# Programación Funcional Avanzada

## Mejoramiento de desempeño

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"



UNIVERSIDAD SIMÓN BOLÍVAR

# Mejoramiento del Desempeño

## Quiero que mi programa corra más rápido...

- Dejar que GHC lo optimize – funciona para la "mente funcional".
- Aprovechar librerías del sistema – optimizadas para su comfort.
- Si no es suficiente, aplicar las técnicas de *profiling* para determinar cuál es la causa del problema.
  - Mejoras dramáticas vienen de cambios dramáticos – usualmente de algoritmos y en los lugares adecuados.
  - Lugares que no puedes identificar si no haces *profiling*.

*La optimización prematura es la raíz de todos los males*
*– Donald Knuth*

UNIVERSIDAD SIMÓN BOLÍVAR

# La flojera es tu amiga

## Procrastinando para ser productivo y eficiente

- Haskell es un lenguaje perezoso – evaluación normal.
- Nada se evalúa a menos que sea absolutamente necesario.
  - Aplica a funciones.
  - Aplica a constructores de datos – que son funciones.
  - Aplica a definiciones locales.
- Evaluación perezosa favorece la programación modular.
  - Memorización implícita – espacio en lugar de cómputo.
  - Permite separar productores de consumidores.
  - Permite especificar productores sobre un conjunto infinito.

UNIVERSIDAD SIMÓN BOLÍVAR

# *Mergesort*

## Mejor manera de ordenar en un lenguaje funcional

```
msort []  = []
msort [x] = [x]
msort xs  = merge (msort miti) (msort mita)
  where
    (miti,mita) = halve xs

merge xs [] = xs
merge [] ys = ys
merge xs@(x:t) ys@(y:u)
      | x <= y     = x : merge t ys
      | otherwise = y : merge xs u
```

- `merge` – combina dos listas ordenadas, preservando el orden.
- `halve` – divide la lista en dos mitades.

¿Cómo escribirmos `halve`?

UNIVERSIDAD SIMÓN BOLÍVAR

# Dividir una lista en dos

## Primera versión – recursión de cola

```
halve = go ([],[])
  where
   go (eac,oac) []         = (eac,oac)
   go (eac,oac) [x]        = (x:eac,oac)
   go (eac,oac) (x:y:xs) = go (x:eac,y:oac) xs
```

- ¿Cómo que usar `length`, dividir entre dos. . . ? – facepalm!
- Es recursiva de cola – eso "tiene que ser bueno".
- Puede escribirse como un `foldl'` ligeramente truculento – queda como ejercicio para el lector.

¿Será realmente la mejor solución?

UNIVERSIDAD SIMÓN BOLÍVAR

# Dividir una lista en dos

## Una función modular quiere ser reutilizada

```
halve = go ([],[])
  where
   go (eac,oac) []         = (eac,oac)
   go (eac,oac) [x]        = (x:eac,oac)
   go (eac,oac) (x:y:xs) = go (x:eac,y:oac) xs
```

Consideren las siguientes expresiones

```
head . fst . halve [1..10000000000000]
head . snd . halve [1..10000000000000]
```

UNIVERSIDAD SIMÓN BOLÍVAR

# Dividir una lista en dos

## Una función modular quiere ser reutilizada

```
halve = go ([],[])
  where
   go (eac,oac) []         = (eac,oac)
   go (eac,oac) [x]        = (x:eac,oac)
   go (eac,oac) (x:y:xs) = go (x:eac,y:oac) xs
```

Consideren las siguientes expresiones

```
head . fst . halve [1..10000000000000]
head . snd . halve [1..10000000000000]
```

Ambas *requieren* procesar **toda** la lista antes de producir el resultado.

UNIVERSIDAD SIMÓN BOLÍVAR

# Dividir una lista en dos

## Agregar flojera paga dividendos

```
halve xs = (ping xs, pong xs)

ping []       = []
ping [x]      = [x]
ping (x:_:xs) = x : ping xs

pong []       = []
pong [x]      = []
pong (_:x:xs) = x : pong xs
```

- Ambas construyen la lista un *thunk* a la vez.
- Y ahora

```
head . fst . halve [1..10000000000000]
head . snd . halve [1..10000000000000]
```

pueden responder en tiempo *constante*

UNIVERSIDAD SIMÓN BOLÍVAR

# *Mergesort*

## ¿Cómo afecta a *mergesort*?

- Cuando `msort` se use para ordenar una lista completa, ambas implantaciones de `halve` producirán el mismo resultado.
  - La primera `halve` recorrerá la lista completa para dividirla en dos – tiene que llegar hasta el final de la recursión de cola.
  - La segunda `halve` también, pero puede ir entregando elementos a `merge` a medida que se los va solicitando.

UNIVERSIDAD SIMÓN BOLÍVAR

# *Mergesort*

## ¿Cómo afecta a *mergesort*?

- Cuando `msort` se use para ordenar una lista completa, ambas implantaciones de `halve` producirán el mismo resultado.
  - La primera `halve` recorrerá la lista completa para dividirla en dos – tiene que llegar hasta el final de la recursión de cola.
  - La segunda `halve` también, pero puede ir entregando elementos a `merge` a medida que se los va solicitando.
- La segunda versión es mejor – requiere menos memoria intermedia.

UNIVERSIDAD SIMÓN BOLÍVAR

# *Mergesort*

## ¿Cómo afecta a *mergesort*?

- Cuando `msort` se use para ordenar una lista completa, ambas implantaciones de `halve` producirán el mismo resultado.
  - La primera `halve` recorrerá la lista completa para dividirla en dos – tiene que llegar hasta el final de la recursión de cola.
  - La segunda `halve` también, pero puede ir entregando elementos a `merge` a medida que se los va solicitando.
- La segunda versión es mejor – requiere menos memoria intermedia.
- ¿Cuán modular es `msort`?

```
min = head . msort
```

  - La primera `halve` hace que `min` sea $O(n \cdot log\ n)$.
  - La segunda `halve` hace que `min` sea $O(n)$.

UNIVERSIDAD SIMÓN BOLÍVAR

# *Mergesort*

## ¿Cómo afecta a *mergesort*?

- Cuando `msort` se use para ordenar una lista completa, ambas implantaciones de `halve` producirán el mismo resultado.
  - La primera `halve` recorrerá la lista completa para dividirla en dos – tiene que llegar hasta el final de la recursión de cola.
  - La segunda `halve` también, pero puede ir entregando elementos a `merge` a medida que se los va solicitando.
- La segunda versión es mejor – requiere menos memoria intermedia.
- ¿Cuán modular es `msort`?

```
min = head . msort
```

  - La primera `halve` hace que `min` sea $O(n \cdot log\ n)$.
  - La segunda `halve` hace que `min` sea $O(n)$.

Compruébenlo usando *profiling*

UNIVERSIDAD SIMÓN BOLÍVAR

# La flojera contraataca

- Demasiada flojera puede impactar negativamente el desempeño
  - Los *thunks* requieren memoria en el *heap*.
  - Se mantienen allí "hasta que hacen falta".
  - Si hacen falta *inmediatamente*, ¿para qué esperar?
- Ya conocen la historia de `foldl` y `foldl'` –
  no armar una cascada de sumas sino sumar de una vez.

UNIVERSIDAD SIMÓN BOLÍVAR

# La flojera contraataca

- Demasiada flojera puede impactar negativamente el desempeño
  - Los *thunks* requieren memoria en el *heap*.
  - Se mantienen allí "hasta que hacen falta".
  - Si hacen falta *inmediatamente*, ¿para qué esperar?
- Ya conocen la historia de `foldl` y `foldl'` –
  no armar una cascada de sumas sino sumar de una vez.
- GHC es capaz de optimizar con **Strictness Analysis**
  - Compilar con la opción `-O`.
  - Detecta funciones estrictas o parcialmente estrictas.
  - Genera código que omite los *thunks* para esos casos.

UNIVERSIDAD SIMÓN BOLÍVAR

# La flojera contraataca

- Demasiada flojera puede impactar negativamente el desempeño
  - Los *thunks* requieren memoria en el *heap*.
  - Se mantienen allí "hasta que hacen falta".
  - Si hacen falta *inmediatamente*, ¿para qué esperar?
- Ya conocen la historia de `foldl` y `foldl'` –
  no armar una cascada de sumas sino sumar de una vez.
- GHC es capaz de optimizar con **Strictness Analysis**
  - Compilar con la opción `-O`.
  - Detecta funciones estrictas o parcialmente estrictas.
  - Genera código que omite los *thunks* para esos casos.

Si el lenguaje es perezoso,
¿como puede tener funciones estrictas?

UNIVERSIDAD SIMÓN BOLÍVAR

# Funciones estrictas o semi-estrictas

## Los argumentos cuentan la historia

Una función es *estricta en un argumento* si para evaluar la función **siempre** es necesario evaluar el argumento.

```
null :: [a] -> Bool
null [] = True
null _  = False

cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e
```

- `null` es estricta en su único argumento – es una función estricta.
- `cond` es estricta en el primer argumento, el resto es perezoso – es una función semi-estricta.

UNIVERSIDAD SIMÓN BOLÍVAR

# *Strictness analysis* tiene sus límites

**Sólo detecta cosas obvias decidibles.**

```
suminit :: [Int] -> Int -> Int -> (Int,[Int])
suminit xs n acc =
  case n == 0 of
    True  -> (acc,xs)
    False -> case xs of
      []     -> (acc,[])
      (x:xs) -> suminit xs (n-1) (acc+x)
```

UNIVERSIDAD SIMÓN BOLÍVAR

# *Strictness analysis* tiene sus límites

**Sólo detecta cosas obvias decidibles.**

```
suminit :: [Int] -> Int -> Int -> (Int,[Int])
suminit xs n acc =
  case n == 0 of
    True  -> (acc,xs)
    False -> case xs of
      []     -> (acc,[])
      (x:xs) -> suminit xs (n-1) (acc+x)
```

- GHC puede *garantizar* que es estricta en el *segundo* argumento (`n`) – genera código para evaluar el `(n-1)` de forma estricta.

UNIVERSIDAD SIMÓN BOLÍVAR

# *Strictness analysis* tiene sus límites

**Sólo detecta cosas obvias decidibles.**

```
suminit :: [Int] -> Int -> Int -> (Int,[Int])
suminit xs n acc =
  case n == 0 of
    True  -> (acc,xs)
    False -> case xs of
      []     -> (acc,[])
      (x:xs) -> suminit xs (n-1) (acc+x)
```

- GHC puede *garantizar* que es estricta en el *segundo* argumento (`n`) – genera código para evaluar el `(n-1)` de forma estricta.
- No puede concluir nada *definitivo* acerca de los usos de `acc` – genera código estándar con *thunks* para las sumas.

UNIVERSIDAD SIMÓN BOLÍVAR

# *Strictness analysis* tiene sus límites

**Sólo detecta cosas obvias decidibles.**

```
suminit :: [Int] -> Int -> Int -> (Int,[Int])
suminit xs n acc =
  case n == 0 of
    True  -> (acc,xs)
    False -> case xs of
      []     -> (acc,[])
      (x:xs) -> suminit xs (n-1) (acc+x)
```

- GHC puede *garantizar* que es estricta en el *segundo* argumento (`n`) – genera código para evaluar el `(n-1)` de forma estricta.
- No puede concluir nada *definitivo* acerca de los usos de `acc` – genera código estándar con *thunks* para las sumas.

Nosotros si podemos concluir cosas,
así que nos ponemos estrictos manualmente.

UNIVERSIDAD SIMÓN BOLÍVAR

# Forzar evaluación estricta cuando conviene

## Chitty-chitty, bang-bang!

```
{-# LANGUAGE BangPatterns #-}

suminit :: [Int] -> Int -> Int -> (Int,[Int])
suminit xs !n !acc =
  case n == 0 of
    True  -> (acc,xs)
    False -> case xs of
      []     -> (acc,[])
      (x:xs) -> suminit xs (n-1) (acc+x)
```

- GHC reescribe los *bang patterns* como aplicaciones de `seq`.
- **No** nos interesa `xs` estricto – ¡para poder operar sobre listas infinitas!

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿Y los constructores de datos?

## Eran funciones, ¿no?

- Los constructores de datos son funciones y perezosas.

```
data Pair a b = MakePair a b Int

ghci> :type MakePair
a -> b -> Int -> Pair a b
```

  - No evalúan sus argumentos a menos que sea necesario.
  - *Thunks* inside!

- Contenidos *escalares* estrictos **siempre** son una buena idea

```
data Pair a b = MakePair a b !Int
```

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿Y los constructores de datos?

## Eran funciones, ¿no?

- Los constructores de datos son funciones y perezosas.

```
data Pair a b = MakePair a b Int

ghci> :type MakePair
a -> b -> Int -> Pair a b
```

  - No evalúan sus argumentos a menos que sea necesario.
  - *Thunks* inside!

- Contenidos *escalares* estrictos **siempre** son una buena idea

```
data Pair a b = MakePair a b !Int
```

Para todo lo demás, usar Mucho Cuidado™.

UNIVERSIDAD SIMÓN BOLÍVAR

# A ver...

## What would Buddah do?

```
data Tree = Leaf | Node Int Tree Tree

insert :: Int -> Tree -> Tree
insert x Leaf            = Leaf
insert x (Node y l r)
  | x < y      = Node y (insert x l) r
  | x > y      = Node y l (insert x r)
  | otherwise = Node x l r
```

- Es estricta en

# A ver...

## What would Buddah do?

```
data Tree = Leaf | Node Int Tree Tree

insert :: Int -> Tree -> Tree
insert x Leaf            = Leaf
insert x (Node y l r)
  | x < y      = Node y (insert x l) r
  | x > y      = Node y l (insert x r)
  | otherwise = Node x l r
```

- Es estricta en el segundo argumento.

UNIVERSIDAD SIMÓN BOLÍVAR

# A ver...

## What would Buddah do?

```
data Tree = Leaf | Node Int Tree Tree

insert :: Int -> Tree -> Tree
insert x Leaf            = Leaf
insert x (Node y l r)
  | x < y      = Node y (insert x l) r
  | x > y      = Node y l (insert x r)
  | otherwise = Node x l r
```

- Es estricta en el segundo argumento.
- El segundo argumento podría ser un *thunk* inmenso...
- ...que siempre es necesario evaluar – es un árbol de búsqueda.

UNIVERSIDAD SIMÓN BOLÍVAR

# A ver...

## What would Buddah do?

```
data Tree = Leaf | Node Int Tree Tree

insert :: Int -> Tree -> Tree
insert x Leaf            = Leaf
insert x (Node y l r)
  | x < y      = Node y (insert x l) r
  | x > y      = Node y l (insert x r)
  | otherwise = Node x l r
```

- Es estricta en el segundo argumento.
- El segundo argumento podría ser un *thunk* inmenso...
- ...que siempre es necesario evaluar – es un árbol de búsqueda.
- Vale la pena que el tipo de datos sea completamente estricto.

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿En qué quedamos?

## Procrastination pro-tips

- Estructuras arbóreas **finitas** *suelen* usar sub-estructuras estrictas

```
data Set a = Tip | Bin !Int a !(Set a) !(Set a)
```

  - Conviene tener el tamaño precalculado – por eso el `Int`
  - Conviene tener la *estructura* de subárboles precalculada.
  - Los *contenidos* se mantienen perezosos.

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿En qué quedamos?

## Procrastination pro-tips

- Estructuras arbóreas **finitas** *suelen* usar sub-estructuras estrictas

```
data Set a = Tip | Bin !Int a !(Set a) !(Set a)
```

  - Conviene tener el tamaño precalculado – por eso el `Int`
  - Conviene tener la *estructura* de subárboles precalculada.
  - Los *contenidos* se mantienen perezosos.

- `$!` es la versión estricta de `$`.
  - Si `f x` utiliza una composición `h (g x)` que no es estricta en `x` vale la pena `h $! g x` – eso simplifica al máximo el argumento de `h`.
  - Caso particular – procure inyectar valores estrictos en un `Monad`.

```
return $! foo x
```

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

- Sistema de Tipos de Haskell extremadamente flexible
  - Tipos escalares nativos.
  - Equivalencia estructural – `type`
  - Equivalencia por nombres – `data` y `newtype`.

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

- Sistema de Tipos de Haskell extremadamente flexible
  - Tipos escalares nativos.
  - Equivalencia estructural – `type`
  - Equivalencia por nombres – `data` y `newtype`.
- Diseñar tipos de datos para lenguajes funcionales requiere tomar en cuenta y balancear astucia de representación y pereza.
  - One does not simply write functional data structures! – lean lo que Chris Okasaki tiene que decir al respecto.
  - La `base` del lenguaje incluye tipos optimizados.
  - Las librerías de la plataforma también.

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

- Sistema de Tipos de Haskell extremadamente flexible
  - Tipos escalares nativos.
  - Equivalencia estructural – `type`
  - Equivalencia por nombres – `data` y `newtype`.
- Diseñar tipos de datos para lenguajes funcionales requiere tomar en cuenta y balancear astucia de representación y pereza.
  - One does not simply write functional data structures! – lean lo que Chris Okasaki tiene que decir al respecto.
  - La `base` del lenguaje incluye tipos optimizados.
  - Las librerías de la plataforma también.

No reinventar la rueda ovalada ni cuadrada.
Veamos con qué contamos.

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

## data vs. newtype

Al definir tipos con **un** constructor y **un** campo:

- Podríamos usar un tipo algebráico completo

```
data Foo a = Foo a
```

- Podríamos usar un tipo sinónimo

```
newtype Foo a = Foo a
```

El constructor del `newtype` se elimina
a tiempo de **compilación** – más eficiente.

UNIVERSIDAD SIMÓN BOLÍVAR

# Prefiera constructores simples

## GHC hace *unpacking*

- Una tupla tiene un constructor simple `(,)` – dos valores, una tupla.
- Cuando se compila la función

```
f (x,y) = {- whatever -}
```

GHC genera código similar a

```
f z = case z of (x,y) -> f' x y

f' x y = {- whatever -}
```

  - `f` es el *wrapper* – *inline* en todos lados.
  - `f'` es el *worker* – hace el trabajo sin tuplas.

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

## *Boxed* vs. *Unboxed* – via `GHC.Exts`

- Tipos **lifted** – *boxed data*
  - Evaluados, parcialmente evaluados (*thunk*) o $\perp$ (`undefined`).
  - Hace falta una indirección.
    - El contenedor – "hey, soy del tipo `Foo`"
    - El contenido – el dato o el *thunk* que lo evaluará.

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

## *Boxed* vs. *Unboxed* – via `GHC.Exts`

- Tipos **lifted** – *boxed data*
  - Evaluados, parcialmente evaluados (*thunk*) o $\perp$ (`undefined`).
  - Hace falta una indirección.
    - El contenedor – "hey, soy del tipo `Foo`"
    - El contenido – el dato o el *thunk* que lo evaluará.
- Tipos **unlifted** – *unboxed data*
  - Sólo pueden estar evaluados completamente – imposible representar $\perp$.
  - No se pueden pasar a funciones polimórficas.
  - No se pueden usar como contenido de un `newtype`.
  - No pueden existir como definiciones *top-level*.
  - Si son escalares, no requieren indirección – ¡mejor desempeño!
  - Aumenta la densidad de los contenedores.
  - Disponibles al *compilar* usando `-fglasgow-exts`.

UNIVERSIDAD SIMÓN BOLÍVAR

# Usando tipos *unboxed*

```
{-# LANGUAGE MagicHash #-}
import GHC.Exts

n = I# 42#
c = I# (ord# '*'#)

main = print c >> (print $ show n)
```

- `42#` es un `Int#` – 32 bits sin adornos.
- No puedo definir `n = 42#`.
- No se puede pasar a `show`, porque es polimórfica.
- `GHC.Exts` provee mecanismos para volver a levantar el tipo – en este caso `I#` produce el `Int` habitual.

Se usan *internamente* en librerías optimizadas,
o para interactuar con librerías externas en C.

# Poca gente quiere escribir...

```
fac# n# = if n# ># 0# then n# *# fac# (n# -# 1#)
                      else 1#

unboxInt (I# i#) = i#

main = do
  n <- (read . head) `liftM` getArgs
  print $ I# (fac# (unboxInt n))
```

- Los *unboxed* generalmente terminan en registros.
- Pero hay demasiado "agitar de manos" para hacerlos cómodos en código general – además, el compilador lo hace cuando puede.

Son usados *internamente* en librerías optimizadas,
o para interactuar con librerías externas.

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

## Tipos numéricos

- `Integer` tiene precisión arbitraria costosa en espacio – use las representaciones directas que aprovechan la máquina.
  - `Data.Int` (con signo) – `Int8`, `Int16`, `Int32` e `Int64`.
  - `Data.Word` (sin signo) – `Word8`, `Word16`, `Word32` y `Word64`.
  - `fromIntegral` del `Prelude` optimizada para convertir entre ellos.
- `Data.Bits` – operaciones bit a bit.
- Use `Double` en lugar de `Float`.
  - Todos los FPU en el fondo usan `Double`.
  - Se reduce el error numérico.
  - Si la máquina es de 64 bits, ambos ocupan 64 bits.

UNIVERSIDAD SIMÓN BOLÍVAR

# Funciones y Módulos

## Sobrecarga no siempre es tu amiga

- Polimorfismo paramétrico se basa en despacho dinámico

```
(Num a) => a -> a -> a
```

  - Requiere un *tercer* argumento **oculto** – tabla de despacho.
  - La tabla particular se incorpora a tiempo de ejecución.
- Use firmas explícitas
  - Evita sobrecarga sin necesidad (`Integral` vs. `Int`).
  - ¿Quiere que el compilador le **obligue** a usar firmas? – `-fwarn-missing-signatures`.

UNIVERSIDAD SIMÓN BOLÍVAR

# Módulos

- Exporte **explícitamente** las funciones, porque eso:
  - Activa *inlining* para funciones que son usadas una sola vez.
  - Activa *inlining* de funciones privadas al módulo.
  - Detecta y elimina código inalcanzable.
  - Minimiza el riesgo de polimorfismo excesivo.
  - Otorga libertad al compilador en la generación de la secuencia de llamada para funciones privadas.

UNIVERSIDAD SIMÓN BOLÍVAR

# Más tipos de Datos

## Alternativas a `String`

- `String` en realidad es `[Char]` – y cada `Char` es *boxed*.
- `Data.Text` – texto UTF-16 empaquetado.
- `Data.ByteString` – bytes empaquetados.
  - Vectores de `Word8` – no son «strings».
  - `pack` y `unpack` – pasar desde y hacia `[Word8]`
  - Funciones optimizadas para manipularlos – `take`, `fold`, `map`, ...
  - Importe calificado – evite conflictos con `Prelude` y `Data.List`.

UNIVERSIDAD SIMÓN BOLÍVAR

# Más tipos de Datos

## Alternativas a `String`

- `String` en realidad es `[Char]` – y cada `Char` es *boxed*.
- `Data.Text` – texto UTF-16 empaquetado.
- `Data.ByteString` – bytes empaquetados.
  - Vectores de `Word8` – no son «strings».
  - `pack` y `unpack` – pasar desde y hacia `[Word8]`
  - Funciones optimizadas para manipularlos – `take`, `fold`, `map`, ...
  - Importe calificado – evite conflictos con `Prelude` y `Data.List`.
- Según el juego de caracteres a procesar
  - `Data.BysteString.Char8` – ASCII, Latin-1, Unicode Basic Latin
  - `Data.ByteString.UTF8` – Unicode UTF-8

UNIVERSIDAD SIMÓN BOLÍVAR

# Más tipos de Datos

## Alternativas a `String`

- `String` en realidad es `[Char]` – y cada `Char` es *boxed*.
- `Data.Text` – texto UTF-16 empaquetado.
- `Data.ByteString` – bytes empaquetados.
  - Vectores de `Word8` – no son «strings».
  - `pack` y `unpack` – pasar desde y hacia `[Word8]`
  - Funciones optimizadas para manipularlos – `take`, `fold`, `map`, ...
  - Importe calificado – evite conflictos con `Prelude` y `Data.List`.
- Según el juego de caracteres a procesar
  - `Data.BysteString.Char8` – ASCII, Latin-1, Unicode Basic Latin
  - `Data.ByteString.UTF8` – Unicode UTF-8
- Ambas disponen de versiones perezosas en `Data.ByteString.Lazy` capaces de procesar cadenas que no caben en memoria.

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿Y entonces?

- `ByteString` – para datos binarios «crudos»
  - No tienen ninguna interpretación particular – data is data.
  - Leer y escribir desde y hacia archivos – *packed*
  - Nunca presentados al usuario en ese formato.
- `Text` – datos textuales «legibles»
  - Interpretados como texto – UTF-8 obviamente...
  - Para recibir o presentar al usuario.
  - Operaciones optimizadas para fusión – sólo un `Text` por operación.
- `String` – sólo para enseñar Haskell

UNIVERSIDAD SIMÓN BOLÍVAR

# Alternativas a String

## Una comparación

- String – [Char]

```
import System.IO
main = readFile "/usr/share/dict/words"
       >>= putStrLn . last . lines
```

# Alternativas a String

## Una comparación

- String – [Char]

```
import System.IO
main = readFile "/usr/share/dict/words"
       >>= putStrLn . last . lines
```

- Data.Text – empaquetar, procesar, desempaquetar.

```
import qualified Data.Text as T
main = readFile "/usr/share/dict/words"
       >>=
       putStr . T.unpack . last . T.lines . T.pack
```

UNIVERSIDAD SIMÓN BOLÍVAR

# Alternativas a String

## Una comparación

- String – [Char]

```
import System.IO
main = readFile "/usr/share/dict/words"
       >>= putStrLn . last . lines
```

- Data.Text – empaquetar, procesar, desempaquetar.

```
import qualified Data.Text as T
main = readFile "/usr/share/dict/words"
       >>=
       putStr . T.unpack . last . T.lines . T.pack
```

- Data.ByteString.Char8 – todo es especial.

```
import qualified Data.ByteString.Char8 as B
main = B.readFile "/usr/share/dict/words"
       >>= B.putStr . last . B.lines
```

UNIVERSIDAD SIMÓN BOLÍVAR

# Use all the string types!

## . . . porque a veces hace falta necesitan

- Usar las funciones de conversión entre tipos – engorroso.
- «Sobrecargar» las cadenas constantes – Wait, what?

```
{-# LANGUAGE OverloadedStrings #-}
```

  - Las cadenas literales serán sobrecargadas – `IString a`
  - Las librerías proveen `fromString :: IString a => a`

- Cadenas literales se convierten al tipo adecuado según el contexto en que sean utilizadas.

UNIVERSIDAD SIMÓN BOLÍVAR

# Funciona compilando y hasta en GHCi

```
> import qualified Data.ByteString.Char8 as BS
> import qualified Data.Text.IO as T
> :set -XOverloadedStrings
> putStrLn "hello"
hello
> BS.putStrLn "hello"
hello
> T.putStrLn "hello"
hello
```

1. `"hello"` es `String` – nada especial.
2. `"hello"` es `ByteString` – `fromString` implícito.
3. `"hello"` es `Texxt` – `fromString` implícito.

UNIVERSIDAD SIMÓN BOLÍVAR

# Tipos de Datos

## Listas vs. Secuencias

- Listas convencionales (`[a]`)
  - $O(1)$ sobre la cabeza de la lista – `(:)` y `head`.
  - $O(i)$ para acceder al $i-$ésimo elemento – `(!!)`.
  - $O(n)$ para acceder al último elemento – `last` o `(++)`.
  - Ideales para representar pilas o *streams*.
- `Data.Sequence` (2-3 *finger trees*)
  - $O(1)$ sobre ambos extremos – `(<|)` y `(|>)`.
  - $O(log(min(i, n-i)))$ para el $i-$ésimo elemento – `index`, `take` y `drop`.
  - $O(log(min(n, m)))$ para concatenar – `(><)`.
  - Ideales para representar colas o deques.
- En el `Monad Writer` *siempre* se usa `Data.Sequence`.

UNIVERSIDAD SIMÓN BOLÍVAR

# Contenedores

## Estructuras de datos para comer con cubiertos

- Conjuntos
  - `Data.Set` – árboles balanceados.
  - `Data.IntSet` – Big-endian Patricia Trees (Radix Trie)

UNIVERSIDAD SIMÓN BOLÍVAR

# Contenedores

## Estructuras de datos para comer con cubiertos

- Conjuntos
  - `Data.Set` – árboles balanceados.
  - `Data.IntSet` – Big-endian Patricia Trees (Radix Trie)
- Diccionarios
  - `Data.Map` – de clave a valor (árboles balanceados).
  - `Data.IntMap` – de enteros a valores (Patricia Trees).
  - Versiones perezosas o estrictas en los *valores*.

UNIVERSIDAD SIMÓN BOLÍVAR

# Contenedores

## Estructuras de datos para comer con cubiertos

- Conjuntos
  - `Data.Set` – árboles balanceados.
  - `Data.IntSet` – Big-endian Patricia Trees (Radix Trie)
- Diccionarios
  - `Data.Map` – de clave a valor (árboles balanceados).
  - `Data.IntMap` – de enteros a valores (Patricia Trees).
  - Versiones perezosas o estrictas en los *valores*.
- `Data.Graph` – grafos y sus algoritmos.

UNIVERSIDAD SIMÓN BOLÍVAR

# Contenedores

## Estructuras de datos para comer con cubiertos

- Conjuntos
  - `Data.Set` – árboles balanceados.
  - `Data.IntSet` – Big-endian Patricia Trees (Radix Trie)
- Diccionarios
  - `Data.Map` – de clave a valor (árboles balanceados).
  - `Data.IntMap` – de enteros a valores (Patricia Trees).
  - Versiones perezosas o estrictas en los *valores*.
- `Data.Graph` – grafos y sus algoritmos.
- `Data.Tree` – Rose Trees

UNIVERSIDAD SIMÓN BOLÍVAR

# Contenedores

## Estructuras de datos para comer con cubiertos

- Conjuntos
  - `Data.Set` – árboles balanceados.
  - `Data.IntSet` – Big-endian Patricia Trees (Radix Trie)
- Diccionarios
  - `Data.Map` – de clave a valor (árboles balanceados).
  - `Data.IntMap` – de enteros a valores (Patricia Trees).
  - Versiones perezosas o estrictas en los *valores*.
- `Data.Graph` – grafos y sus algoritmos.
- `Data.Tree` – Rose Trees
- `Data.Sequence` – 2-3 finger trees.

UNIVERSIDAD SIMÓN BOLÍVAR

# Programas concurrentes

- No compile `-threaded` para máquinas con un sólo núcleo.
- No use el hilo principal para hacer el trabajo.
  - La comunicación entre el hilo principal y el resto es mucho más lenta que entre los hilos subordinados.
  - El hilo principal corresponde a un hilo del sistema operativo y tiene un costo mayor de mantenimiento.

UNIVERSIDAD SIMÓN BOLÍVAR

# Quiero saber más...

- Documentación de `Data.Bits`
- Documentación de `Data.Int`
- Documentación de `Data.Word`
- Documentación de `Data.Text`
- Documentación de `Data.ByteString`
- Documentación del paquete `containers`
- Documentación de `GHC` en relación a *unboxed types*
- Purely Functional Data Structures
  Okasaki, 1996

UNIVERSIDAD SIMÓN BOLÍVAR