función que dados dos naturales devuelve un elemento de bool. Indicar previamente su tipo.
5. Definir la relación \geq entre naturales como una función que dados dos naturales devuelve un elemento de bool. Indicar previamente su tipo.
6. Definir la relación \leq entre naturales como una función que dados dos naturales devuelve un elemento de bool. Indicar previamente su tipo.
7. Definir la relación <> entre naturales como una función que dados dos naturales devuelve un elemento de bool. Indicar previamente su tipo.
8. Definir la función producto: $N \rightarrow N \rightarrow N$, que dados dos naturales, devuelve el producto de ambos.
9. Definir la función suma_hasta_n: $N \rightarrow N$, que dado un natural n, devuelve la suma de todos los naturales entre 0 y n.
10. Definir la función potencia: $N \rightarrow N \rightarrow N$, que dados dos naturales a y b, devuelve a^b .
11. Definir la función resta: $N \rightarrow N \rightarrow N$, que dados dos naturales a y b, devuelve $a-b$ si $a \geq b$, 0 en otro caso.
12. Definir el predicado unario EsPar, que dado un natural indica si el natural es par.
13. Definir el predicado unario EsImpar, que dado un natural indica si el natural es impar.
14. Definir todas las funciones anteriores en Haskell, utilizando el tipo Nat.

Algunas estructuras útiles de Haskell:

Expresiones condicionales

Una expresión condicional utiliza un valor de tipo bool para realizar una elección. Su forma general es la siguiente:

```
if expresión_booleana then exp1 else exp2
```

La expresión booleana se evalúa en primer lugar, si su valor es True entonces el condicional devuelve el valor exp1, si es False devuelve exp2.

Las expresiones denominadas exp1 y exp2 deben tener el mismo tipo, ese es el tipo de toda la expresión condicional.

Por ejemplo,

`if 2 < 3 then "pájaro" else "pescado"` es una expresión de tipo String y su valor es "pájaro".

Sin embargo, las siguientes expresiones son incorrectas, explicar por qué.

```
if 2 < 3 then 10
```

```
if 2 + 2 then 1 else 2
```

```
if True then "pájaro" else 7
```

- Definir la función absoluto que dado un entero devuelva su valor absoluto.

Uso de la palabra reservada de Haskell *otherwise*

Si estamos definiendo una función por casos y sabemos que no se aplicará ninguno de los casos listados más arriba se utiliza la palabra reservada de Haskell *otherwise*

Ejemplos:

```
min' :: Int → Int → Int
min' x y | x <= y = x
         | otherwise = y
```

Observar que este es un particular de la estructura **if..then..else**

```
min'' :: Int → Int → Int → Int
min'' x y z | x <= y && x <= z = x
            | y <= x && y <= z = y
            | otherwise = z
```

Variables y funciones locales

Constructor let

Hay muchas oportunidades en que tenemos que utilizar un valor ya calculado más de una vez. En lugar de recalcularlo varias veces, es mejor darles un nombre local, que pueda ser reutilizado.

Esto se hace con la expresión **let**. Su forma general es la siguiente:

```
let ecuación
    ecuación
    ...
    ecuación
in expresión
```

Toda esta construcción es sólo una gran expresión, y se puede usar en cualquier lugar donde *expresión* sea válida. Cuando se evalúa, las ecuaciones locales dan valores temporales a las variables que se encuentran a su izquierda. La *expresión* final luego del **in** es el valor de toda la expresión.

Por ejemplo, considere la siguiente definición de la función que resuelve la ecuación $ax^2+bx+c=0$ en el conjunto de los reales, esta función devuelve dos valores reales: (x1, x2)

```
cuadratica :: Float → Float → Float → (Float,Float)
cuadratica a b c = if a == 0 then
    error "no es cuadrática"
  else
    let d = b^2 - 4*a*c
        in
        if d < 0 then
            error "no tiene soluciones reales"
        else
            ((- b + sqrt d) / 2*a, (- b - sqrt d) / 2*a)
```

La expresión **let** introduce la variable **d**, cuyo valor puede ser utilizado únicamente localmente. Fuera del **let** el nombre **d** no está definido.

Constructor where

Haskell provee dos maneras de definir localmente las funciones y variables auxiliares: los constructores **let** y **where**. El ejemplo anterior se podría haber definido con **where** de la siguiente manera:

```
cuadratica' :: Float → Float → Float → (Float,Float)
cuadratica' a b c = if a == 0 then
    error "no es cuadrática"
  else
    ((- b + sqrt d) / 2*a, (- b - sqrt d) / 2*a)
  where
    d = b^2 - 4*a*c
```

Observar que en esta definición no se contempla el caso en que no tenga soluciones reales, pensar como resolver este problema en Haskell.