Programación Funcional

Curso 2019-20

DEFINICIÓN DE TIPOS Y CLASES

Tipos definidos por el usuario I

Alias de tipo (o tipos sinónimos) type

- type String = [Char] está en `Prelude`
- type Coordenada = Float
  type Punto = (Coordenada,Coordenada)
  distancia:: Punto -> Punto -> Float
  distancia (x,y) (x',y') = sqrt ((x-x')^2 + (y-y')^2)
-
```
> :t distancia
Punto -> Punto -> Float
```
```
> :t (2,3)::Punto
Punto
```
```
> ((2,3)::Punto) == ((2,3)::(Float,Float))
True
```
- No pueden ser recursivos
  type T = (Int,[T]) Error!
- Pero sí pueden ser paramétricos
  type Terna a = (a,a,a)

Tipos definidos por el usuario II

Tipos de datos estructurados data

Definidos por constructoras de datos

Un caso particular: tipos enumerados

data $T = C_1 \mid \ldots \mid C_n$

data DiaSemana = L | M | X | J | V | S | D
ayer:: DiaSemana → DiaSemana
ayer L = D
ayer M = L
. . .
ayer D = S

data Palo = Oros | Copas | Espadas | Bastos

Algunos tipos predefinidos se ajustan a este modelo

data Bool = True | False
data Char = 'A' | . . . 'Z' | 'a' | . . .|'z'
data Int = -2147483648 | . . . -1 | 0 | 1 | . . . | 2147483647

Forma general de un tipo de datos construido

data $T\ \alpha_1 \dots \alpha_n = C_1\ \tau_{11} \dots \tau_{1k_1} \mid \dots \mid C_m\ \tau_{m1} \dots \tau_{mk_m}$

- T es un identificador (constructora de tipos): aumentan la sintaxis de los tipos simples
- Cada C_i es un identificador (constructora de datos para el tipo T)
- Las constructoras de datos han de ser distintas para cada tipo.
- $\alpha_1 \dots \alpha_n$ son variables de tipo (parámetros formales del tipo T). En el lado derecho de la definición *data* deben aparecer todas ellas (y ninguna otra)
- Puede ser $n = 0$ (T es un tipo no parametrizado o monomórfico) o $n > 0$ (T es un tipo parametrizado o polimórfico)
- Para cada $i = 1, \dots, m$ puede ser $k_i = 0$ (C_i es una constante de datos) o $k_i > 0$ (C_i es una constructora de datos no constante)

Tipos estructurados monomórficos

Un tipo monomórfico: cartas de la baraja

- `data Carta = Carta Int Palo`
- Esta definición determina el tipo de las constructoras de datos: *Carta::Int* $\rightarrow$ *Palo* $\rightarrow$ *Carta*
- Algunos valores del tipo *Carta*: *Carta 1 Oros* , *Carta 22 Copas*

O también

```
data Valor = As | Dos | Tres | ...| Sota | Caballo | Rey
data Carta = Carta Valor Palo
```

Carta::Valor $\rightarrow$ *Palo* $\rightarrow$ *Carta*

Los valores ahora son más expresivos:

Carta As Oros , *Carta Caballo Copas*

Tipos estructurados recursivos y/o polimórficos

Números naturales al estilo Peano

- `data Nat = Cero | Suc Nat`
- Esta definición determina el tipo de las constructoras de datos:
  Cero::Nat   *Suc::Nat* $\rightarrow$ *Nat*
- Algunos valores del tipo Nat: *Cero*, *Suc Cero*, *Suc (Suc Cero)*

Listas polimórficas

- `data List a = Nil | Cons a (List a)`
- *Nil::List a*   *Cons:: a* $\rightarrow$ *List a* $\rightarrow$ *List a* $\forall a$ implícito
- Esta definición es isomorfa a la de las listas predefinidas
- *List Int* $\ni$ *Cons 5 (Cons 2 (Cons 1 Nil))* $\simeq [5, 2, 1]$
  List (List Bool) $\ni$ *Cons Nil (Cons (Cons True Nil) Nil)*
  $\simeq [[], [True]]$

Árboles binarios

- Con información solo en las hojas
  data Arbol a = Hoja a | Nodo (Arbol a) (Arbol a)
- Con información en hojas y nodos
  data Arbol' a b = Hoja' a | Nodo' b (Arbol' a b) (Arbol' a b)

¿Cómo definirías un tipo para representar árboles con un número indeterminado de hijos?

Otros tipos estructurados predefinidos en Prelude

Tipos *Maybe* y *Either*

- data Maybe a = Nothing | Just a
- data Either a b = Left a | Right b

Clases de tipos: polimorfismo *ad-hoc*

Polimorfismo paramétrico (sistema de *Hindley-Milner*)

length :: $\forall a.[a] \rightarrow Int$

- *length* se puede aplicar a listas de *cualquier* tipo
- La definición de *length* es uniforme para todos los tipos

¿Qué tipo queremos que tengan $+$, $*$, $==, \ldots$?

Polimorfismo *ad-hoc*

$$(+) \quad :: \forall a \in Num.a \rightarrow a \rightarrow a$$

$$(==) \quad :: \forall a \in Eq.a \rightarrow a \rightarrow Bool$$

```
(+)::Num a ⇒ a → a → a
(==)::Eq a ⇒ a → a → Bool
```

Tipos cualificados

- *Num* es una *clase* de tipos
- La definición de $+$ puede ser distinta para cada tipo de *Num*

Las clases de tipos fueron propuestas por *Wadler*

¿Qué es una clase

Clase de tipo = colección de tipos + métodos de clase

- Además de las clases predefinidas, el usuario puede definir sus propias clases.
- Al definir una clase $\mathcal{C}$ se introducen sus métodos, pero no qué tipos están en $\mathcal{C}$.
- Los tipos de $\mathcal{C}$ se van introduciendo por declaraciones de instancia de $\mathcal{C}$.
- La definición de una clase $\mathcal{C}$
  - Debe incluir la declaración de sus métodos
  - Puede incluir la definición por defecto de sus métodos
  - Al declarar un tipo como instancia de $\mathcal{C}$ se puede cambiar la definición por defecto de los métodos
- Una vez definida $\mathcal{C}$ se pueden definir funciones con tipos cualificados por $\mathcal{C}$

Definición de clases de tipos

Ejemplo: la clase *Eq*

```
class Eq a where
  (==),(/=) ::  a -> a -> Bool -- Métodos de la clase Eq
  x == y = not (x/=y)    -- Definiciones por defecto de los métodos
  x /= y = not (x==y)
```

- Los tipos de los métodos quedan cualificados:
  `(==),(/=)::  Eq a => a -> a -> Bool`
- Las definiciones por defecto de los métodos son opcionales
- En una instancia de `Eq` bastará redefinir == o bien /=
  (Tanto $\{==\}$ como $\{/=\}$ son *conjuntos minimales suficientes* de métodos de Eq)

☞ La clase `Eq` ya está definida en `Prelude`

Ahora podemos definir funciones que usen métodos de *Eq*

- ```
  elem x [] = False
  elem x (y:ys) = if x==y then True else elem x ys
  ```

  ¿Tipo de *elem*?

  ```
  elem::  Eq a => a -> [a] -> Bool
  ```

- No hace falta tener definidas instancias de la clase para definir funciones que usen los métodos de la clase.
  ```

Declaración de instancias de clase

Declaración de *Bool* como instancia de *Eq*

```
data Bool = False | True

instance Eq Bool where
  False == False = True
  False == True = False
  True == False = False
  True == True = True
```

Hemos optado por redefinir ==

Al declarar un tipo T como instancia de una clase $\mathcal{C}$, todos los métodos de $\mathcal{C}$ deben quedar definidos para T, bien por su definición por defecto, si la tienen, o por la (re)definición del usuario

Otras declaraciones (condicionadas) de instancia de *Eq*

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

☞ `[a]` está en `Eq` si `a` está en `Eq`

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x,y) == (x',y') = x==x' && y==y'
```

☞ `(a,b)` está en `Eq` si `a` y `b` están en `Eq`

Estas declaraciones concretas ya están en el `Prelude`

Declaraciones automáticas de instancia de clase

Usar *deriving* nos ahorra trabajo

```
data Bool = True | False deriving Eq

data Arbol a b = Hoja a | Nodo b (Arbol a b) (Arbol a b)
    deriving Eq
```

- Al usar *deriving Eq* al definir un tipo construido T se genera automáticamente la definición de $==$ como igualdad estructural (o *sintáctica*) de los valores de T
- Se puede usar *deriving* para las clases *Eq*,*Ord*,*Enum*,*Show*, entre otras
- No se puede usar *deriving* para clases definidas por el usuario

deriving no es siempre adecuado o posible

```
type Numerador = Integer
type Denominador = Integer
infixl 7 :/
data Fraccion = Numerador :/ Denominador
instance Eq Fraccion where
  a:/b== c:/d = a*d == b*c
instance Num Fraccion where
  a:/b + c:/d = (a*d+b*c) :/ b*d
  a:/b * c:/d = (a*c) :/ b*d
  --- Más operaciones de la clase Num ---
```

Alias de tipo

Constructora de datos infija

$Integer \in Eq$

Clases de tipos: subclases

La clase *Ord* como subclase de *Eq*

- En la clase *Ord* queremos tener a los tipos cuyos valores pueden ser comparados por $<, >, \leq, \geq$.
- Para que $x \leq y$ tenga sentido, ha de tenerlo $x == y$.
- Así pues, todo tipo de *Ord* debe estar también en *Eq*. Podemos decir que *Ord* es subclase de *Eq*.
- Declaramos `class Eq a => Ord a where`

```
-- métodos de Ord --
```

- La comprobación de tipos usa que si $a \in Ord$ entonces forzosamente $a \in Eq$

Pero se escribe `Eq a => Ord a`

- Pero declarar un tipo como instancia de *Ord* **no** implica tenerlo declarado como instancia de *Eq*. Hay que hacerlo explícitamente.

```
data Ordering = LT | EQ | GT
class Eq a => Ord a where
  infix 4 <,>,<=,>=
  compare ::  a -> a -> Ordering
  (<), (<=), (>=), (>) ::  a -> a -> Bool
  max, min ::  a -> a -> a
  -- Conjunto minimal suficiente:  (<=) o compare
  compare x y | x==y = EQ
              | x<=y = LT
              | otherwise = GT
  x <= y = compare x y /= GT
  x < y = compare x y == LT
  ...
```

Ya incluido en el `Prelude`

Bool como instancia de Ord

```
-- Tenemos data Bool = False | True

instance Ord Bool where
  True <= False = False
  _    <= _     = True
```

O bien: `data Bool = False | True deriving (Eq,Ord)`

Listas polimórficas como instancia de Ord

```
-- Tenemos data [a] = [] | a:[a]

instance Ord a => Ord [a] where
  [] <= [] = True
  [] <= (_:_) = True
  (_:_) <= [] = False
  (x:xs) <= (y:ys) = x < y || x == y && xs <= ys
```

Aquí también valdría *deriving*

Orden inducido por *deriving Ord*

Supongamos `data T = C1 ...| C2 ...| Cn ...deriving Ord` , y supongamos dos valores `x` e `y` de `T`

Para evaluar `x <= y` :

1. Se comparan las constructoras más externas de `x` e `y` , que están ordenadas según el orden de aparición en la def. de `T`
2. Si son iguales, se comparan lexicográficamente las tuplas de argumentos. Es decir, se comparan primero los primeros argumentos; si son iguales, se pasa al segundo, etc.

El orden inducido por *deriving* no siempre es el adecuado

```
instance Ord Fraccion where
  (a :/ b) <= (c :/ d) = a*d <= b*c
```

¿Qué orden resultaría con `data Fracción = ...deriving Ord` ?

Jerarquía de clases predefinidas en Haskell

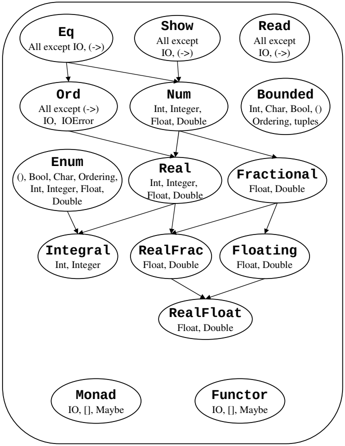

Figure 6.1: Standard Haskell Classes

Extraído del Haskell2010 report

Ambigüedad de tipos

- Es un problema específico del polimorfismo de clases
- Se presenta cuando al evaluar una expresión `e` cuyo tipo incluye una restricción de clase `C a => ...` el análisis de tipos no tiene información suficiente para saber qué instancias particulares deben usarse para evaluar los métodos de clase que puedan intervenir en `e`. En ese caso el sistema nos da un error de *ambigüedad de tipos*, que no indica que la expresión esté mal tipada, sino que se necesita dar información más explícita.

Ejemplo

```
>:t toEnum 0
Enum a => a
```

```
> toEnum 0
?
```

`toEnum 0` está bien tipada: dado cualquier tipo `a` de la clase `Enum`, `toEnum 0` nos da un valor del tipo `a`. Por ejemplo `toEnum 0` nos da `0` en `Int`, `False` en `Bool`, ... Pero por sí sola `toEnum 0` no tiene información suficiente para saber cómo evaluarla y por defecto devuelve (). Sin embargo:

- `(toEnum 0)::Int` dará `0`
- `(toEnum 0)::Bool` dará `False`
- ¿Qué da `head [toEnum 0]`?
- ¿Qué da `head [toEnum 0,1]`? ¿Por qué?

`()` $\leadsto$ tipo con un solo valor, que es también `()`