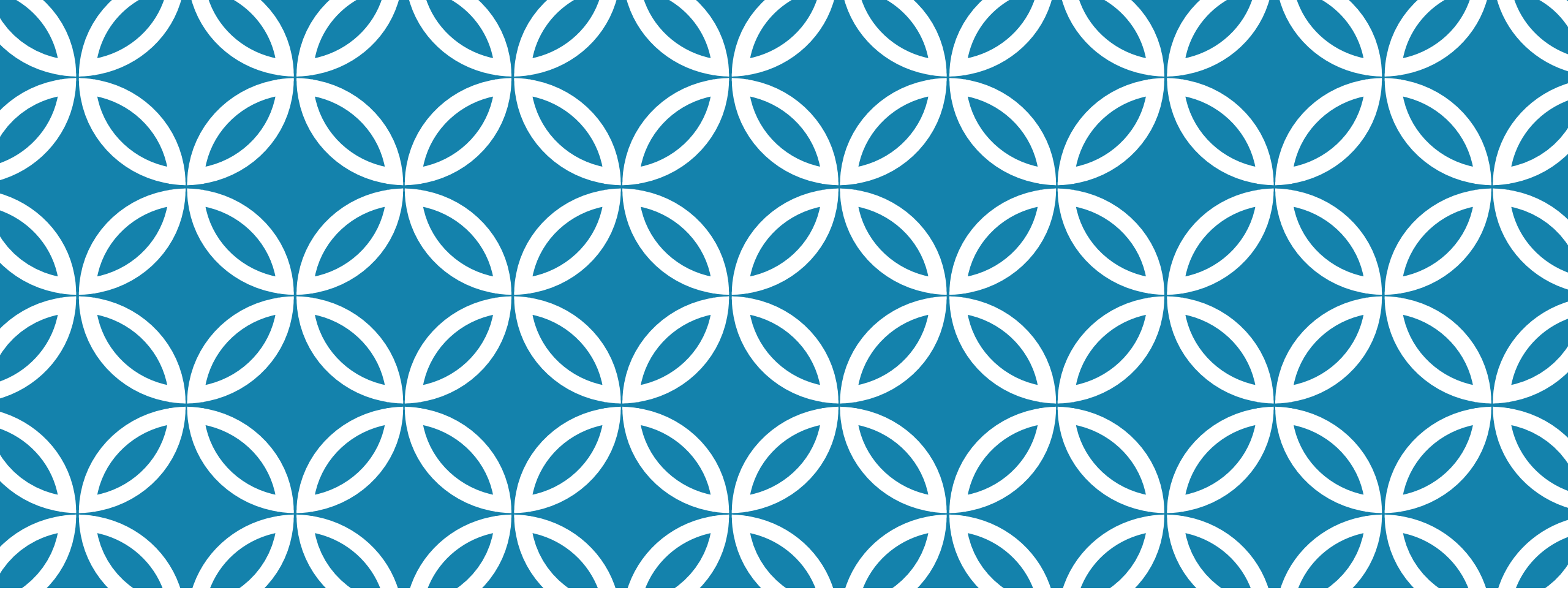


ESTRUCTURAS DISCRETAS

Clases de Tipos y Origami

Javier Enríquez Mendoza Mauricio Hernández Olvera

TIPOS

Un tipo es una abstracción para un conjunto de valores.

A un conjunto de valores con características similares y operaciones equivalentes, se les pone una etiqueta indicando que pertenecen a un determinado tipo.

En Haskell el tipo de dato **Int** es la abstracción para el conjunto de números **enteros.**

Bool es la abstracción para el conjunto de las **constantes lógicas** (True y False).

SISTEMA DE TIPOS

Haskell tiene un sistema de tipos que permite inferir el tipo de una expresión sin necesidad de especificarlo explícitamente.

Todos los elementos del lenguaje tienen asociado un tipo de dato.

Esto permite encontrar errores antes de la ejecución de un programa.

Por ejemplo al intentar aplicar una operación definida para Int a un Bool, el compilador lanza un error.

VARIABLES DE TIPO

Como hemos visto durante el curso, para hacer funciones un poco mas generales, podemos definir las firmas de éstas de la siguiente manera:

```
cabeza :: [ a ] -> a
```

Esto no quiere decir que **a** sea un tipo. En realidad **a** es una variable de tipo

En Haskell una variable de tipo es una variable que puede tomar cualquier tipo de dato.

Las funciones que usan variables de tipo son llamadas **funciones polimorficas.**

TIPOS EN HASKELL

En Haskell los tipos se dividen en 2 grandes categorías:

Tipos de dato Primitivos.

Tipos de dato definidos por el Programador.

TIPOS DE DATO PRIMITIVOS

Los tipos de dato primitivos son aquellos que ya estan definidos en el kernel del lenguaje Estos a su vez se dividen en 2.

Tipos Elementales

- Char
- Int
- Bool
- Float

Tipos Estructurados

- Listas
- Tuplas

TIPOS DEFINIDOS POR EL PROGRAMADOR

Hay dos formas de definir tipos de datos en Haskell.

Sinónimos de tipo

- Usamos la palabra reservada **type**.
- Definimos un tipo con base en los tipos que ya están definidos previamente.
- Sirve para renombrar un tipo de dato y hacer funciones mas legibles.

Tipos de dato Algebraicos

- Se usa la palabra reservada **data**.
- Se definen cada uno de sus constructores.
- Son abstractos porque la definición de estos no nos dice nada de su comportamiento.

CLASES DE TIPOS

Una clase de tipo es una especie de interfaz que define comportamiento de un conjunto de tipos. Tenemos las siguientes :

- **Eq** permite hacer comparaciones por igualdad.
- **Ord** los tipos de datos poseen un orden.
- **Show** indica como mostrar un tipo de dato en pantalla.
- **Read** dada una cadena, puede convertirse a otro tipo de dato.
- **Enum** los tipos de datos poseen un orden secuencial.
- **Bounded** representa a los tipos de datos que poseen un límite inferior y superior.
- **Num** los tipos de dato que se comportan como números.
- **Integral** una clase numérica que solo acepta enteros, Int e Integer son parte de ésta clase.
- **Floating** clase numérica que acepta flotantes, Float y Double forman parte de ésta.

DERIVACIONES E INSTANCIAS

Para hacer que un tipo de dato forme parte de una clase, se puede usar una derivación o una implementación de la clase.

Cada clase incluye n grupo de funciones necesarias para que el tipo forme parte de ella.

Si se hace una derivación se tomara la definición por defecto de estas funciones.

Si se desea definir el comportamiento de estas funciones, entonces creamos una instancia de la clase en la que tenemos que definir cada una de ellas.

RESTRICCIONES DE CLASE

Las clases de tipos pueden usarse también para restringir el parámetro de una función.

Esto con el fin de garantizar la robustez e integridad de una función.

Para incluir una restricción se usa **=>** en la firma de la función. Con la restricción del lado izquierdo y el resto de la firma del lado derecho.

Por ejemplo, la función == tiene la siguiente firma:

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

Esta restricción indica que los parámetros de la función deben de ser miembros de la clase **Eq.**

FUNCIONES DE ORDEN SUPERIOR

En Haskell las funciones son de primera clase, esto quiere decir que pueden ser pasadas como parámetros, regresadas por otra función o almacenadas en una estructura de datos.

Cuando una función recibe como parámetro a otra decimos que es una **función de orden superior.**

Durante el semestre vimos 2 funciones de orden superior muy conocidas, **map** y **filter.**

- **map** recibe una lista y una función y aplica la función a cada elemento de la lista.
- **filter** recibe un predicado (función que regresa un booleano) y regresa una lista con los elementos de la original que cumplen el predicado.

Hay otras funciones de orden superior también muy famosas llamadas **folds** o funciones de plegado.

FOLDS

Cuando comenzamos a trabajar con listas y recursión, notamos que la mayoría de las funciones definidas para ellas siguen el siguiente patrón:

- Definen un comportamiento para el caso base [ ]
- Y otro para el caso de la lista con cabeza y cola (x:xs)

Resulta que este es un patrón muy común. Así que se definieron funciones muy útiles que lo encapsularan. Los **folds.**

Su comportamiento es muy parecido al de **map** solo que estas funciones reducen la lista a un solo valor.

```
fold :: (a -> b -> b) -> b -> [a] -> b
```

ENCAPSULAMIENTO

El patrón de las operaciones sobre listas es el siguiente

- caso base para [ ]
- caso recursivo para (x:xs)

Esas funciones se pueden encapsular con en el siguiente patrón:

- fold :: (a -> b -> b) -> b -> [a] -> b
- fold f e [ ] = e
- fold f e (x:xs) = f x (fold f e xs)

COMPORTAMIENTO

Fold toma una función binaria, un valor de inicio y una lista para doblar.

Fold va a ir aplicando la función binaria a los elementos de la lista mientras la va doblando hasta que al final quede reducida a un solo valor.

Pero como tenemos dos formas de recorrer la lista entonces podemos definir dos tipos de folds.

Uno que comience con el primer elemento y otro que lo haga con el ultimo.

Entonces tenemos un foldl y un foldr.

FOLDR Y FOLDL

El fold que definimos anteriormente es foldr pues comienza a aplicar la función a partir del ultimo elemento.

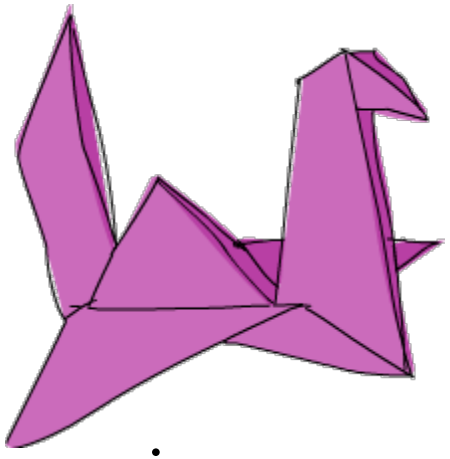
- foldr :: (a -> b -> b) -> b -> [a] -> b
- foldr f e [ ] = e
- foldr f e (x:xs) = f x (foldr f e xs)

Mientras que foldl comienza a aplicar la función desde el primer elemento y se define de la siguiente forma:

- foldl :: (b -> a -> -> b) -> [a] -> b
- folldl f e [ ] = e
- foldl f e (x:xs) = foldl f (f e a) xs

Con esto podemos observar que mientras una cambia la estructura de la lista la otra cambia el valor inicial, se comporta como un acumulador.

ORIGAMI



Los folds pueden usarse para definir **cualquier** función que siga el patrón que vimos anteriormente.

Se pueden generar funciones folds para **todos los tipos de datos** que haya sido definido recursivamente.

Esto es lo que los hace tan útiles y poderosos en programación funcional.

Existe un estilo de programación en el que todo es definido a partir de folds.

Llamado **programación origami**.

Es fácil de leer y explicar pero no taaan fácil de programar.

OTROS FOLDS DE HASKELL

Haskell implementa **foldl1** y **fodr1** que funcionan de la misma manera que **foldl** y **foldr** respectivamente, pero la diferencia es que no reciben un valor inicial si no que toman el primer (o último) elemento de la lista como valor inicial.

También tenemos **scanl** y **scanr** que hacen lo mismo que **foldl** y **foldr** respectivamente pero regresan una lista con cada uno de los valores del acumulador en cada iteración.

Existen **scanl1** y **scanr1** análogos a **foldl1** y **foldr1**.