Declaraciones de tipos y clases

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 9: Declaraciones de tipos y clases

Declaraciones de tipos como sinónimos

- Se puede definir un nuevo nombre para un tipo existente mediante una declaración de tipo.
- Ejemplo: Las cadenas son listas de caracteres.

Prelude

```
type String = [Char]
```

- El nombre del tipo tiene que empezar por mayúscula.

Declaraciones de tipos nuevos

- Las declaraciones de tipos pueden usarse para facilitar la lectura de tipos. Por ejemplo,
  - Las posiciones son pares de enteros.

```
type Pos = (Int,Int)
```

  - `origen` es la posición (0,0).

```
origen :: Pos
origen = (0,0)
```

  - `(izquierda p)` es la posición a la izquierda de la posición p. Por ejemplo,

```
izquierda (3,5)  ⇝  (2,5)
```

```
izquierda :: Pos -> Pos
izquierda (x,y) = (x-1,y)
```

Declaraciones de tipos parametrizadas

- Las declaraciones de tipos pueden tener parámetros. Por ejemplo,
  - `Par a` es el tipo de pares de elementos de tipo `a`

```
type Par a = (a,a)
```

  - `(multiplica p)` es el producto del par de enteros `p`. Por ejemplo,

    `multiplica (2,5)` ⇝ `10`

```
multiplica :: Par Int -> Int
multiplica (x,y) = x*y
```

  - `(copia x)` es el par formado con dos copias de `x`. Por ejemplo,

    `copia 5` ⇝ `(5,5)`

```
copia :: a -> Par a
copia x = (x,x)
```

Declaraciones anidadas de tipos

- Las declaraciones de tipos pueden anidarse. Por ejemplo,
  - Las posiciones son pares de enteros.

```
type Pos = (Int,Int)
```

  - Los movimientos son funciones que va de una posición a otra.

```
type Movimiento = Pos -> Pos
```

- Las declaraciones de tipo no pueden ser recursivas. Por ejemplo, el siguiente código es erróneo.

```
type Arbol = (Int,[Arbol])
```

Al intentar cargarlo da el mensaje de error

```
Cycle in type synonym declarations
```

Definición de tipos con data

- En Haskell pueden definirse nuevos tipos mediante `data`.
- El tipo de los booleanos está formado por dos valores para representar lo falso y lo verdadero.

Prelude
```
data Bool = False | True
```

- El símbolo `|` se lee como “o”.
- Los valores `False` y `True` se llaman los constructores del tipo `Bool`.
- Los nombres de los constructores tienen que empezar por mayúscula.

Uso de los valores de los tipos definidos

- Los valores de los tipos definidos pueden usarse como los de los predefinidos.
- Definición del tipo de movimientos:

```
data Mov = Izquierda | Derecha | Arriba | Abajo
```

- Uso como argumento: `(movimiento m p)` es la posición obtenida aplicando el movimiento `m` a la posición `p`. Por ejemplo,

  `movimiento Arriba (2,5)` $\rightsquigarrow$ `(2,6)`

```
movimiento :: Mov -> Pos -> Pos
movimiento Izquierda (x,y) = (x-1,y)
movimiento Derecha   (x,y) = (x+1,y)
movimiento Arriba    (x,y) = (x,y+1)
movimiento Abajo     (x,y) = (x,y-1)
```

Uso de los valores de los tipos definidos

- Uso en listas: `(movimientos ms p)` es la posición obtenida aplicando la lista de movimientos `ms` a la posición `p`. Por ejemplo,
  ```
  movimientos [Arriba, Izquierda] (2,5)  ⇝  (1,6)
  ```

```
movimientos :: [Mov] -> Pos -> Pos
movimientos []     p = p
movimientos (m:ms) p = movimientos ms (movimiento m p)
```

- Uso como valor: `(opuesto m)` es el movimiento opuesto de `m`.
  ```
  movimiento (opuesto Arriba) (2,5)  ⇝ (2,4)
  ```

```
opuesto :: Mov -> Mov
opuesto Izquierda = Derecha
opuesto Derecha   = Izquierda
opuesto Arriba    = Abajo
opuesto Abajo     = Arriba
```

Uso de los valores de los tipos definidos

- Uso en listas: (movimientos ms p) es la posición obtenida aplicando la lista de movimientos ms a la posición p. Por ejemplo,

  `movimientos [Arriba, Izquierda] (2,5)  ⇝  (1,6)`

```
movimientos :: [Mov] -> Pos -> Pos
movimientos []     p = p
movimientos (m:ms) p = movimientos ms (movimiento m p)
```

- Uso como valor: (opuesto m) es el movimiento opuesto de m.

  `movimiento (opuesto Arriba) (2,5)  ⇝ (2,4)`

```
opuesto :: Mov -> Mov
opuesto Izquierda = Derecha
opuesto Derecha   = Izquierda
opuesto Arriba    = Abajo
opuesto Abajo     = Arriba
```

Uso de los valores de los tipos definidos

- Uso en listas: `(movimientos ms p)` es la posición obtenida aplicando la lista de movimientos `ms` a la posición `p`. Por ejemplo,

```
movimientos [Arriba, Izquierda] (2,5)  ⇝  (1,6)
```

```
movimientos :: [Mov] -> Pos -> Pos
movimientos []     p = p
movimientos (m:ms) p = movimientos ms (movimiento m p)
```

- Uso como valor: `(opuesto m)` es el movimiento opuesto de `m`.

```
movimiento (opuesto Arriba) (2,5)  ⇝ (2,4)
```

```
opuesto :: Mov -> Mov
opuesto Izquierda = Derecha
opuesto Derecha   = Izquierda
opuesto Arriba    = Abajo
opuesto Abajo     = Arriba
```

Definición de tipo con constructores con parámetros

- Los constructores en las definiciones de tipos pueden tener parámetros.
- Ejemplo de definición

```haskell
data Figura = Circulo Float | Rect Float Float
```

- Tipos de los constructores:

```
*Main> :type Circulo
Circulo :: Float -> Figura
*Main> :type Rect
Rect :: Float -> Float -> Figura
```

- Uso del tipo como valor: `(cuadrado n)` es el cuadrado de lado `n`.

```haskell
cuadrado :: Float -> Figura
cuadrado n = Rect n n
```

Definición de tipo con constructores con parámetros

- Uso del tipo como argumento: `(area f)` es el área de la figura `f`. Por ejemplo,

```
area (Circulo 1)    ⇝  3.1415927
area (Circulo 2)    ⇝  12.566371
area (Rect 2 5)     ⇝  10.0
area (cuadrado 3)   ⇝  9.0
```

```
area :: Figura -> Float
area (Circulo r) = pi*r^2
area (Rect x y)  = x*y
```

Definición de tipos con parámetros

- Los tipos definidos pueden tener parámetros.
- Ejemplo de tipo con parámetro

Prelude
```
data Maybe a = Nothing | Just a
```

- `(divisionSegura m n)` es la división de `m` entre `n` si `n` no es cero y nada en caso contrario. Por ejemplo,

```
divisionSegura 6 3  ~>  Just 2
divisionSegura 6 0  ~>  Nothing
```

```
divisionSegura :: Int -> Int -> Maybe Int
divisionSegura _ 0 = Nothing
divisionSegura m n = Just (m `div` n)
```

Definición de tipos con parámetros

- (headSegura xs) es la cabeza de xs si xs es no vacía y nada en caso contrario. Por ejemplo,

```
headSegura [2,3,5]  ⇝  Just 2
headSegura []       ⇝  Nothing
```

```haskell
headSegura :: [a] -> Maybe a
headSegura [] = Nothing
headSegura xs = Just (head xs)
```

Definición de tipos recursivos: Los naturales

- Los tipos definidos con `data` pueden ser recursivos.
- Los naturales se construyen con el cero y la función sucesor.

```
data Nat = Cero | Suc Nat
           deriving Show
```

- Tipos de los constructores:

```
*Main> :type Cero
Cero :: Nat
*Main> :type Suc
Suc :: Nat -> Nat
```

- Ejemplos de naturales:

```
Cero
Suc Cero
Suc (Suc Cero)
Suc (Suc (Suc Cero))
```

Definiciones con tipos recursivos

- (nat2int n) es el número entero correspondiente al número natural n. Por ejemplo,

  ```
  nat2int (Suc (Suc (Suc Cero)))  ⇝  3
  ```

  ```haskell
  nat2int :: Nat -> Int
  nat2int Cero     = 0
  nat2int (Suc n) = 1 + nat2int n
  ```

- (int2nat n) es el número natural correspondiente al número entero n. Por ejemplo,

  ```
  int2nat 3  ⇝  Suc (Suc (Suc Cero))
  ```

  ```haskell
  int2nat :: Int -> Nat
  int2nat 0 = Cero
  int2nat n = Suc (int2nat (n-1))
  ```

Definiciones con tipos recursivos

- (nat2int n) es el número entero correspondiente al número natural n. Por ejemplo,

```
nat2int (Suc (Suc (Suc Cero)))  ⇝  3
```

```
nat2int :: Nat -> Int
nat2int Cero    = 0
nat2int (Suc n) = 1 + nat2int n
```

- (int2nat n) es el número natural correspondiente al número entero n. Por ejemplo,

```
int2nat 3  ⇝  Suc (Suc (Suc Cero))
```

```
int2nat :: Int -> Nat
int2nat 0 = Cero
int2nat n = Suc (int2nat (n-1))
```

Definiciones con tipos recursivos

- `(nat2int n)` es el número entero correspondiente al número natural `n`. Por ejemplo,

  ```
  nat2int (Suc (Suc (Suc Cero)))  ⇝  3
  ```

  ```haskell
  nat2int :: Nat -> Int
  nat2int Cero    = 0
  nat2int (Suc n) = 1 + nat2int n
  ```

- `(int2nat n)` es el número natural correspondiente al número entero `n`. Por ejemplo,

  ```
  int2nat 3  ⇝  Suc (Suc (Suc Cero))
  ```

  ```haskell
  int2nat :: Int -> Nat
  int2nat 0 = Cero
  int2nat n = Suc (int2nat (n-1))
  ```

Definiciones con tipos recursivos

- `(suma m n)` es la suma de los número naturales `m` y `n`. Por ejemplo,

```
*Main> suma (Suc (Suc Cero)) (Suc Cero)
Suc (Suc (Suc Cero))
```

```
suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)
```

- Ejemplo de cálculo:

```
  suma (Suc (Suc Cero)) (Suc Cero)
= Suc (suma (Suc Cero) (Suc Cero))
= Suc (Suc (suma Cero (Suc Cero)))
= Suc (Suc (Suc Cero))
```

Definiciones con tipos recursivos

- `(suma m n)` es la suma de los número naturales `m` y `n`. Por ejemplo,

```
*Main> suma (Suc (Suc Cero)) (Suc Cero)
Suc (Suc (Suc Cero))
```

```
suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)
```

- Ejemplo de cálculo:

```
  suma (Suc (Suc Cero)) (Suc Cero)
= Suc (suma (Suc Cero) (Suc Cero))
= Suc (Suc (suma Cero (Suc Cero)))
= Suc (Suc (Suc Cero))
```

Tipo recursivo con parámetro: Las listas

- Definicón del tipo lista:

```
data Lista a = Nil | Cons a (Lista a)
```

- (longitud xs) es la longitud de la lista xs. Por ejemplo,

```
longitud (Cons 2 (Cons 3 (Cons 5 Nil)))  ⇝  3
```

```
longitud :: Lista a -> Int
longitud Nil            = 0
longitud (Cons _ xs) = 1 + longitud xs
```

Tipo recursivo con parámetro: Las listas

- Definicón del tipo lista:

```
data Lista a = Nil | Cons a (Lista a)
```

- (longitud xs) es la longitud de la lista xs. Por ejemplo,

```
longitud (Cons 2 (Cons 3 (Cons 5 Nil)))  ⇝  3
```

```
longitud :: Lista a -> Int
longitud Nil           = 0
longitud (Cons _ xs) = 1 + longitud xs
```

Definición de tipos recursivos: Los árboles binarios

- Ejemplo de árbol binario:

```
        5
       / \
      /   \
     3     7
    / \   / \
   1   4 6   9
```

- Definición del tipo de árboles binarios:

```
data Arbol = Hoja Int | Nodo Arbol Int Arbol
```

- Representación del ejemplo

```
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))
```

Definiciones sobre árboles binarios

- (ocurre m a) se verifica si m ocurre en el árbol a. Por ejemplo,

```
ocurre 4  ejArbol  ⇝  True
ocurre 10 ejArbol  ⇝  False
```

```
ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d
```

- (aplana a) es la lista obtenida aplanando el árbol a. Por ejemplo,

```
aplana ejArbol  ⇝  [1,3,4,5,6,7,9]
```

```
aplana :: Arbol -> [Int]
aplana (Hoja n)     = [n]
aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d
```

Definiciones sobre árboles binarios

- (ocurre m a) se verifica si m ocurre en el árbol a. Por ejemplo,

```
ocurre 4  ejArbol  ⇝  True
ocurre 10 ejArbol  ⇝  False
```

```haskell
ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d
```

- (aplana a) es la lista obtenida aplanando el árbol a. Por ejemplo,

```
aplana ejArbol  ⇝  [1,3,4,5,6,7,9]
```

```haskell
aplana :: Arbol -> [Int]
aplana (Hoja n)     = [n]
aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d
```

Definiciones sobre árboles binarios

- `(ocurre m a)` se verifica si `m` ocurre en el árbol `a`. Por ejemplo,

  ```
  ocurre 4  ejArbol  ⇝  True
  ocurre 10 ejArbol  ⇝  False
  ```

  ```haskell
  ocurre :: Int -> Arbol -> Bool
  ocurre m (Hoja n)     = m == n
  ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d
  ```

- `(aplana a)` es la lista obtenida aplanando el árbol `a`. Por ejemplo,

  ```
  aplana ejArbol  ⇝  [1,3,4,5,6,7,9]
  ```

  ```haskell
  aplana :: Arbol -> [Int]
  aplana (Hoja n)     = [n]
  aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d
  ```

Definiciones sobre árboles binarios

- Un árbol es ordenado si el valor de cada nodo es mayor que los de su subárbol izquierdo y menor que los de su subárbol derecho.
- El árbol del ejemplo es ordenado.
- `(ocurreEnArbolOrdenado m a)` se verifica si `m` ocurre en el árbol ordenado `a`. Por ejemplo,

```
ocurreEnArbolOrdenado 4  ejArbol  ⇝  True
ocurreEnArbolOrdenado 10 ejArbol  ⇝  False
```

```
ocurreEnArbolOrdenado :: Int -> Arbol -> Bool
ocurreEnArbolOrdenado m (Hoja n)  =  m == n
ocurreEnArbolOrdenado m (Nodo i n d)
     | m == n       =  True
     | m < n        =  ocurreEnArbolOrdenado m i
     | otherwise    =  ocurreEnArbolOrdenado m d
```

Definiciones sobre árboles binarios

- Un árbol es ordenado si el valor de cada nodo es mayor que los de su subárbol izquierdo y menor que los de su subárbol derecho.
- El árbol del ejemplo es ordenado.
- `(ocurreEnArbolOrdenado m a)` se verifica si `m` ocurre en el árbol ordenado `a`. Por ejemplo,

```
ocurreEnArbolOrdenado 4  ejArbol  ~>  True
ocurreEnArbolOrdenado 10 ejArbol  ~>  False
```

```
ocurreEnArbolOrdenado :: Int -> Arbol -> Bool
ocurreEnArbolOrdenado m (Hoja n)  =  m == n
ocurreEnArbolOrdenado m (Nodo i n d)
     | m == n        =  True
     | m < n         =  ocurreEnArbolOrdenado m i
     | otherwise     =  ocurreEnArbolOrdenado m d
```

Definiciones de distintos tipos de árboles

- Árboles binarios con valores en las hojas:

```
data Arbol a = Hoja a | Nodo (Arbol a) (Arbol a)
```

- Árboles binarios con valores en los nodos:

```
data Arbol a = Hoja | Nodo (Arbol a) a (Arbol a)
```

- Árboles binarios con valores en las hojas y en los nodos:

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

- Árboles con un número variable de sucesores:

```
data Arbol a = Nodo a [Arbol a]
```

Sintaxis de la lógica proposicional

- Definición de fórmula proposicional:
  - Las variables proposicionales son fórmulas.
  - Si F es una fórmula, entonces $\neg F$ también lo es.
  - Si F y G son fórmulas, entonces $F \wedge G$ y $F \rightarrow G$ también lo son.
- Tipo de dato de fórmulas proposicionales:

```
data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
           deriving Show
```

Sintaxis de la lógica proposicional

- Ejemplos de fórmulas proposicionales:
  1. $A \wedge \neg A$
  2. $(A \wedge B) \rightarrow A$
  3. $A \rightarrow (A \wedge B)$
  4. $(A \rightarrow (A \rightarrow B)) \rightarrow B$

```
p1, p2, p3, p4 :: FProp
p1 = Conj (Var 'A') (Neg (Var 'A'))
p2 = Impl (Conj (Var 'A') (Var 'B')) (Var 'A')
p3 = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
p4 = Impl (Conj (Var 'A') (Impl (Var 'A') (Var 'B')))
          (Var 'B')
```

Semántica de la lógica proposicional

- Tablas de verdad de las conectivas:

i	$\neg i$
T	F
F	T

i	j	$i \wedge j$	$i \to j$
T	T	T	T
T	F	F	F
F	T	F	T
F	F	F	T

- Tabla de verdad para $(A \to B) \vee (B \to A)$:

A	B	$(A \to B)$	$(B \to A)$	$(A \to B) \vee (B \to A)$
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

Semántica de la lógica proposicional

- Las interpretaciones son listas formadas por el nombre de una variable proposicional y un valor de verdad.

```
type Interpretacion = [(Char, Bool)]
```

- `(valor i p)` es el valor de la fórmula `p` en la interpretación `i`. Por ejemplo,

```
valor [('A',False),('B',True)] p3  ⇝  True
valor [('A',True),('B',False)] p3  ⇝  False
```

```
valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)    = b
valor i (Var x)      = busca x i
valor i (Neg p)      = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q
```

Semántica de la lógica proposicional

- Las interpretaciones son listas formadas por el nombre de una variable proposicional y un valor de verdad.

```
type Interpretacion = [(Char, Bool)]
```

- `(valor i p)` es el valor de la fórmula `p` en la interpretación `i`. Por ejemplo,

```
valor [('A',False),('B',True)] p3  ⇝  True
valor [('A',True),('B',False)] p3  ⇝  False
```

```
valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q
```

Semántica de la lógica proposicional

- (busca c t) es el valor del primer elemento de la lista de asociación t cuya clave es c. Por ejemplo,

```
busca 2 [(1,'a'),(3,'d'),(2,'c')]  ⇝  'c'
```

```
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']
```

- (variables p) es la lista de los nombres de las variables de p.

```
variables p3  ⇝  "AAB"
```

```
variables :: FProp -> [Char]
variables (Const _)   = []
variables (Var x)     = [x]
variables (Neg p)     = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

Semántica de la lógica proposicional

- (busca c t) es el valor del primer elemento de la lista de asociación t cuya clave es c. Por ejemplo,

  ```
  busca 2 [(1,'a'),(3,'d'),(2,'c')]  ⇝  'c'
  ```

```haskell
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']
```

- (variables p) es la lista de los nombres de las variables de p.

  ```
  variables p3  ⇝  "AAB"
  ```

```haskell
variables :: FProp -> [Char]
variables (Const _)   = []
variables (Var x)     = [x]
variables (Neg p)     = variables p
variables (Conj p q)  = variables p ++ variables q
variables (Impl p q)  = variables p ++ variables q
```

Semántica de la lógica proposicional

- (busca c t) es el valor del primer elemento de la lista de asociación t cuya clave es c. Por ejemplo,

  busca 2 [(1,'a'),(3,'d'),(2,'c')] ⇝ 'c'

```
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']
```

- (variables p) es la lista de los nombres de las variables de p.

  variables p3 ⇝ "AAB"

```
variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

Semántica de la lógica proposicional

- `(interpretacionesVar n)` es la lista de las interpretaciones con `n` variables. Por ejemplo,

```
*Main> interpretacionesVar 2
[[False,False],
 [False,True],
 [True,False],
 [True,True]]
```

```
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0 = [[]]
interpretacionesVar n =
    map (False:) bss ++ map (True:) bss
    where bss = interpretacionesVar (n-1)
```

Semántica de la lógica proposicional

- `(interpretacionesVar n)` es la lista de las interpretaciones con `n` variables. Por ejemplo,

```
*Main> interpretacionesVar 2
[[False,False],
 [False,True],
 [True,False],
 [True,True]]
```

```haskell
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0 = [[]]
interpretacionesVar n =
    map (False:) bss ++ map (True:) bss
    where bss = interpretacionesVar (n-1)
```

Semántica de la lógica proposicional

- `(interpretaciones p)` es la lista de las interpretaciones de la fórmula p. Por ejemplo,

```
*Main> interpretaciones p3
[[('A',False),('B',False)],
 [('A',False),('B',True)],
 [('A',True),('B',False)],
 [('A',True),('B',True)]]
```

```
interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
    [zip vs i | i <- interpretacionesVar (length vs)]
    where vs = nub (variables p)
```

Semántica de la lógica proposicional

- `(interpretaciones p)` es la lista de las interpretaciones de la fórmula p. Por ejemplo,

```
*Main> interpretaciones p3
[[('A',False),('B',False)],
 [('A',False),('B',True)],
 [('A',True),('B',False)],
 [('A',True),('B',True)]]
```

```
interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
    [zip vs i | i <- interpretacionesVar (length vs)]
    where vs = nub (variables p)
```

Decisión de tautología

- (esTautologia p) se verifica si la fórmula p es una tautología. Por ejemplo,

```
esTautologia p1  ⇝  False
esTautologia p2  ⇝  True
esTautologia p3  ⇝  False
esTautologia p4  ⇝  True
```

```
esTautologia :: FProp -> Bool
esTautologia p =
    and [valor i p | i <- interpretaciones p]
```

Decisión de tautología

- (esTautologia p) se verifica si la fórmula p es una tautología. Por ejemplo,

```
esTautologia p1  ⇝  False
esTautologia p2  ⇝  True
esTautologia p3  ⇝  False
esTautologia p4  ⇝  True
```

```haskell
esTautologia :: FProp -> Bool
esTautologia p =
    and [valor i p | i <- interpretaciones p]
```

Evaluación de expresiones aritméticas

- Una expresión aritmética es un número entero o la suma de dos expresiones.

```
data Expr =  Num Int | Suma Expr Expr
```

- `(valorEA x)` es el valor de la expresión aritmética `x`.

  `valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4))` ⇝ 9

```
valorEA :: Expr -> Int
valorEA (Num n)    = n
valorEA (Suma x y) = valorEA x + valorEA y
```

- Cálculo:

```
  valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + (valorEA (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + 4
= (valorEA (Num 2) + (valorEA (Num 3))) + 4
= (2 + 3) + 4
= 9
```

Evaluación de expresiones aritméticas

- Una expresión aritmética es un número entero o la suma de dos expresiones.

```
data Expr =  Num Int | Suma Expr Expr
```

- `(valorEA x)` es el valor de la expresión aritmética `x`.

```
valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4))  ⇝  9
```

```
valorEA :: Expr -> Int
valorEA (Num n)     = n
valorEA (Suma x y) = valorEA x + valorEA y
```

- Cálculo:

```
  valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + (valorEA (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + 4
= (valorEA (Num 2) + (valorEA (Num 3))) + 4
= (2 + 3) + 4
= 9
```

Máquina de cálculo aritmético

- La pila de control de la máquina abstracta es una lista de operaciones.

```
type PControl = [Op]
```

- Las operaciones son meter una expresión en la pila o sumar un número con el primero de la pila.

```
data Op =  METE Expr | SUMA Int
```

Máquina de cálculo aritmético

- `(eval x p)` evalúa la expresión `x` con la pila de control `p`. Por ejemplo,

```
eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) []  ⇝  9
eval (Suma (Num 2) (Num 3)) [METE (Num 4)]     ⇝  9
eval (Num 3) [SUMA 2, METE (Num 4)]            ⇝  9
eval (Num 4) [SUMA 5]                          ⇝  9
```

```
eval :: Expr -> PControl -> Int
eval (Num n)    p = ejec p n
eval (Suma x y) p = eval x (METE y : p)
```

Máquina de cálculo aritmético

- `(eval x p)` evalúa la expresión `x` con la pila de control `p`. Por ejemplo,

```
eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) []   ~>  9
eval (Suma (Num 2) (Num 3)) [METE (Num 4)]       ~>  9
eval (Num 3) [SUMA 2, METE (Num 4)]              ~>  9
eval (Num 4) [SUMA 5]                            ~>  9
```

```
eval :: Expr -> PControl -> Int
eval (Num n)      p = ejec p n
eval (Suma x y) p = eval x (METE y : p)
```

Máquina de cálculo aritmético

- `(ejec p n)` ejecuta la lista de control `p` sobre el entero `n`. Por ejemplo,

```
ejec [METE (Num 3), METE (Num 4)] 2  ⇝  9
ejec [SUMA 2, METE (Num 4)]       3  ⇝  9
ejec [METE (Num 4)]               5  ⇝  9
ejec [SUMA 5]                     4  ⇝  9
ejec []                           9  ⇝  9
```

```
ejec :: PControl -> Int -> Int
ejec []           n = n
ejec (METE y : p) n = eval y (SUMA n : p)
ejec (SUMA n : p) m = ejec p (n+m)
```

Máquina de cálculo aritmético

- `(ejec p n)` ejecuta la lista de control `p` sobre el entero `n`. Por ejemplo,

  ```
  ejec [METE (Num 3), METE (Num 4)] 2  ⇝  9
  ejec [SUMA 2, METE (Num 4)]       3  ⇝  9
  ejec [METE (Num 4)]               5  ⇝  9
  ejec [SUMA 5]                     4  ⇝  9
  ejec []                           9  ⇝  9
  ```

```
ejec :: PControl -> Int -> Int
ejec []            n = n
ejec (METE y : p) n = eval y (SUMA n : p)
ejec (SUMA n : p) m = ejec p (n+m)
```

Máquina de cálculo aritmético

- `(evalua e)` evalúa la expresión aritmética e con la máquina abstracta. Por ejemplo,

  ```
  evalua (Suma (Suma (Num 2) (Num 3)) (Num 4))  ~>  9
  ```

```
evalua :: Expr -> Int
evalua e = eval e []
```

- Evaluación:

  ```
    eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) []
  = eval (Suma (Num 2) (Num 3)) [METE (Num 4)]
  = eval (Num 2) [METE (Num 3), METE (Num 4)]
  = ejec [METE (Num 3), METE (Num 4)] 2
  = eval (Num 3) [SUMA 2, METE (Num 4)]
  = ejec [SUMA 2, METE (Num 4)] 3
  = ejec [METE (Num 4)] (2+3)
  = ejec [METE (Num 4)] 5
  = eval (Num 4) [SUMA 5]
  = ejec [SUMA 5] 4
  = ejec [] (5+4)
  = ejec [] 9
  = 9
  ```

Máquina de cálculo aritmético

- (evalua e) evalúa la expresión aritmética e con la máquina abstracta. Por ejemplo,

```
evalua (Suma (Suma (Num 2) (Num 3)) (Num 4))  ~>  9
```

```
evalua :: Expr -> Int
evalua e = eval e []
```

- Evaluación:

```
  eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) []
= eval (Suma (Num 2) (Num 3)) [METE (Num 4)]
= eval (Num 2) [METE (Num 3), METE (Num 4)]
= ejec [METE (Num 3), METE (Num 4)] 2
= eval (Num 3) [SUMA 2, METE (Num 4)]
= ejec [SUMA 2, METE (Num 4)] 3
= ejec [METE (Num 4)] (2+3)
= ejec [METE (Num 4)] 5
= eval (Num 4) [SUMA 5]
= ejec [SUMA 5] 4
= ejec [] (5+4)
= ejec [] 9
= 9
```

Declaraciones de clases

- Las clases se declaran mediante el mecanismo `class`.
- Ejemplo de declaración de clases:

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

    -- Minimal complete definition: (==) or (/=)
    x == y = not (x/=y)
    x /= y = not (x==y)
```

Declaraciones de instancias

- Las instancias se declaran mediante el mecanismo `instance`.
- Ejemplo de declaración de instancia:

Prelude

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

Extensiones de clases

- Las clases pueden extenderse mediante el mecanismo `class`.
- Ejemplo de extensión de clases:

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

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x <= y      = y
              | otherwise   = x
    min x y   | x <= y      = x
              | otherwise   = y
```

Instancias de clases extendidas

- Las instancias de las clases extendidas pueden declararse mediante el mecanismo `instance`.
- Ejemplo de declaración de instancia:

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

Clases derivadas

- Al definir un nuevo tipo con `data` puede declarse como instancia de clases mediante el mecanismo `deriving`.
- Ejemplo de clases derivadas:

Prelude

```
data Bool = False | True
            deriving (Eq, Ord, Read, Show)
```

- Comprobación:

```
False == False          ⇝  True
False < True            ⇝  True
show False              ⇝  "False"
read "False" :: Bool    ⇝  False
```

Clases derivadas

- Para derivar un tipo cuyos constructores tienen argumentos como derivado, los tipos de los argumentos tienen que ser instancias de las clases derivadas.
- Ejemplo:

```
data Figura = Circulo Float | Rect Float Float
              deriving (Eq, Ord, Show)
```

se cumple que `Float` es instancia de `Eq`, `Ord` y `Show`.

```
*Main> :info Float
...
instance Eq Float
instance Ord Float
instance Show Float
...
```

Bibliografía

1. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
    - Cap. 10: Declaring types and classes.
2. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
    - Cap. 4: Definición de tipos.
    - Cap. 5: El sistema de clases de Haskell.
3. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
    - Cap. 12: Overloading and type classes.
    - Cap. 13: Checking types.
    - Cap. 14: Algebraic types.