Implementación de TADs en lenguajes funcionales

Estructuras de Datos y Algoritmos /
/ Algoritmos y Estructuras de Datos II
Año 2025

Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

El Lenguaje Haskell

- Haskell fue introducido en 1987 con el objetivo de desarrollar un lenguaje funcional moderno
- Vamos a usar el compilador Glasgow Haskell, que se puede obtener en http://www.haskell.org/ghc/.

Tipos Básicos de Haskell

Haskell tiene un conjunto rico de tipos básicos de datos

- **Booleanos**, tipos de booleanos con las operaciones lógicas,
- **Int**: Enteros de precisión fija,
- **Char**: ‘a’, ‘b’, ‘c’, etc
- **Integer**: Enteros de precisión variable,
- **Float**: Números reales.

Escribimos: `E::T` cuando `E` es de tipo `T`

```
 5  :: Integer
'a' :: Char                          [1,2,3] :: [Integer]
inc :: Integer -> Integer            ('b',4) :: (Char,Integer)
```

El Tipo Bool

El tipo `Bool` tiene dos valores `true` y `false`, y las siguientes operaciones:

- `&& :: Bool->Bool->Bool`
- `|| :: Bool->Bool->Bool`
- `not :: Bool->Bool`

Cualquier función: `f::A->Bool` es llamado predicado, y puede utilizarse con estas operaciones.

`==::A->A->Bool`

La igualdad es la más conocida

El Tipo Char

El tipo `char` contiene los valores `'a','b','c',` ... etc, y las siguientes funciones:

- `ord::Char -> Int` convierte caracteres a enteros.
- `chr::Int -> Char` convierte enteros a caracteres.

Los Strings se modelan como una lista de chars.

Sistema de Números

Haskell tiene varios tipos numéricos:

- `Int`, enteros con precisión limitada $[-2^{29}, 2^{29})$.
- `Float`, reales 3.14159
- `Double`, reales con doble precisión.

Con las operaciones típicas: `+`, `-`, `*`, `/`, ...

Tuplas

Usando los tipos básicos podemos construir tuplas y listas.
Dados tipos `A` y `B`:

`(A,B)` — El tipo de pares de `A` y `B`

Por ejemplo: `(True,1)::(Bool,Int)`

Operaciones:

`fst::(a,b)->a` — Devuelve el primer componente

`snd::(a,b)->b` — Devuelve el segundo componente

Listas

Dado un tipo cualquiera `a`:

`[a]` Es el tipo a las listas con elementos de tipo `a`

Las listas son tipos recursivos, y sus valores se crean usando los siguientes constructores:

- `[]` es la lista vacía
- `x:xs` es la lista con `x` a la cabeza y luego `xs` a la cola

Por ejemplo:

La lista `[1,2,3]` se construye con: `1:(2:(3:[]))`

Las listas en Haskell son genéricas.
Todos los elementos de la lista son del mismo tipo.

Operaciones sobre Listas

Algunas funciones útiles sobre listas:

- `head::[a]->a`, devuelve la cabeza de la lista.
- `last::[a]->a`, devuelve el último elemento.
- `tail::[a]->[a]`, devuelve la cola de la lista.
- `length::[a]->Int`, retorna la longitud de la lista.
- `++::[a]->[a]->[a]`, concatena dos listas.

Funciones

Dados dos tipos a y b:

`a->b`

Es el tipo de las funciones de `a` en `b`

Por ejemplo:

`Not:Bool->Bool`

Función de `Bool` en `Bool`

Las funciones de alto orden son aquellas que toman como parámetros funciones o retornan funciones

`::(b->c)->(a->b)->(a->c)`

Definir la composición de funciones

Definición de Funciones

Una función se puede definir por casos:

```
sign :: Int -> Int
sign x | x>=0 = 1
       | x<0  = -1
```

También usando recursión y pattern matching:

```
suma :: [Int] -> Int
suma [] = 0
suma (x:xs) = x + suma xs
```

Si ejecutamos esta función para [1,2,3,4,5] obtenemos:

```
suma [1,2,3,4,5] -> 1+(2+(3+(4+(5+0)) -> 15
```

Ejemplos de operaciones sobre listas

```
head :: [a] -> a
head (x:xs) = x
```

`head`, `tail` y `last` son funciones parciales, dan error para `[]`

```
tail :: [a] -> a
tail (x:xs) = xs
```

```
last :: [a] -> a
last [x]    = x
last (x:xs) = last xs
```

`[x]` es simplemente una forma abreviada para `(x:[])`

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Todas estas funciones vienen predefinidas en el prelude de Haskell

Polimorfismo

Consideremos las siguientes funciones:

Descarta los primeros n elementos de la lista

```
drop::Int->[a]->[a]
drop n [] = []
drop 0 xs = xs
drop n (x:xs) = drop (n-1) xs
```

Toma los primeros n elementos de una lista

```
take :: Int->[a]->[a]
take 0 xs = []
take n [] = []
take n (x:xs) = x:take (n-1) xs
```

En `[a]`, `a` puede ser cualquier tipo (es una variable de tipos).
Se dice que `drop` y `take` son funciones polimórficas.

Sinónimos de tipos

Podemos definir sinónimos de tipos con el constructor `type`:

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

Es la definición de String en Haskell

También por ejemplo:

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

Posiciones en un tablero

Y podemos usar este tipo nuevo:

```
type Board =[Pos]
```

Un tablero es una lista de posiciones

Tipos definidos por el programador

Podemos definir tipos con nuevos valores:

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

Y se pueden definir tipos inductivos

```
data Nat = Zero | Succ Nat
```

Definen los naturales por medio de dos constructores:

`Zero::Nat` y `Succ::Nat->Nat`

Algunos valores del tipo son:

`Zero, Succ Zero, Succ (Succ Zero), …`

Clases en Haskell

Una operación se dice sobrecargada si puede utilizarse para varios tipos.

Solo está bien definida si el tipo tiene la igualdad definida

```
elem x [] = False
elem x (y:ys) | x==y = True
              | otherwise = elem x ys
```

Una **clase** en Haskell define una colección de tipos que tienen una operación en común

Clases en Haskell

La clase que tiene la igualdad se define:

Todos los tipos que **instancien** esta clase deben tener la igualdad definida

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

Por ejemplo, para decir que Nat pertenecen a Eq:

```
instance Eq Nat where
  Zero == Zero        = True
  Zero == Succ n      = False
  Succ n == Zero      = False
  Succ n == Succ m    = n == m
```

Utilizando Clases

En el ejemplo anterior el perfil de la función sería:

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

Dado que el tipo a pertenece a la clase Eq

La función tiene este perfil

Si derivamos la igualdad con “deriving” Haskell la deriva de la forma más simple posible: dos expresiones son iguales cuando se escriben igual

Cuando no queremos esta igualdad, la tenemos que instanciar nosotros.

La Clase Show

En la clase Show “están” todos los tipos que tienen implementado el método show, que se define como:

```
class Show a where
    show :: a -> String
```

El que usa Haskell para mostrar por pantalla

Si la instanciamos con “deriving” Haskell mostrará las expresiones de la misma forma que se escriben.

Si queremos algo mejor podemos redefinir show, por ejemplo:

```
toInt :: Nat -> Int
toInt Zero = 0
toInt (Succ m) = 1 + toInt m

instance Show Nat where
    show :: Nat -> String
    show n = show (toInt n)
```

La Clase Ord

Los miembros de la clase Ord proveen un orden sobre sus elementos:

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

Eq se dice **superclase** Ord

Solo es necesario implementar uno de ellos

Los miembros de la clase Ord proveen un orden sobre sus elementos:

```
instance Ord Nat where
  Zero <= _ = True
  Succ n <= Zero = False
  Succ n <= Succ m = n <= m
```

Instanciamos el Ord para los naturales

La Clase Num

Intuitivamente, la clase Num provee los métodos básicos que un tipo númerico tiene que tener

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Para instanciar la clase Num debemos implementar estos métodos

Ejercicio: Instanciar Num con Nat

Listas

Podemos dar nuestra propia implementación de listas en Haskell de la siguiente manera:

```
data MyList a = Vacia | Ins a (MyList a)
```

Algunos ejemplos de funciones sobre listas:

```
addLast :: a -> MyList a -> MyList a
addLast x Vacia = Ins x Vacia
addLast y (Ins x xs) = Ins x (addLast y xs)

contains :: (Eq a) => a -> MyList a -> Bool
contains _ Vacia = False
contains x (Ins y xs) = x == y || contains x xs
```

Conjuntos

Podemos implementar conjuntos en Haskell de la siguiente manera:

```
data MySet a = Vacio | Ins a (MySet a)
```

Algunos ejemplos de funciones sobre conjuntos:

```
insert :: (Eq a) => a -> MySet a -> MySet a
insert x Vacio = Ins x Vacio
insert y (Ins x xs) | x == y = (Ins x xs)
                    | x /= y = Ins x (insert y xs)

contains :: (Eq a) => a -> MySet a -> Bool
contains _ Vacio = False
contains x (Ins y xs) = x == y || contains x xs
```

Especificación de TADs en Haskell

- En Haskell especificamos los TADs mediante clases

```haskell
module TADSet where

{--
  Sets are unbounded sets of objects of type a.
  A typical Set is {o1 o2 . . . on}.

  Set requires that the type a implements the
  Eq class, since == is used to determine
  equality of elements.
--}
class Set s where
    --  Operations over sets
    empty :: s a
    insert :: Eq a => a -> s a -> s a
    contains :: Eq a => a -> s a -> Bool
    union :: Eq a => s a -> s a -> s a
```

- Las clases nos permiten definir las operaciones disponibles para el TAD
  - En el ejemplo, las operaciones de `Set` son `empty`, `insert`, `contains` y `union`

Especificación de TADs en Haskell

- En Haskell especificamos los TADs mediante clases

```haskell
module TADSet where

{--
  Sets are unbounded sets of objects of type a.
  A typical Set is {o1 o2 . . . on}.

  Set requires that the type a implements the
```

Notar la similaridad entre esta forma de usar las clases en Haskell y las interfaces en Java

```haskell
class Set s where
    --  Operations over sets
    empty :: s a
    insert :: Eq a => a -> s a -> s a
    contains :: Eq a => a -> s a -> Bool
    union :: Eq a => s a -> s a -> s a
```

- Las clases nos permiten definir las operaciones disponibles para el TAD
    - En el ejemplo, las operaciones de `Set` son `empty`, `insert`, `contains` y `union`

Implementación de TADs en Haskell

- Implementación basada en tipos inductivos

```
module SetInd where
import TADSet

data SetInd a = Vacio | Ins a (SetInd a)

instance Set SetInd where
    empty :: SetInd a
    empty = Vacio

    insert :: Eq a => a -> SetInd a -> SetInd a
    insert x Vacio = Ins x Vacio
    insert y (Ins x xs) | x == y = (Ins x xs)
                        | x /= y = Ins x (insert y xs)

    contains :: Eq a => a -> SetInd a -> Bool
    contains _ Vacio = False
    contains x (Ins y xs) = x == y || contains x xs

    union :: Eq a => SetInd a -> SetInd a -> SetInd a
    union Vacio xs = xs
    union (Ins x xs) ys = insert x (union xs ys)
```

Implementación de TADs en Haskell

- La función de abstracción es implementada por la función show

```
elemsToStr :: Show a => SetInd a -> String
elemsToStr Vacio = ""
elemsToStr (Ins x xs) = show x ++ " " ++ elemsToStr xs

instance Show a => Show (SetInd a) where
    show :: SetInd a -> String
    show xs = "{ " ++ (elemsToStr xs) ++ "}"
```

Implementación de TADs en Haskell

- Implementación basada en tipos preexistentes de Haskell

```
module SetList where
import TADSet

data SetList a = MakeSet [a]

containsImpl :: Eq a => a -> [a] -> Bool
containsImpl _ [] = False
containsImpl x (y:xs) = x == y || containsImpl x xs

unionImpl :: Eq a => [a] -> [a] -> [a]
unionImpl [] xs = xs
unionImpl (x:xs) ys = x : (unionImpl xs ys)
```

Implementación de TADs en Haskell

- Implementación basada en tipos preexistentes de Haskell

```
module SetList where
import TADSet

data SetList a = MakeSet [a]

containsImpl :: Eq a => a -> [a] -> Bool
contains
contains

unionImp
unionImp
unionImp
```

```
instance Set SetList where
    empty :: SetList a
    empty = MakeSet []

    insert :: Eq a => a -> SetList a -> SetList a
    insert x (MakeSet xs) = MakeSet (x:xs)

    contains :: Eq a => a -> SetList a -> Bool
    contains x (MakeSet xs) = containsImpl x xs

    union :: Eq a => SetList a -> SetList a -> SetList a
    union (MakeSet xs) (MakeSet ys) = MakeSet (unionImpl xs ys)
```

Implementación de TADs en Haskell

- La función de abstracción es implementada por la función show

```
rmdups :: [String] -> [String]
rmdups [] = []
rmdups (x:xs)   | elem x xs     = rmdups xs
                | otherwise     = x : rmdups xs

elemsToStr :: [String] -> String
elemsToStr [] = ""
elemsToStr (x:xs) =  x ++ " " ++ elemsToStr xs

instance Show a => Show (SetList a) where
    show :: Show a => SetList a -> String
    show (MakeSet xs) = "{ " ++ (elemsToStr (rmdups (map show xs))) ++ "}"
```

Actividades

- Leer los capítulos 1-5 de "A Gentle Introduction to Haskell". P. Hudak, J. Peterson & J. Fasel. 1998. Disponible en: https://www.haskell.org/tutorial/

Bibliografía

- "A gentle introduction to Haskell". P. Hudak, J. Peterson & J. Fasel. 1998. Disponible en: https://www.haskell.org/tutorial/