Laboratorio de Lenguajes de Programación

Tipos de datos abstractos - Parte II

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

31 de agosto de 2016

Objetivos

Objetivos

- Seguir practicando la definición de tipos de datos abstractos.

Objetivos

- Seguir practicando la definición de tipos de datos abstractos.
- Seguir pacticando en el uso de tipos de datos abstractos mediante la técnica de caza de patrones.

¡Pero antes!

¡Pero antes!

Recordando la pregunta....

¡Pero antes!

Recordando la pregunta....

¿Por qué `foldl` funciona distinto en Racket y en Haskell?

¡Pero antes!

Recordando la pregunta....

¿Por qué `foldl` funciona distinto en Racket y en Haskell?

```
;; foldl en Racket
(define (foldl f v l)
   (match
      ['() v]
      [(cons x xs) (foldl f (f x v) xs)]))

;; foldl en Haskell
(define (foldl f v l)
   (match
      ['() v]
      [(cons x xs) (foldl f (f v x) xs)]))
```

¿No deberían funcionar igual?

¿No deberían funcionar igual?

Ambas implementaciones son correctas pues dependen de la definición de cada primitiva y éstas las define el diseñador del lenguaje.

¿No deberían funcionar igual?

Ambas implementaciones son correctas pues dependen de la definición de cada primitiva y éstas las define el diseñador del lenguaje.

La definición de `foldl` en Haskell es la misma en muchos lenguajes, por lo que podría ser la esperada por el programador, sin embargo no podemos decir que la otra es incorrecta pues así la definió el diseñador.

¿No deberían funcionar igual?

Ambas implementaciones son correctas pues dependen de la definición de cada primitiva y éstas las define el diseñador del lenguaje.

La definición de `foldl` en Haskell es la misma en muchos lenguajes, por lo que podría ser la esperada por el programador, sin embargo no podemos decir que la otra es incorrecta pues así la definió el diseñador.

Esto es un excelente ejemplo de como algo tan simple puede cambiar la evaluación de una misma expresión en diferentes lenguajes. Más aún si cambia algún día la versión de Racket podría cambiar `foldl` en este caso y dar resultados diferentes.

¿No deberían funcionar igual?

Ambas implementaciones son correctas pues dependen de la definición de cada primitiva y éstas las define el diseñador del lenguaje.

La definición de `foldl` en Haskell es la misma en muchos lenguajes, por lo que podría ser la esperada por el programador, sin embargo no podemos decir que la otra es incorrecta pues así la definió el diseñador.

Esto es un excelente ejemplo de como algo tan simple puede cambiar la evaluación de una misma expresión en diferentes lenguajes. Más aún si cambia algún día la versión de Racket podría cambiar `foldl` en este caso y dar resultados diferentes.

Sólo tiene que ver con la implementación de `foldl` en el lenguaje y no tiene que seguir lo que otro lenguaje haya establecido en su diseño.

Para recordar...

Para recordar...

Un tipo abstracto de dato en Racket se define a partir de una gramática que nos indica cómo se construyen las expresiones de un determinado Lenguaje.

Para recordar...

Un tipo abstracto de dato en Racket se define a partir de una gramática que nos indica cómo se construyen las expresiones de un determinado Lenguaje.

Por lo general son gramáticas en BNF.

Para recordar...

Un tipo abstracto de dato en Racket se define a partir de una gramática que nos indica cómo se construyen las expresiones de un determinado Lenguaje.

Por lo general son gramáticas en BNF.

```
(define-type Booleano
   [falso]
   [verdadero]
   [mi-not (a Booleano?)]
   [mi-and (a Booleano?)  (b Booleano?)]
   [mi-or (a Booleano?)  (b Booleano?)]
   [impl (a Booleano?)  (b Booleano?)]
   [syss (a Booleano?)  (b Booleano?)])
```

Para recordar...

Para recordar...

Para usar realizar acciones sobre el nuevo tipo de dato es usual seguir la técnica de caza de patrones. Esta puede lograrse usando `type-case`, `match` o `cases`.

Para recordar...

Para usar realizar acciones sobre el nuevo tipo de dato es usual seguir la técnica de caza de patrones. Esta puede lograrse usando `type-case`, `match` o `cases`.

```
(define (evalua b)
   (type-case Booleano b
      [falso () (falso)]
      [verdadero () (verdadero)]
      [mi-not (a) (aplica-not (evalua a))]
      [mi-and (a b)
         (if (and (verdadero?  a) (verdadero?  b))
            (verdadero)
            (falso))]
      [mi-or (a b) (if (and (falso?  a) (falso?  b))
            (falso)
            (verdadero))]
      [impl (a b) (if (and (verdadero?  a) (falso?  b))
                  (falso)
                  (verdadero))]
      [syss (a b) (if (equal?  a b)
                  (verdadero)
                  (falso))]))
```

Para recordar...

Para usar realizar acciones sobre el nuevo tipo de dato es usual seguir la técnica de caza de patrones. Esta puede lograrse usando `type-case`, `match` o `cases`.

```
(define (evalua b)
   (type-case Booleano b
      [falso () (falso)]
      [verdadero () (verdadero)]
      [mi-not (a) (aplica-not (evalua a))]
      [mi-and (a b)
         (if (and (verdadero?  a) (verdadero?  b))
            (verdadero)
            (falso))]
      [mi-or (a b) (if (and (falso?  a) (falso?  b))
            (falso)
            (verdadero))]
      [impl (a b) (if (and (verdadero?  a) (falso?  b))
                  (falso)
                  (verdadero))]
      [syss (a b) (if (equal?  a b)
                  (verdadero)
                  (falso))]))
```

¿Qué error tiene la función anterior?

Corrección para el tipo `Booleano`

Corrección para el tipo `Booleano`

```
(define (evalua b)
   (type-case Booleano b
      [falso () (falso)]
      [verdadero () (verdadero)]
      [mi-not (a) (aplica-not (evalua a))]
      [mi-and (a b)
         (if (and (verdadero?  (evalua a)) (verdadero?  (evalua b)))
            (verdadero)
            (falso))]
      [mi-or (a b) (if (and (falso?  (evalua a)) (falso?  (evalua b)))
            (falso)
            (verdadero))]
      [impl (a b) (if (and (verdadero?  (evalua a)) (falso?  (evalua b)))
                  (falso)
                  (verdadero))]
      [syss (a b) (if (equal?  (evalua a) (evalua b))
                  (verdadero)
                  (falso))]))
```

Definiendo Listas

Definiendo Listas

Las listas en Racket se define como:

Definiendo Listas

Las listas en Racket se define como:

- La lista vacía es una lista y se representa por `'()`.
- Si `x` es un elemento de un conjunto cualquiera, entonces `(cons x xs)` es una lista. A `x` se le llama la cabeza y a `xs` la cola de la lista.
- Solo éstas son listas.

Definiendo Listas

Las listas en Racket se define como:

- La lista vacía es una lista y se representa por `'()`.
- Si `x` es un elemento de un conjunto cualquiera, entonces `(cons x xs)` es una lista. A `x` se le llama la cabeza y a `xs` la cola de la lista.
- Solo éstas son listas.

Podemos usar esta definición para definir nuestro propio tipo de dato Lista.

El tipo de dato `Lista`

El tipo de dato `Lista`

Si recordamos una definición más práctica de listas, podemos representar a una lista como una sucesión de nodos, donde cada uno guarda un elemento y tiene una referencia al nodo siguiente.

El tipo de dato `Lista`

Si recordamos una definición más práctica de listas, podemos representar a una lista como una sucesión de nodos, donde cada uno guarda un elemento y tiene una referencia al nodo siguiente.

Siguiendo esta idea, podemos definir las listas mediante un constructor `nodo` en lugar del típico `cons`.

El tipo de dato `Lista`

Si recordamos una definición más práctica de listas, podemos representar a una lista como una sucesión de nodos, donde cada uno guarda un elemento y tiene una referencia al nodo siguiente.

Siguiendo esta idea, podemos definir las listas mediante un constructor `nodo` en lugar del típico `cons`.

```
(define-type Nodo
   [vacio]
   [nodo (elemento integer?)  (siguiente Nodo?)])
```

El tipo de dato `Lista`

Si recordamos una definición más práctica de listas, podemos representar a una lista como una sucesión de nodos, donde cada uno guarda un elemento y tiene una referencia al nodo siguiente.

Siguiendo esta idea, podemos definir las listas mediante un constructor `nodo` en lugar del típico `cons`.

```
(define-type Nodo
   [vacio]
   [nodo (elemento integer?)  (siguiente Nodo?)])
```

Por ejemplo, la lista `'(1 2 3 4)` se representa como `(nodo 1 (nodo 2 (nodo 3 (nodo 4 vacia))))`.

El tipo de dato `Lista`

Si recordamos una definición más práctica de listas, podemos representar a una lista como una sucesión de nodos, donde cada uno guarda un elemento y tiene una referencia al nodo siguiente.

Siguiendo esta idea, podemos definir las listas mediante un constructor `nodo` en lugar del típico `cons`.

```
(define-type Nodo
   [vacio]
   [nodo (elemento integer?)  (siguiente Nodo?)])
```

Por ejemplo, la lista `'(1 2 3 4)` se representa como `(nodo 1 (nodo 2 (nodo 3 (nodo 4 vacia))))`.

Y si... ¿mejor hacemos listas genéricas?

Listas genéricas

Listas genéricas

Para definir una lista genérica, necesitamos de un predicado que se evalúe a verdadero con cualquier tipo de dato y que no acepte sólo enteros, cadenas o símbolos por ejemplo.

Listas genéricas

Para definir una lista genérica, necesitamos de un predicado que se evalúe a verdadero con cualquier tipo de dato y que no acepte sólo enteros, cadenas o símbolos por ejemplo.

Llamaremos a este predicado `any?` y su definición es la siguiente:

Listas genéricas

Para definir una lista genérica, necesitamos de un predicado que se evalúe a verdadero con cualquier tipo de dato y que no acepte sólo enteros, cadenas o símbolos por ejemplo.

Llamaremos a este predicado `any?` y su definición es la siguiente:

```
(define (any?  a)
   #t)
```

Listas genéricas

Para definir una lista genérica, necesitamos de un predicado que se evalúe a verdadero con cualquier tipo de dato y que no acepte sólo enteros, cadenas o símbolos por ejemplo.

Llamaremos a este predicado `any?` y su definición es la siguiente:

```
(define (any?  a)
   #t)
```

De esta forma, nuestro tipo para representar listas queda como sigue:

Listas genéricas

Para definir una lista genérica, necesitamos de un predicado que se evalúe a verdadero con cualquier tipo de dato y que no acepte sólo enteros, cadenas o símbolos por ejemplo.

Llamaremos a este predicado `any?` y su definición es la siguiente:

```
(define (any?  a)
  #t)
```

De esta forma, nuestro tipo para representar listas queda como sigue:

```
(define-type Nodo
  [vacio]
  [nodo (elemento any?)  (siguiente Nodo?)])
```

Listas genéricas

Para definir una lista genérica, necesitamos de un predicado que se evalúe a verdadero con cualquier tipo de dato y que no acepte sólo enteros, cadenas o símbolos por ejemplo.

Llamaremos a este predicado `any?` y su definición es la siguiente:

```
(define (any?  a)
  #t)
```

De esta forma, nuestro tipo para representar listas queda como sigue:

```
(define-type Nodo
  [vacio]
  [nodo (elemento any?)  (siguiente Nodo?)])
```

¡Ahora tenemos listas genéricas y heterogéneas!

Ejemplos

Ejemplos

Example

Define una función `(longitud l)` que nos devuelva la longitud de una lista.

Ejemplos

Example

Define una función `(longitud l)` que nos devuelva la longitud de una lista.

```
(define (longitud l)
   (type-case Nodo l
      [vacio () 0]
      [nodo (e s) (+ 1 (longitud s))]))
```

Ejemplos

Example

Define una función `(longitud l)` que nos devuelva la longitud de una lista.

```
(define (longitud l)
   (type-case Nodo l
      [vacio () 0]
      [nodo (e s) (+ 1 (longitud s))]))
```

Example

Define una función `(vacio? l)` que nos indique si una lista es vacía.

Ejemplos

Example

Define una función `(longitud l)` que nos devuelva la longitud de una lista.

```
(define (longitud l)
   (type-case Nodo l
      [vacio () 0]
      [nodo (e s) (+ 1 (longitud s))]))
```

Example

Define una función `(vacio? l)` que nos indique si una lista es vacía.

¡Esta función ya existe! Nos la regala el lenguaje.

Ejemplos

Ejemplos

Example

Define una función `(agrega-final x l)` que agrega un elemento al final de una lista.

Ejemplos

Example

Define una función `(agrega-final x l)` que agrega un elemento al final de una lista.

```
(define (agrega-final x l)
   (type-case Nodo l
      [vacio () (nodo x (vacio))]
      [nodo (e s) (nodo e (agrega-final x s))]))
```

Ejemplos

Example

Define una función `(agrega-final x l)` que agrega un elemento al final de una lista.

```
(define (agrega-final x l)
   (type-case Nodo l
      [vacio () (nodo x (vacio))]
      [nodo (e s) (nodo e (agrega-final x s))]))
```

Example

Define una función `(elimina-primero l)` que elimina el primer elemento de una lista.

Ejemplos

Example

Define una función `(agrega-final x l)` que agrega un elemento al final de una lista.

```
(define (agrega-final x l)
   (type-case Nodo l
      [vacio () (nodo x (vacio))]
      [nodo (e s) (nodo e (agrega-final x s))]))
```

Example

Define una función `(elimina-primero l)` que elimina el primer elemento de una lista.

```
(define (elimina-primero l)
   (type-case Nodo l
      [vacio () (error ‘‘Error:  Lista vacía’’)]
      [nodo (e s) s]))
```

Ejemplos

Ejemplos

Example

Define una función `(contiene? x l)` que determina si un elemento forma parte de una lista.

Ejemplos

Example

Define una función `(contiene? x l)` que determina si un elemento forma parte de una lista.

```
(define (contiene? x l)
  (type-case Nodo l
    [vacio () #f]
    [nodo (e s)
          (if (equal? x e)
            #t
            (contiene? x s))]))
```

Ejercicios

Los ejercicios se encuentran en
`sites.ciencias.unam.mx/ldp171/practicas`

Enviar al correo `manu+ldp@ciencias.unam.mx` **un archivo** `equipo_sesion4.rkt` **con asunto** `[LDP-Sesión 4]` **a más tardar a las 18:59:59.**

Incluir el nombre de los integrantes en el cuerpo del correo.

Sólo pueden entregar aquellos alumnos que aparezcan en la lista de asistencia de la sesión. No es válido apuntar a miembros del equipo que no estén presentes.