Programación Declarativa 2020-2
Clases de tipos

Javier Enríquez Mendoza

Facultad de Ciencias UNAM

1 de mayo de 2020

Patrones

En Haskell existen funciones que definen comportamiento para mas de un tipo

Patrones

En Haskell existen funciones que definen comportamiento para mas de un tipo

Como `(==)`, `(<)`, `show`, etc.

Patrones

En Haskell existen funciones que definen comportamiento para mas de un tipo

Como `(==)`, `(<)`, `show`, etc.

En primera instancia podría parecer que se trata de funciones con tipos polimórficos

Patrones

En Haskell existen funciones que definen comportamiento para mas de un tipo

Como `(==)`, `(<)`, `show`, etc.

En primera instancia podría parecer que se trata de funciones con tipos polimórficos

```
(==) :: t -> t -> Bool
```

```
(<) :: t -> t -> Bool
```

```
show :: t -> String
```

Diferencia

Sin embargo hay una diferencia respecto a las funciones polimóficas que hemos visto hasta ahora

Diferencia

Sin embargo hay una diferencia respecto a las funciones polimóficas que hemos visto hasta ahora

Las funciones `(==)`, `(<)` y `show` estan sobrecargadas

Diferencia

Sin embargo hay una diferencia respecto a las funciones polimóficas que hemos visto hasta ahora

Las funciones `(==)`, `(<)` y `show` estan sobrecargadas

Esto significa que:

Diferencia

Sin embargo hay una diferencia respecto a las funciones polimóficas que hemos visto hasta ahora

Las funciones `(==)`, `(<)` y `show` estan sobrecargadas

Esto significa que:

Están definidas para mas de un tipo

Diferencia

Sin embargo hay una diferencia respecto a las funciones polimóficas que hemos visto hasta ahora

Las funciones `(==)`, `(<)` y `show` estan sobrecargadas

Esto significa que:

- Están definidas para mas de un tipo
- La definición de estas funciones depende el tipo que recibe

Ejemplo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

Ejemplo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

```
length :: [a] -> Int
legth = foldr (const (+1)) 0
```

Ejemplo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

```
length :: [a] -> Int
legth = foldr (const (+1)) 0
```

Está es una definición de una función polimórfica pues sin importar quien sea a, la función va a computar la longitud de la lista

Ejemplo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

```
length :: [a] -> Int
legth = foldr (const (+1)) 0
```

Está es una definición de una función polimórfica pues sin importar quien sea a, la función va a computar la longitud de la lista

En otras palabras `length` funciona para listas de cualquier tipo

Polimorfismo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

Polimorfismo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

```
length :: [a] -> Int
length = foldr (const (+1)) 0
```

Polimorfismo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

```
length :: [a] -> Int
length = foldr (const (+1)) 0
```

Está es una definición de una función polimórfica pues sin importar quien sea a, la función va a computar la longitud de la lista

Polimorfismo

Consideremos la siguiente definición de la función `length` que calcula la longitud de una lista

```
length :: [a] -> Int
length = foldr (const (+1)) 0
```

Está es una definición de una función polimórfica pues sin importar quien sea `a`, la función va a computar la longitud de la lista

En otras palabras `length` funciona para listas de cualquier tipo

Sobrecargas

Ahora consideremos el operador (==)

Sobrecargas

Ahora consideremos el operador (==)

La definición de igualdad para el tipo `Bool` es la siguiente:

Sobrecargas

Ahora consideremos el operador (==)

La definición de igualdad para el tipo `Bool` es la siguiente:

```
(==) :: Bool -> Bool -> Bool
True == True   = True
False == False = True
_ == _         = False
```

Sobrecargas

Ahora consideremos el operador (==)

La definición de igualdad para el tipo `Bool` es la siguiente:

```
(==) :: Bool -> Bool -> Bool
True == True   = True
False == False = True
_ == _         = False
```

La definición del mismo operador para el tipo `Nat` es:

Sobrecargas

Ahora consideremos el operador (==)

La definición de igualdad para el tipo `Bool` es la siguiente:

```
(==) :: Bool -> Bool -> Bool
True == True   = True
False == False = True
_ == _         = False
```

La definición del mismo operador para el tipo `Nat` es:

```
(==) :: Nat -> Nat -> Bool
0 == 0         = True
(S n) == (S m) = n == m
_ == _         = False
```

Sobrecargas

Es claro que se trata de definiciones diferentes para el mismo operador (==)

Sobrecargas

Es claro que se trata de definiciones diferentes para el mismo operador (==)

La definición depende del tipo

Sobrecargas

Es claro que se trata de definiciones diferentes para el mismo operador (==)

La definición depende del tipo

A esto se le llama sobrecarga

Sobrecargas

Es claro que se trata de definiciones diferentes para el mismo operador (==)

La definición depende del tipo

A esto se le llama sobrecarga

A la colección de tipos sobre los cuales se define una función sobrecargada, le llamamos Clase.

Sobrecargas

Es claro que se trata de definiciones diferentes para el mismo operador (==)

La definición depende del tipo

A esto se le llama sobrecarga

A la colección de tipos sobre los cuales se define una función sobrecargada, le llamamos Clase.

El conjunto de tipos para los cuales está definida la función (==), es la clase igualdad (`Eq`)

Definición de Clases

Nosotros podemos definir nuevas clases en Haskell

Definición de Clases

Nosotros podemos definir nuevas clases en Haskell

Para esto hay que especificar qué es lo que debe de cumplir el tipo para pertenecer a la clase

Definición de Clases

Nosotros podemos definir nuevas clases en Haskell

Para esto hay que especificar qué es lo que debe de cumplir el tipo para pertenecer a la clase

Por ejemplo para que un tipo pertenezca a la clase `Eq` se tiene que definir la función `(==)`, las funciones de una clase se llaman **métodos**

Definición de Clases

Nosotros podemos definir nuevas clases en Haskell

Para esto hay que especificar qué es lo que debe de cumplir el tipo para pertenecer a la clase

Por ejemplo para que un tipo pertenezca a la clase `(Eq)` se tiene que definir la función `(==)`, las funciones de una clase se llaman **métodos**

Para definir clases se usa la sentencia `class`

Definición de Clases

Nosotros podemos definir nuevas clases en Haskell

Para esto hay que especificar qué es lo que debe de cumplir el tipo para pertenecer a la clase

Por ejemplo para que un tipo pertenezca a la clase `(Eq)` se tiene que definir la función `(==)`, las funciones de una clase se llaman **métodos**

Para definir clases se usa la sentencia `class`

Por ejemplo, la definición de la clase `Eq` es la siguiente:

Definición de Clases

Nosotros podemos definir nuevas clases en Haskell

Para esto hay que especificar qué es lo que debe de cumplir el tipo para pertenecer a la clase

Por ejemplo para que un tipo pertenezca a la clase `(Eq)` se tiene que definir la función `(==)`, las funciones de una clase se llaman **métodos**

Para definir clases se usa la sentencia `class`

Por ejemplo, la definición de la clase `Eq` es la siguiente:

```
class Eq t where
  (==) :: t -> t -> Bool
```

Definición de Clases

Nosotros podemos definir nuevas clases en Haskell

Para esto hay que especificar qué es lo que debe de cumplir el tipo para pertenecer a la clase

Por ejemplo para que un tipo pertenezca a la clase `(Eq)` se tiene que definir la función `(==)`, las funciones de una clase se llaman **métodos**

Para definir clases se usa la sentencia `class`

Por ejemplo, la definición de la clase `Eq` es la siguiente:

```
class Eq t where
  (==) :: t -> t -> Bool
```

Los miembros de una clase **C** se llaman instancias de **C**

Definición de Clases

Cuando se define una clase se puede dar una definición por default de los métodos de está

Definición de Clases

Cuando se define una clase se puede dar una definición por default de los métodos de está

Esto es útil cuando queremos que el método de la clase se comporte igual para todas las instancias

Definición de Clases

Cuando se define una clase se puede dar una definición por default de los métodos de está

Esto es útil cuando queremos que el método de la clase se comporte igual para todas las instancias

Por ejemplo en la clase `Eq` está definido el método `(/=)` como sigue

Definición de Clases

Cuando se define una clase se puede dar una definición por default de los métodos de está

Esto es útil cuando queremos que el método de la clase se comporte igual para todas las instancias

Por ejemplo en la clase `Eq` está definido el método `(/=)` como sigue

```
class Eq t where
  (/=) :: t -> t -> Bool
  x /= y = not $ x == y
```

Definición de Clases

Cuando se define una clase se puede dar una definición por default de los métodos de está

Esto es útil cuando queremos que el método de la clase se comporte igual para todas las instancias

Por ejemplo en la clase `Eq` está definido el método `(/=)` como sigue

```
class Eq t where
  (/=) :: t -> t -> Bool
  x /= y = not $ x == y
```

En esta clase, la definición del método `(/=)` ya está dada y cuando se instancia la clase ya no se debe definir

Instancias de Clases

Si queremos que un tipo sea instancia de una clase, tenemos que definir los métodos descritos en ella

Instancias de Clases

Si queremos que un tipo sea instancia de una clase, tenemos que definir los métodos descritos en ella

Para definir una instancia de una clase se utiliza la sentencia `instance`

Instancias de Clases

Si queremos que un tipo sea instancia de una clase, tenemos que definir los métodos descritos en ella

Para definir una instancia de una clase se utiliza la sentencia `instance`

Por ejemplo, la instancia de `Eq` para listas es la siguiente

Instancias de Clases

Si queremos que un tipo sea instancia de una clase, tenemos que definir los métodos descritos en ella

Para definir una instancia de una clase se utiliza la sentencia `instance`

Por ejemplo, la instancia de `Eq` para listas es la siguiente

```
instance Eq [] where
 [] == []           = True
 (x:xs) == (y:ys) = (x == y) && (xs == ys)
 _ == _             = False
```

Restricciones

Para poder usar los métodos de una clase en las definiciones de funciones se tiene que garantizar que el tipo tenga una instancia definida de la clase

Restricciones

Para poder usar los métodos de una clase en las definiciones de funciones se tiene que garantizar que el tipo tenga una instancia definida de la clase

Esto se hace usando restricciones sobre los tipos

Restricciones

Para poder usar los métodos de una clase en las definiciones de funciones se tiene que garantizar que el tipo tenga una instancia definida de la clase

Esto se hace usando restricciones sobre los tipos

Por ejemplo, definimos la función polimórfica que verifica si todos los elementos de una lista son iguales

Restricciones

Para poder usar los métodos de una clase en las definiciones de funciones se tiene que garantizar que el tipo tenga una instancia definida de la clase

Esto se hace usando restricciones sobre los tipos

Por ejemplo, definimos la función polimórfica que verifica si todos los elementos de una lista son iguales

```
same :: (Eq a) => [a] -> Bool
same [] = True
same (x:xs) = foldr ((&&).(x==)) True xs
```

Restricciones

Para poder usar los métodos de una clase en las definiciones de funciones se tiene que garantizar que el tipo tenga una instancia definida de la clase

Esto se hace usando restricciones sobre los tipos

Por ejemplo, definimos la función polimórfica que verifica si todos los elementos de una lista son iguales

```
same :: (Eq a) => [a] -> Bool
same [] = True
same (x:xs) = foldr ((&&).(x==)) True xs
```

En donde se pone la restricción sobre el tipo a de instanciar a la clase Eq para usar el método (==)

Extensiones de Clases

Las clases pueden extenderse

Extensiones de Clases

Las clases pueden extenderse

Esto se declara en la definición de la clase

Extensiones de Clases

Las clases pueden extenderse

Esto se declara en la definición de la clase

Por ejemplo, la clase `Ord` extiende a `Eq`

Extensiones de Clases

Las clases pueden extenderse

Esto se declara en la definición de la clase

Por ejemplo, la clase `Ord` extiende a `Eq`

```
class (Eq t) => Ord t where
  compare              :: t -> t -> Ordering
  (<),(>),(<=),(>=)    :: t -> t -> Bool
  min,max              :: t -> t -> t
```

Extensiones de Clases

Las clases pueden extenderse

Esto se declara en la definición de la clase

Por ejemplo, la clase `Ord` extiende a `Eq`

```
class (Eq t) => Ord t where
  compare              :: t -> t -> Ordering
  (<),(>),(<=),(>=)    :: t -> t -> Bool
  min,max              :: t -> t -> t
```

Lo que significa que para crear una instancia de la clase `Ord` también se tiene que instanciar a la clase `Eq`

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- Eq

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- `Eq`
- `Ord`

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- `Eq`
- `Ord`
- `Show`

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- `Eq`
- `Ord`
- `Show`
- `Read`

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- `Eq`
- `Ord`
- `Show`
- `Read`
- `Num`

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- `Eq`
- `Ord`
- `Show`
- `Read`
- `Num`
- `Fractional`

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- `Eq`
- `Ord`
- `Show`
- `Read`
- `Num`
- `Fractional`
- `Floating`

Clases básicas

Haskell tiene ya predefinidas algunas clases como son:

- `Eq`
- `Ord`
- `Show`
- `Read`
- `Num`
- `Fractional`
- `Floating`
- `Integral`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`
- `Enum`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`
- `Enum`
- `Foldable`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`
- `Enum`
- `Foldable`
- `Functor`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`
- `Enum`
- `Foldable`
- `Functor`
- `Applicative`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`
- `Enum`
- `Foldable`
- `Functor`
- `Applicative`
- `Monoid`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`
- `Enum`
- `Foldable`
- `Functor`
- `Applicative`
- `Monoid`
- `Monad`

Otras clases

Otras clases mas complejas definidas en el Preludio son:

- `Real`
- `Enum`
- `Foldable`
- `Functor`
- `Applicative`
- `Monoid`
- `Monad`
- `Bounded`
- `Traversable`


I KNOW
HASKELL
memegenerator.net