Programación Declarativa, 2021-1
Nota de clase 11: Tipos de Datos*

Manuel Soto Romero

24 de noviembre de 2020
Facultad de Ciencias UNAM

Un tipo es en pocas palabras, una abstracción para un conjunto de valores [2]. Esto es, a un conjunto de valores con características similares y operaciones equivalentes, se les pone una etiqueta indicando que pertenecen a un determinado tipo. Por ejemplo, en Haskell el tipo de dato `Int` es la abstracción para el conjunto de números enteros, mientras que `Bool` es la abstracción para el conjunto de las constantes lógicas (`True` y `False`) [3].

Haskell cuenta con un sistema de tipos muy poderoso y estricto que permite inferir el tipo de una expresión sin necesidad de especificarlo explícitamente. Es un lenguaje tipificado estáticamente, lo cual permite al compilador detectar errores antes de ejecutar cualquier programa. Por ejemplo, si se intenta realizar la suma de un número con una cadena, se reporta un error [3].

```
Prelude> 1 + ‘‘Hola’’
<interactive>:1:1: error:
• No instance for (Num [Char]) arising from a use of ‘+’
• In the expression: 1 + "Hola"
In an equation for ‘it’: it = 1 + "Hola"
```

Para conocer el tipo de una expresión, el intérprete de GHC provee el comando `:type` o su versión simplificada `:t` [3].

```
Prelude> :t True
True :: Bool
Prelude> :t [True,False,True]
[True,False,True] :: [Bool]
Prelude> :t take
take :: Int -> [a] -> [a]
```

En los ejemplos anteriores, la función `take` de Haskell tiene asociado el tipo `Int` $\rightarrow$ `[a]` $\rightarrow$ `[a]`, esto no quiere decir que `a` sea un tipo como tal, en realidad representa una *variable de tipo*. Una variable de tipo, en Haskell, es una variable que puede tomar cualquier tipo de dato, por ejemplo, una lista puede ser de tipo `Int` o `String` sin definir una función para cada tipo de dato. Las funciones que usan variables de tipo son llamadas *funciones polimórficas*[3].

Además de los tipos básicos del lenguaje, Haskell provee distintas primitivas para renombrar tipos, crearlos y agruparlos en clases o familias de acuerdo a su comportamiento. En esta nota se revisan estos elementos del lenguaje de programación.

*Basada en *Manual de Prácticas para la Asignatura de Programación Declarativa.*

11.1. Sinónimos de tipo

Un *sinónimo de tipo* es una forma que se tiene para renombrar un tipo de dato, la cual permite definir funciones de una forma más legible, pues hacen más claro el uso del tipo en la solución de un problema dado [3]. Un ejemplo ilustrativo sobre el uso de sinónimos, son las cadenas, si se consulta el tipo de una cadena desde el intérprete se obtiene:

```
Prelude> :t "Hola"
"Hola" :: [Char]
```

Esto se debe a que `String` es un sinónimo de `[Char]`, las cadenas son en realidad listas de caracteres, es una forma de azúcar sintáctica. Para definir un sinónimo se usa la palabra reservada `type` [3]. Por ejemplo, en el Código 1 se usa el sinónimo `Punto` para representar puntos en el plano y calcular su distancia. La definición del sinónimo se aprecia en la línea 2. La línea 5 define la firma de la función a través del nuevo sinónimo, pero su uso se mantiene mediante el uso de tuplas como se muestra en la caza de patrones.

```
-- Un punto es una pareja de flotantes
type Punto = (Float , Float)

-- Función que calcula la distancia entre dos puntos.
distancia :: Punto → Punto → Float
distancia (x1,y1) (x2,y2) = sqrt ((x2 - x1)**2 + (y2 - y1)**2)
```

Código 1: Función que calcula la distancia entre dos puntos

Una vez que se define un sinónimo, puede usarse en la firma de una función como si se tratara de cualquier otro tipo de dato del lenguaje. De la misma forma, se puede usar caza de patrones con el tipo al que se asocia el sinónimo [3].

También es posible definir sinónimos que reciben parámetros, éstos suelen utilizarse por ejemplo, cuando se definen estructuras de datos y se necesita que éstas sean genéricas. Por ejemplo, en el Código 2 se muestra la implementación de una pila mediante sinónimos. La línea 2 define una pila con un parámetro `t` que indica que es sinónimo de una lista de ese tipo, `t` es una variable de tipo, por lo tanto pueden definirse funciones polimórficas sobre este sinónimo como lo hacen las funciones `pop` y `push`, esto puede apreciarse en el tipo `Pila Int` de las firmas de estas funciones.

```
-- Se representa una pila con listas.
type Pila t = [t]

-- Función que extrae el último elemento que ingresó en una
-- Pila de números enteros .
pop :: Pila Int → Int
pop [] = error "La pila está vacía"
pop (x:xs) = x

-- Función que añade un elemento en una Pila de números enteros.
push :: Int → Pila Int → Pila Int
```

```
push x p = x:p
```

Código 2: Pilas

El parámetro del sinónimo `Pila` es `t` y puede usarse reemplazándolo por cualquier tipo de dato del lenguaje, otro sinónimo o alguno otro definido por el programa. En el Código anterior se usar pilas de números enteros `(Pila Int)`, sin embargo, pudo usarse una lista de números flotantes `(Pila Float)` o incluso se pudieron definir usando variables de tipo `(Pila a)` para indicar que pueden ser de cualquier tipo.

11.2. Tipos de Datos Algebraicos

Adicional a los tipos de datos que provee HASKELL, es posible definir tipos de datos a partir de sus constructores, mismos que representan los valores del tipo. Se conoce a éstos como *Tipos de Datos Algebraicos* y se definen a través de los valores que pertenecen al tipo de dato (constructores) y algunas operaciones asociadas a los mismos.

Un ejemplo de Tipo de Dato Algebraico es el tipo `Natural`. Este tipo de dato representa a los números mediante un constructor para el cero y a partir de éste, se usa la función sucesor para generar el resto de números. Algunas operaciones asociadas a los números naturales son:

suma : `Natural` $\times$ `Natural` $\rightarrow$ `Natural`
producto : `Natural` $\times$ `Natural` $\rightarrow$ `Natural`

Para implementar un Tipo de Dato Algebraico, se puede utilizar la primitiva `data` que permite crear nuevos tipos de datos a partir de sus *constructores de datos* y *constructores de tipo* [3]. Como se aprecia en el Código 3 todo lo que se encuentra a la derecha del símbolo `=` es un *constructor de datos* y todo lo que aparece a la izquierda del mismo, es un *constructor de tipos*.

```
data Natural = Cero
             | Suc Natural
```

Código 3: Definición del tipo `Natural`

11.2.1. Constructores de datos

Al definir un Tipo de Dato Algebraico, se deben especificar uno o más constructores de datos que pueden recibir uno o más parámetros. A través de estos parámetros se indica el tipo de dato que encapsula dicho constructor[3].

Los constructores de datos son en realidad funciones que devuelven valores del tipo de dato que se está definiendo [3]. Por ejemplo, para el tipo `Natural`, las funciones asociadas a los constructores son:

`Cero` : `Natural`
`Suc` : `Natural` $\rightarrow$ `Natural`

11.2.2. Constructores de tipo

Al definir un nuevo tipo de dato, se proporciona un nombre para el mismo y, al igual que con los constructores de datos, se puede proporcionar algún parámetro, lo cual permite (entre otras cosas) definir estructuras de datos genéricas[3].

Por ejemplo, en el Código 4 se define el tipo de dato `ArbolB` que representa árboles binarios que almacenan números enteros en sus nodos. Se especifican dos constructores, el árbol vacío y el árbol con un elemento (entero) y dos subárboles izquierdo y derecho respectivamente.

```
data ArbolB = Void
            | Tree Int ArbolB ArbolB
```

Código 4: Definición del tipo `ArbolB`

Para permitir que el tipo `ArbolB` sea genérico, esto es, que se puedan construir árboles binarios que almacenen cualquier tipo de dato, se agrega un parámetro (variable de tipo) al constructor de tipos `ArbolB`, como se muestra en el Código 5.

```
data ArbolB a = Void
              | Tree a (ArbolB a) (ArbolB a)
```

Código 5: Definición del tipo `ArbolB` genérico

La variable de tipo `a` permite construir estructuras de datos homogéneas de forma genérica. Los parámetros del constructor de datos `Tree` también se modifican, pues el tipo de dato requiere que se proporcione un tipo de dato como parámetro.

11.2.3. Familias de tipos

En Haskell, existen dos formas de especificar constructores de tipo, ya sea simplemente mediante el nombre del tipo de dato o con la posibilidad de agregar uno o más parámetros. Estas formas de definir datos a través de sus constructores, forman lo que se conoce como *familias de tipos*[1] [3].

Una familia de tipos es una forma de indicar el *tipo de un tipo* [3]. Esto es útil, entre otras cosas, para identificar cuál es la estructura del tipo y si éste recibe argumentos o no. Para conocer la familia a la que pertenece un tipo, se usa el comando `:kind` o su abreviatura `:k`:

```
Ejemplo> :k Natural
Natural :: *
Ejemplo> :k ArbolB
ArbolB :: * -> *
Ejemplo> :k Bool
Bool :: *
Ejemplo> :k [Int]
[Int] :: *
Ejemplo :k (Int,Float)
(Int,Float) :: *
```

[1] *Kinds* en inglés

Un tipo de dato que no recibe parámetros en su definición, se representa con la familia `*` y se dice que es un *tipo concreto*, mientras que si el tipo recibe un parámetro, se representa como `* → *`, esto es, recibe un tipo concreto y genera un nuevo tipo [3]. Si el constructor de tipos del tipo, recibe más de un parámetro se denota poniendo tantos símbolos `*` como argumentos se pasan al tipo, incluyendo otro para el tipo de regreso.

11.2.4. Uso de tipos de datos algebraicos

Una vez que se define un tipo de dato, es posible usar sus constructores en el intérprete como cualquier otro valor, sin embargo, es necesario indicarle al intérprete cómo debe mostrar este tipo de dato en pantalla, de lo contrario, se genera un error.

```
Ejemplo> Void
<interactive>:2:1: error:
· No instance for (Show (ArbolB a0)) arising from a use of ‘print’
· In a stmt of an interactive GHCi command: print int
```

Para que HASKELL muestre un tipo de dato, éste debe pertenecer a la clase de tipos `Show`. Una *clase de tipos* es una especie de interfaz que define el comportamiento de un grupo de tipos. Por ejemplo, la clase de tipos `Show`, define el comportamiento necesario para mostrar tipos de datos en pantalla, a través de la función `show` y todo tipo que pertenece a esta clase debe implementar dicha función o *derivar* la implementación por defecto de la clase.

En el Código 6 se modifica el tipo `ArbolB` que deriva a la clase de tipos `Show`. La línea 2 muestra la adición del código `deriving (Show)` que le indica al tipo que debe derivar de dicha clase. En general, esta sintaxis se usa para derivar de otras clases, no únicamente de la clase `Show`.

```
data ArbolB a = Void
              | Tree a (ArbolB a) (ArbolB a) deriving (Show)
```

Código 6: Definición del tipo `ArbolB` genérico que deriva de la clase de tipos `Show`

```
Ejemplo> Void
Void
Ejemplo> Tree 1 Void Void
Tree 1 Void Void
Ejemplo> Tree 1 (Tree 2 Void Void) (Tree 3 Void Void)
Tree 1 (Tree 2 Void Void) (Tree 3 Void Void)
```

La ejecución del Código 6 muestra como los constructores son impresos en pantalla tal y como se definieron dentro del código. Más adelante se muestra cómo modificar la instancia de la clase `Show` para que muestre los constructores de unas forma más amigable para el usuario, siendo ésta definida por el programador. De la misma forma, más adelante se estudian con profundidad las clases de tipos.

11.2.5. Funciones sobre tipos de datos algebraicos

Es posible definir funciones que reciban como parámetro o regresen tipos de datos algebraicos. Para definir este tipo de funciones suele usarse la técnica de caza de patrones, pues permite procesar de forma sencilla los constructores definidos con sus parámetros.

En el Código 7 se definen las funciones que realizan la suma y producto de dos números naturales respectivamente. Se usan los constructores de datos como patrones. Las líneas 2, 3, 6 y 7 muestra el uso de los patrones para definir las funciones. Los patrones, en este caso, son los constructores que se indicaron al definir del tipo de dato.

```
suma :: Natural → Natural → Natural
suma Cero n = n
suma (Suc n) m = Suc (suma n m)

producto :: Natural → Natural → Natural
producto Cero _ = Cero
producto (Suc n) m = suma m (producto n m)
```

Código 7: Funciones `suma` y `producto` sobre números naturales

```
Ejemplo> suma (Suc (Suc Cero)) (Suc (Suc (Suc Cero)))
Suc (Suc (Suc (Suc (Suc Cero))))
Ejemplo> producto (Suc (Suc (Suc Cero))) (Suc (Suc Cero))
Suc (Suc (Suc (Suc (Suc (Suc Cero)))))
```

La ejecución del Código 7 muestra la suma de los números 2 y 3 y el producto de los números 3 y 2.

En el Código 8 se definen funciones para recorrer árboles binarios en *inorden*, *preorden* y *postorden* respectivamente. Para devolver los recorridos se usan listas. La ejecución del Código 8 muestra la ejecución de las tres funciones. Se usan paréntesis para agrupar cada número. Recordar que los constructores de datos son funciones y por lo tanto, deben usarse espacios para delimitar sus parámetros.

```
inorden :: (ArbolB a) → [a]
inorden Void = []
inorden (Tree cbz izq der) =
   (inorden izq) ++ [cbz] ++ (inorden der)

preorden :: (ArbolB a) → [a]
preorden Void = []
preorden (Tree cbz izq der) =
   [cbz] ++ (preorden izq) ++ (preorden der)

postorden :: (ArbolB a) → [a]
postorden (Tree cbz izq der) =
   (postorden izq) ++ (postorden der) ++ [cbz]
```

Código 8: Funciones `inorden`, `preorden`, `postorden` sobre árboles binarios

```
Ejemplo> let a = Tree 1 (Tree 2 Void Void) (Tree 3 Void Void)
Ejemplo> inorden a
[2,3,1]
Ejemplo> preorden a
[1,2,3]
Ejemplo> postorden a
[2,3,1]
```

11.2.6. Sintaxis de registro

Además de la definición de tipos de datos algebraicos mediante sus constructores de datos y de tipo, es posible usar una sintaxis de registro que permite acceder de forma sencilla a los parámetros de los constructores de datos [3].

Por ejemplo, en el Código 9 se define el tipo de dato `Figura` que incluye dos constructores, uno para representar cuadrados (línea 1) y otro para representar círculos (línea 2).

```
data Figura = Cuadrado Float
            | Circulo Float (Float,Float) deriving (Show)
```

Código 9: Definición del tipo `Figura`

El constructor `Cuadrado`, define cuadrados a partir de la longitud de sus lados, mientras que el constructor `Circulo` definir círculos a partir de su radio y centro. Una vez definida la figura, se necesitarían definir las funciones del Código 10 para acceder a sus parámetros. Las líneas 1 y 2 definen la función que obtiene el lado de un cuadrado; las líneas 4 y 5 obtienen el radio de un círculo y finalmente las líneas y 7 y 8 obtienen el centro de un círculo.

```
getLado :: Cuadrado → Float
getLado (Cuadrado lado) = lado

getRadio :: Circulo → Float
getRadio (Circulo radio _) = radio

getCentro :: Circulo → (Float,Float)
getCentro (Circulo _ centro) = centro
```

Código 10: Funciones para acceder a los parámetros del tipo `Figura`

La definición de estas funciones, es un principio, sencilla, pero si se desea obtener el valor de los parámetros para tipos de datos con más constructores, se vuelve una tarea tediosa.

La *sintaxis de registro* facilita esta tarea, mediante la asignación de un nombre a los parámetros de los constructores de datos. En el Código 11 se modifica el tipo de dato `Figura` usando sintaxis de registro. El constructor `Cuadrado` tiene ahora un parámetro `lado` (línea 1) y el constructor `Circulo` tiene ahora dos parámetros `radio` y `centro` respectivamente (línea 2).

```
data Figura = Cuadrado {lado :: Float}
            | Circulo {radio :: Float, centro :: (Float,Float)}
            deriving (Show)
```

Código 11: Tipo de dato `Figura` con sintaxis de registro

```
Ejemplo> let a = Circulo {centro = (2.0,3.0), radio = 10.0}
Ejemplo> let b = Cuadrado {lado = 2.0}
Ejemplo> radio a
10.0
Ejemplo> centro a
(2.0,3.0)
Ejemplo> lado b
2.0
```

La ejecución del Código 11 muestra la creación de un `Círculo` y un `Cuadrado` usando sintaxis de registro. Esta forma de definir tipos de datos, crea automáticamente las funciones que se definieron anteriormente, permitiendo obtener el valor de los parámetros de un tipo de dato de forma más clara y rápida [3]. Además, permite construir valores del tipo, especificando a qué parámetro corresponde el valor asignado.

11.3. Clases de tipos

Como se mencionó anteriormente, una clase de tipos es una especie de interfaz que define el comportamiento de un conjunto de tipos [4]. En la sección anterior se revisó la clase de tipos `Show` que indica a Haskell cómo mostrar un tipo de dato en pantalla, sin embargo, existen otras clases de tipos [3] [2]:

`Eq` Esta clase representa los tipos de datos que permiten hacer comparaciones por igualdad. Los miembros de esta clase deben implementar las funciones `==` o `/=`.

`Ord` Esta clase representa a los tipos de datos que poseen algún tipo de orden. Los miembros de esta clase deben implementar las funciones `<`, `>`, `>=` y `<=`. Además, es necesario que formen parte de la clase `Eq`.

`Read` Esta clase es contraria a `Show`. Los miembros de esta clase, pueden obtenerse a partir de cadenas, es decir, dada una cadena, se puede convertir a otro tipo de dato. Los miembros de esta clase deben implementar la función `read`.

`Enum` Esta clase representa a los tipos de datos que poseen un orden secuencial. De esta forma se pueden usar las funciones `succ` y `pred` para recorrer los valores del tipo de dato.

`Bounded` Esta clase representa a los tipos de de datos que poseen un límite inferior y superior, esto es, si están acotados.

`Num` Esta clase representa a los tipos de datos que pueden comportarse como números. Para que un tipo pertenezca a esta clase, también debe formar parte de `Show` y `Enum`.

[2] Para consultar el comportamiento de otras clases, se puede consultar el sitio oficial de Haskell: https://www.haskell.org/hoogle/

`Integral` Al igual que `Num`, es una clase numérica, sin embargo, sólo acepta números enteros. `Int` e `Integer` forman parte de esta clase.

`Floating` Al igual que `Num`, es una clase numérica, sin embargo, sólo acepta números flotantes. `Float` y `Double` forman parte de esta clase.

Para hacer que un tipo de dato forme parte de una clase, puede usarse una derivación de ésta. Cada clase incluye un grupo de funciones mínimas que se deben implementar para que el tipo forme parte de la clase. Al derivar una clase, se toma la implementación de las funciones por defecto de la misma. Si no se desea derivar la clase, se debe crear una nueva instancia de las funciones [3].

Por ejemplo, en el Código 12 se implementa la función `show` para que el tipo de dato `Natural` forme parte de la clase `Show`, en lugar de sólo derivar el tipo de dato. La función se define usando la técnica de caza de patrones, esta función debe regresar cadenas. Las líneas 4 y 5 muestran el mapeo entre datos de tipo `Natural` y `String`. La ejecución del Código 12 muestra la impresión en el formato especificado.

```
data Natural = Cero | Suc Natural

instance Show Natural where
  show Cero = "0"
  show (Suc n) = "(S " ++ show n ++ ")"
```

Código 12: Instancia de la función `show` para el tipo `Natural`

```
Ejemplo> Cero
0
Ejemplo> Suc Cero
(S 0)
Ejemplo> Suc (Suc Cero)
(S (S 0))
Ejemplo> Suc (Suc (Suc (Suc Cero)))
(S (S (S (S 0))))
Ejemplo> Suc $ Suc $ Suc $ Suc Cero
(S (S (S (S 0))))
```

11.3.1. Funciones con restricciones de clase

Las clases de tipos pueden usarse para restringir los parámetros de una función. Esto permite garantizar la robustez de la función y su integridad. Para incluir una restricción de clase en una función se debe anteponer la firma de la función, la clase a la que pertenece un determinado parámetro, el símbolo `=>` y el resto de la firma [3]. Por ejemplo, la función `==` tiene la siguiente firma:

```
(==) :: (Eq a) => a -> a -> Bool
```

Esta restricción indica a la función, que los parámetros de la misma deben ser miembros de la clase `Eq`, es decir, que puedan ser comparados por igualdad.

Otros ejemplos de restricciones de clase:

```
Ejemplo> :t (+)
(+) :: Num a => a -> a -> a
Ejemplo> :t (*)
(*) :: Num a => a -> a -> a
Ejemplo> :t (/)
(/) :: Fractional a => a -> a -> a
Ejemplo> :t (div)
(div) :: Integral a => a -> a -> a
Ejemplo> :t (^)
(^) :: (Num a, Integral b) => a -> b -> a
Ejemplo> :t (<)
(<) :: Ord a => a -> a -> Bool
Ejemplo> :t (>)
(>) :: Ord a => a -> a -> Bool
Ejemplo> :t (<=)
(<=) :: Ord a => a -> a -> Bool
Ejemplo> :t (>=)
(>=) :: Ord a => a -> a -> Bool
Ejemplo> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
Ejemplo> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
Ejemplo> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

11.3.2. Creación de clases de tipos

Además de usar las clases de tipos como restricciones en funciones, también es posible definir clases de tipos propias para derivarlas, crear instancias de sus funciones y usarlas como restricciones de clase en funciones[3].

En el Código 13, se muestra una posible definición de la clase `Eq`. La palabra reservada `class` (línea 1) permite definir nuevas clases, especificando un nombre, un parámetro que representa a los tipos que formarán parte de la clase y se especifican las funciones requeridas para pertenecer a esta clase (líneas 2 y 3), incluyendo, si se desea, una implementación por defecto para estas funciones (lineas 5 y 6) [3].

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)
```

Código 13: Definición de la clase de tipos `Eq`

También es posible, indicar que una clase sea subclase de otra, es decir, que además de las funciones requeridas, se debe garantizar que el tipo de dato pertenezca a otra clase. Por ejemplo, la clase `Num`,

necesita que los tipos sean miembros de la clase `Eq`, es decir, `Num` es subclase de `Eq`. En el Código 14 se muestra la declaración de la clase `Num`.

```
class (Eq a) => Num a where
  ...
```

Código 14: Definición de la clase de tipos `Num`

En el Código 15, se muestra la definición de la clase `Coleccion` (líneas 2 a 7) que permite definir estructuras de datos con las operaciones de inserción, borrado y búsqueda (líneas 5 a 7). Se incluye una restricción de clase en las operaciones para que los elementos de las estructuras puedan ser comparados y se hace al tipo `Lista` parte de `Coleccion` (líneas 14 a 26) y `Show` (líneas 30 a 34).

```
-- Clase de tipos Coleccion para estructuras de datos.
class Coleccion a where

  -- Funciones mínimas para pertenecer a Coleccion
  agrega :: (Eq e) => a e -> e -> a e
  eliminaUltimo :: (Eq e) => a e -> a e
  contiene :: (Eq e) => a e -> e -> Bool

-- Tipo de dato abstracto Lista
data Lista a = Empty
             | Cons a (Lista a)

-- Implementación de la clase Coleccion
instance Coleccion Lista where

  -- agrega
  agrega lista e = (Cons e lista)

  -- eliminaUltimo
  eliminaUltimo Empty = Empty
  eliminaUltimo (Cons x Empty) = Empty
  eliminaUltimo (Cons x xs) = Cons x (elimina xs)

  -- contiene
  contiene Empty _ = False
  contiene (Cons x xs) e = x == e || contiene xs e

-- Implementación de la clase Show
-- La variable de tipo a debe ser parte de Show.
instance (Show a) => Show (Lista a) where

  -- show
  show Empty = "[]"
  show lista = "[" ++ (construye lista) ++ "]"

-- Función que convierte una lista a cadena.
construye :: (Show a) => (Lista a) -> String
```

```
construye (Cons x Empty) = show x
construye (Cons x xs) = show x + ", " ++ construye xs
```

Código 15: Clase de tipos `Coleccion`

Referencias

[1] Manuel Soto, *Manual de Prácticas para la Asignatura de Programación Declarativa,* Facultad de Ciencias UNAM, Proyecto de Titulación, 2019.

[2] Krishnamurthi Shriram, *Programming Languages: Application and Interpretation*, Primera Edición, 2007.

[3] Miran Lipovaca, *Learn You a Haskell for Great Good! A Beginner's Guide*, Primera Edición, No Starch Press, 2011.

[4] Graham Hutton, *Programming in Haskell*, Tercera Edición, Cambridge University Press, 2008.