El Sistema de Clases de Haskell

- Habitualmente, un tipo de datos Γ es visto como una tupla $\Gamma = (T , \{f , g , ...\})$ donde T es una colección de datos y $\{f , g , ...\}$ es un conjunto de funciones aplicables a T.

- En nuestro lenguaje funcional f es aplicable a T si aparece T en el tipo de f.

Es de interés sobrecargar funciones aplicables a distintos tipos

- Por ejemplo, Int, Integer, Float, Double, etc. , comparten una serie de funciones aplicables a estos tipos, como (+), (-), (*), . . . ; en este caso tenemos una clase de tipos, en el sentido de que cada elemento de la clase (o instancia) dispone de un conjunto de valores particulares (o tipo de datos) y una forma también particular de calculo de cada función de la clase (es decir de cada función sobrecargada).

Eq

- Es una clase de tipos, cuyas instancias comparten las funcione s sobrecargadas (==) y (/=).
- class Eq a where
- (==) : a -> a -> Bool
- (/=) : a -> a -> Bool
- --minimo implementar : (==) o bien (/=)
- x==y = not (x/=)
- x/=y = not (x==y)

Eq

- A Eq a se le llama un contexto, a representa un tipo genérico instancia de la clase, y su uso es para poder parametrizar el tipo de la función sobrecargada :
- a -> a -> Bool

- Cada instancia podrá definir una forma particular de calculo de la función; ello se consigue a través de declaraciones de instancia :

Por ejemplo

- instance Eq Int where
- 0 == 0 = True
- . . .
- instance Eq Float where
- 0.0 == 0.0 = True
- . . .

En general podríamos tener un contexto con varias restricciones

- f :: (A a , B a, C a) => [a] -> Bool

- Que indica que a debe ser una instancia de las clases A, B y C.

- Es importante reseñar que cada función miembro de una clase implícitamente en su tipo el contexto apropiado.

Las relaciones entre contextos dan lugar a una jerarquía de clases

- class A a where
- f :: a -> Bool
- class B a where
- g :: A a => a -> a -> a
- g x y | f x = y
- | otherwise = x
- Siendo el tipo de la función g :
- g :: (A a , B a) => a -> a -> a

Podemos restringir el tipo a de la clase B en el contexto de A a

- class A a => B a where
- g :: a -> a -> a
- g x y | f x = y
- | otherwise = x

- Siendo ahora el tipo de g :
- g :: B a => a -> a -> a
- En este caso se dice que B es una subclase de A.

Ord es una subclase de Eq

- class (Eq a) => Ord a where
- (<),(<=),(>=),(>) :: a -> a ->Bool
- max , min :: a -> a -> a

Herencia

- ***Ord*** es un **subclase** de ***Eq*.**
- ***Eq*** es una **superclase** de ***Ord***

- La **subclase hereda las operaciones de la superclase**
- Cualquier tipo que sea instancia de ***Ord***, también ha de serlo de ***Eq***

Herencia

- Un beneficio de la herencia es que las restricciones de tipo son más cortas.

- Una función que use operaciones tanto de la clase ***Eq*** como de la clase ***Ord*** puede limitarse a usar la restricción ***Ord*** .

Herencia múltiple

- **Un clase puede tener más de una superclase.**

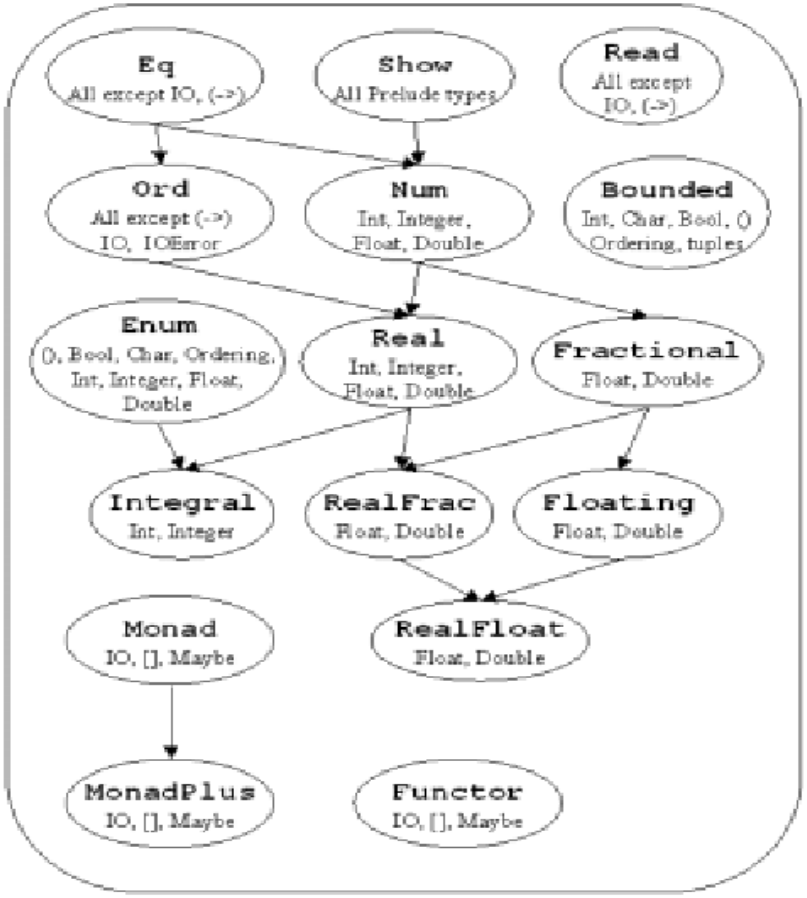
- Un ejemplo de ello lo constituye la declaración de la clase ***Num***, que recolecta a los tipos numéricos predefinidos en Haskell. Cada tipo numérico es instancia de varias clases de tipos organizadas en una jerarquía

Herencia múltiple

- La razón para ello es que hay ciertas operaciones como (+) que tienen sentido para todos los tipos numéricos, mientras que otras operaciones como las trigonométricas serian sólo aplicables a tipos que representan a los números reales

- En la parte más alta de la jerarquía está la clase ***Num*** , que contiene a todas las operaciones válidas para cualquier tipo numérico .

Jerarquía de clases de PRELUDE



Herencia múltiple

- Esta definición de clase supone que todos los tipos numéricos, tienen definidas, además de las funciones que aparecen en la signatura, todas las funciones de la clase Eq, y de la clase Show que agrupa a todos los tipos cuyos valores pueden convertirse al tipo String, es decir, que pueden convertirse en cadenas.

Herencia múltiple

- class (Eq a , Show a) => Num a where
- (+) , (-) , (*) :: a -> a -> a
- negate :: a -> a
- abs , signum :: a -> a
- fromInteger :: Integer ->a
- fromInt :: Int -> a
- x-y = x + negate y
- fromInt = fromInteger
- negate x = 0 - x

Herencia múltiple

- Observe que el operador de la división (/) no forma parte de la clase de tipos *Num*.

- `Hugs> :t (/)`
- `(/) :: Fractional a => a -> a -> a`

- La función **fromtInt** convierte el entero, del tipo Int, en uno de tipo *Num a =>a* , análogamente para **fromInteger**.

Herencia múltiple

- ¿Cómo es posible escribir un número, como 45, tanto en un contexto que requiera el uso de una expresión de tipo Int como en otro que requiera el tipo Float?

- La respuesta es que:
- `Hugs> :t 45`
- `45 :: Num a => a`
- para algún tipo numérico *a* que será determinado por el contexto en que aparezca.

Herencia múltiple

- Esto se consigue en Haskell haciendo que 45 sea una abreviatura de *fromtInteger 45, que es del tipo Num a => a*
- En el prelude estándar se define sólo los tipos numéricos básicos: Int, Integer, Float y Double.
- Otros tipos numéricos como los racionales o los complejos se definen en bibliotecas.

Herencia múltiple

- Haskell proporciona un mecanismo mediante el cual se puede crear automáticamente declaraciones de instancia para tipos de una clase.
- Esto se consigue incluyendo una cláusula ***deriving*** en la declaración del tipo de datos: *Por ejemplo :*
- *data MiBool = Falso | Cierto deriving Eq*

Herencia múltiple

- De manera análoga, podriamos derivar una instancia de *Ord* para *MiBool*

data MiBool = Falso | Cierto deriving (Eq,Ord)

Tipos Numéricos

- En matemáticas se trabaja con diversos sistemas numéricos: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{C}$. Que a la hora de trasladarlos a los lenguajes de programación, se presentan serios problemas.

- Un número puede necesitar una cantidad arbitraria de memoria para ser representado, incluso una cantidad infinita, como ocurriría con el número ¶

Tipos Numéricos

- La mayoría de lenguajes no lidian muy bien con estos problemas. Haskell lo hace algo mejor que la mayoría, puesto que proporciona representaciones exactas para enteros (*Integer*) y para racionales (Rational) .
- Pero tanto en Haskell como en la mayoría de los lenguajes de programación, no hay una representación exacta para los números reales, que se aproximan usando números en punto flotante con precisión simple o precisión doble (Float o Double en Haskell).

Tipos Numéricos

- La programación con números en punto flotante de aplicaciones sofisticadas a menudo precisa de un profundo conocimiento del análisis numérico para ser capaces de producir algoritmos adecuados y escribir programas correctos.
- La representación de un valor de tipo ***Integer*** en Haskell es algo que no nos va a preocupar, sin embargo, se deberá contar con un mecanismo dinámico de almacenamiento, que hará que los programas que usan ***Integer*** sean, más lentos.

Tipos Numéricos

- Haskell proporciona otro tipo de datos entero, ***Int*** , con valores mínimo y máximo.
- Esto produce errores de desbordamiento por defecto o por exceso (underflow y overflow) cuando un valor de tipo ***Int*** excede el valor mínimo o máximo.

- Por lo que debe quedarnos claro que tenemos que usar ***Int*** con cuidado, y sólo cuando esta elección sea razonable.

Tipos Numéricos

- class (Num a) => Fractional a where
- (/) :: a -> a -> a
- recip :: a -> a
- fromRational :: Rational -> a
- fromDouble :: Double -> a
- recip x = 1 / x
- fromDouble = fromRational . toRational
- x / y = x * recip y

Tipos Numéricos

- pi = 4 * atan 1
  - x ** y = exp (log x * y)
  - logBase x y = log y / log x
  - sqrt x = x ** 0.5
  - tan x = sin x / cos x
  - sinh x = (exp x - exp (-x)) / 2
  - cosh x = (exp x + exp (-x)) / 2
  - tanh x = sinh x / cosh x
  - asinh x = log (x + sqrt (x*x + 1))
  - acosh x = log (x + sqrt (x*x - 1))
  - atanh x = (log (1 + x) - log (1 - x)) / 2

Tipos Numéricos

- class (Fractional a) => Floating a where
- pi :: a
- exp, log, sqrt :: a -> a
- (**), logBase :: a -> a -> a
- sin, cos, tan :: a -> a
- asin, acos, atan :: a -> a
- sinh, cosh, tanh :: a -> a
- asinh, acosh, atanh :: a -> a
-