Programación Funcional Avanzada

Marcos Viera Alberto Pardo

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ByteStrings

Strings en Haskell

$$\textbf{type}\ \mathit{String} = [\mathit{Char}]$$

Strings en Haskell

$$\textbf{type}\ \mathit{String} = [\mathit{Char}]$$

Strings implementados con listas

- convenientes de usar
- ineficientes

Strings en Haskell

$$\textbf{type}\ \mathit{String} = [\mathit{Char}]$$

Strings implementados con listas

- convenientes de usar
- ineficientes

¿Qué tal usar arreglos?

Ejemplo

Calcular una función de hash a los caracteres alfabéticos de un archivo:

$$
\begin{aligned}
&\mathit{return} \circ \mathit{foldl'}\ \mathit{hash}\ 5381 \circ \mathit{map}\ \mathit{toLower} \circ \mathit{filter}\ \mathit{isAlpha} \\
&\qquad =\!\!\ll \mathit{readFile}\ f \\
&\quad \textbf{where}\ \mathit{hash}\ h\ c = h * 33 + \mathit{ord}\ c
\end{aligned}
$$

dados

- $\mathit{readFile} :: \mathit{FilePath} \to \mathit{IO}\ \mathit{String}$
- $(=\!\!\ll) :: (\mathit{Monad}\ m) \Rightarrow (a \to m\ b) \to m\ a \to m\ b$

Ejemplo

Calcular una función de hash a los caracteres alfabéticos de un archivo:

$$\begin{aligned} &\mathit{return} \circ \mathit{foldl'}\ \mathit{hash}\ 5381 \circ \mathit{map}\ \mathit{toLower} \circ \mathit{filter}\ \mathit{isAlpha} \\ &\qquad =\!\!\ll \mathit{readFile}\ f \\ &\quad \textbf{where}\ \mathit{hash}\ h\ c = h * 33 + \mathit{ord}\ c \end{aligned}$$

dados

- $\mathit{readFile} :: \mathit{FilePath} \to \mathit{IO}\ \mathit{String}$
- $(=\!\!\ll) :: (\mathit{Monad}\ m) \Rightarrow (a \to m\ b) \to m\ a \to m\ b$

Strings se suelen usar como flujos (streams)

- el string se recorre, modifica y escribe
- posiblemente varias veces

Ejemplo

Calcular una función de hash a los caracteres alfabéticos de un archivo:

$$\begin{aligned} &\mathit{return} \circ \mathit{foldl'}\ \mathit{hash}\ 5381 \circ \mathit{map}\ \mathit{toLower} \circ \mathit{filter}\ \mathit{isAlpha} \\ &\qquad =\!\!\ll \mathit{readFile}\ f \\ &\quad \textbf{where}\ \mathit{hash}\ h\ c = h * 33 + \mathit{ord}\ c \end{aligned}$$

dados

- $\mathit{readFile} :: \mathit{FilePath} \to \mathit{IO}\ \mathit{String}$
- $(=\!\!\ll) :: (\mathit{Monad}\ m) \Rightarrow (a \to m\ b) \to m\ a \to m\ b$

Strings se suelen usar como flujos (streams)

- el string se recorre, modifica y escribe
- posiblemente varias veces

Objetivo: optimizaciones sin perder el estilo declarativo

ByteString

Reimplementa eficientemente la mayoría de las funciones de lista

```
import Data.ByteString.Lazy as B
return ∘ B.foldl' hash 5381 ∘ B.map toLower ∘ B.filter isAlpha
        =≪ B.readFile f
  where hash h c = h ∗ 33 + ord c
```

ByteString

Reimplementa eficientemente la mayoría de las funciones de lista

```
import Data.ByteString.Lazy as B
return ∘ B.foldl' hash 5381 ∘ B.map toLower ∘ B.filter isAlpha
        =≪ B.readFile f
  where hash h c = h ∗ 33 + ord c
```

Estructura eficiente

- **Data.ByteString**: arreglo de caracteres (unboxed)
  data *ByteString* = *BS* ! (*ForeignPtr Word8*) ! *Int* ! *Int*

ByteString

Reimplementa eficientemente la mayoría de las funciones de lista

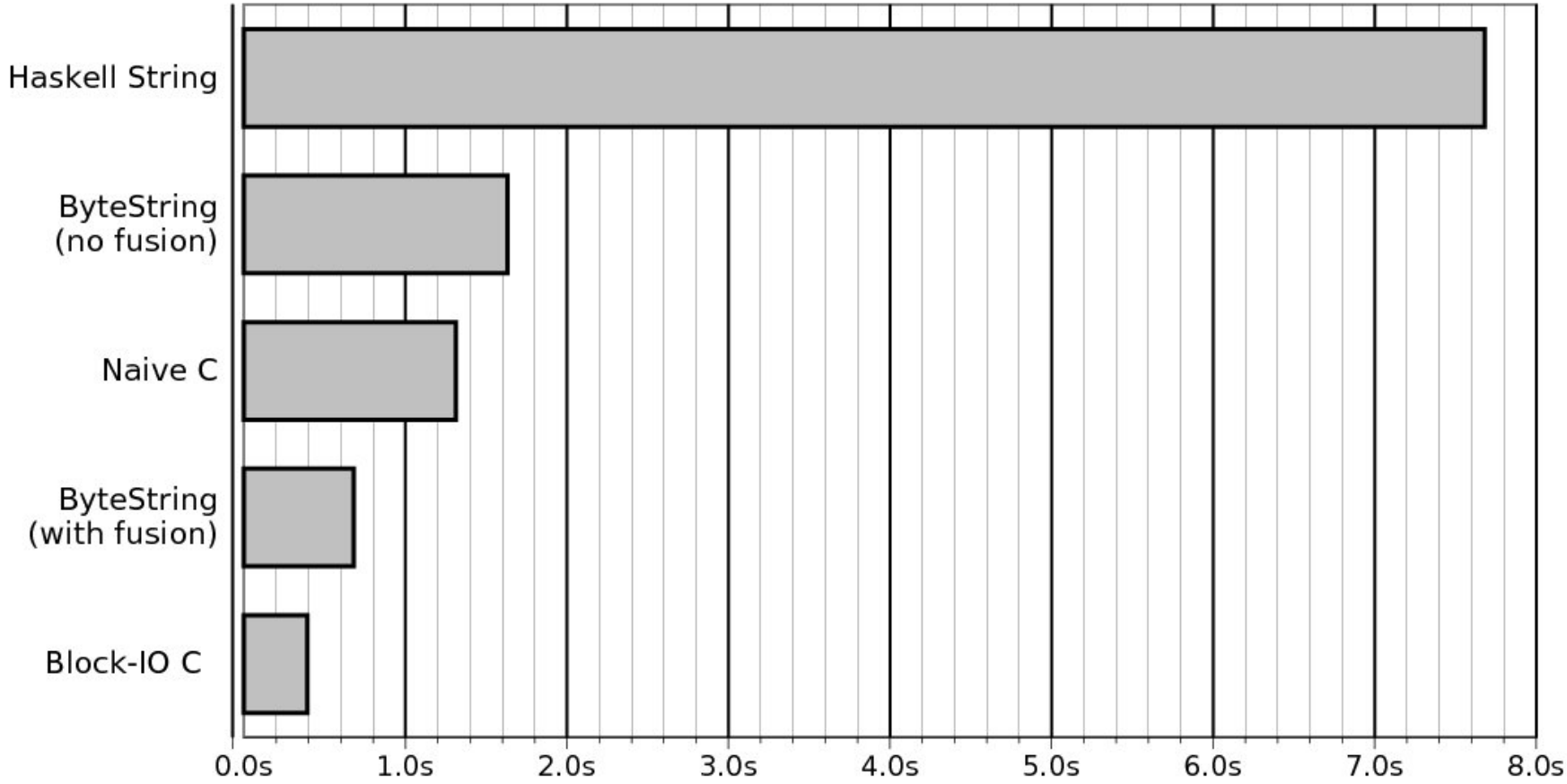
```
import Data.ByteString.Lazy as B
return ∘ B.foldl' hash 5381 ∘ B.map toLower ∘ B.filter isAlpha
        =≪ B.readFile f
  where hash h c = h ∗ 33 + ord c
```

Estructura eficiente

- **Data.ByteString**: arreglo de caracteres (unboxed)
  data *ByteString* = *BS* ! (*ForeignPtr Word8*) ! *Int* ! *Int*
- **Data.ByteString.Lazy**: lista de trozos estrictos
  data *ByteString* = *Empty* | *Chunk* ! *S.ByteString ByteString*

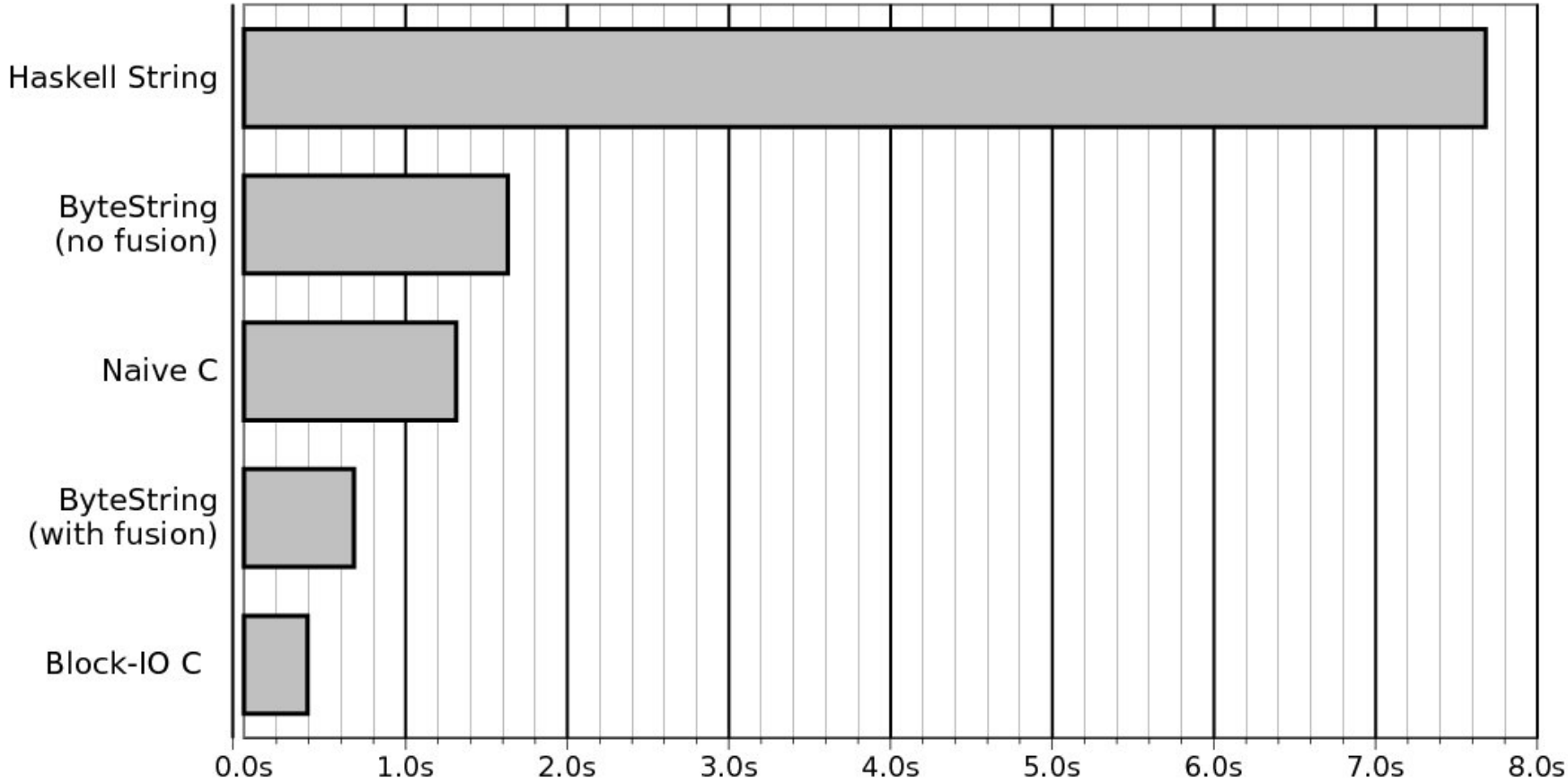
Eficiencia

Comparando...



Eficiencia

Comparando...



Uso de array fusion para disminuir

- recorridas
- copias de la estructura de datos

Fusión y Deforestación

Fusión: habilidad de juntar múltiples recorridas sobre una estructura de datos
Deforestación: eliminación de estructuras de datos intermedias

Fusión y Deforestación

Fusión: habilidad de juntar múltiples recorridas sobre una estructura de datos
Deforestación: eliminación de estructuras de datos intermedias

Existe bastante teoría para el caso de listas

Fusión y Deforestación

Fusión: habilidad de juntar múltiples recorridas sobre una estructura de datos
Deforestación: eliminación de estructuras de datos intermedias

Existe bastante teoría para el caso de listas

El paper "Rewriting Haskell Strings" presenta una nueva forma de hacer fusión de arreglos/flujos

Fusión y Deforestación

Fusión: habilidad de juntar múltiples recorridas sobre una estructura de datos
Deforestación: eliminación de estructuras de datos intermedias

Existe bastante teoría para el caso de listas

El paper "Rewriting Haskell Strings" presenta una nueva forma de hacer fusión de arreglos/flujos

Idea

- leer el arreglo para producir un flujo
- procesar los elementos transformando el flujo
- escribir el flujo resultante en un nuevo arreglo

unfoldr

Construye una lista a partir de un valor semilla

$$
\begin{aligned}
&\mathit{unfoldr} :: (s \to \mathit{Maybe}\ (a, s)) \to s \to [a] \\
&\mathit{unfoldr}\ \mathit{next}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \quad \to [\,] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ (x, r) \to x : \mathit{unfoldr}\ \mathit{next}\ r
\end{aligned}
$$

unfoldr

Construye una lista a partir de un valor semilla

$$
\begin{aligned}
&\mathit{unfoldr} :: (s \to \mathit{Maybe}\ (a, s)) \to s \to [a] \\
&\mathit{unfoldr}\ \mathit{next}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \quad \to [\,] \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ (x, r) \to x : \mathit{unfoldr}\ \mathit{next}\ r
\end{aligned}
$$

ejemplos:

$$
\begin{aligned}
\mathit{repeat} &= \mathit{unfoldr}\ (\lambda x \to \mathit{Just}\ (x, x)) \\
\mathit{replicate}\ n\ x &= \mathit{unfoldr}\ (\lambda n \to \textbf{if}\ n == 0\ \textbf{then}\ \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{Just}\ (x, n-1))\ n \\
\mathit{enumFromTo}\ b\ e &= \mathit{unfoldr}\ (\lambda b \to \textbf{if}\ b > e\ \textbf{then}\ \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{Just}\ (b, b+1))\ b
\end{aligned}
$$

Representando strings como flujos

Se utiliza unfoldr como representación

- abstraer de la representación concreta
- permitir diferentes patrones de acceso

Representando strings como flujos

Se utiliza unfoldr como representación

- abstraer de la representación concreta
- permitir diferentes patrones de acceso

$$
\begin{aligned}
&\mathit{unfoldr} :: (s \to \mathit{Maybe}\ (a, s)) \to s \to [a] \\
&\mathit{unfoldr}\ \mathit{next}\ s = \mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \quad \to [\,] \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ (x, r) \to x : \mathit{unfoldr}\ \mathit{next}\ r
\end{aligned}
$$

Representando strings como flujos

Se utiliza unfoldr como representación

- abstraer de la representación concreta
- permitir diferentes patrones de acceso

$$
\begin{aligned}
&\mathit{unfoldr} :: (s \to \mathit{Maybe}\ (a, s)) \to s \to [a] \\
&\mathit{unfoldr}\ \mathit{next}\ s = \mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \quad \to [\,] \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ (x, r) \to x : \mathit{unfoldr}\ \mathit{next}\ r
\end{aligned}
$$

$$
\mathbf{data}\ \mathit{Stream}\ s = \mathit{Stream}\ (s \to \mathit{Maybe}\ (\mathit{Word8}, s))\ s
$$

Representando strings como flujos (2)

$$\textbf{data}\ \mathit{Stream}\ s = \mathit{Stream}\ (s \to \mathit{Maybe}\ (\mathit{Word8}, s))\ s$$

Problemas

- no estamos interesados en el tipo s, sólo nos importa aplicar la función a la semilla
- por razones de eficiencia es bueno saber el largo del string
- también es bueno tener una tercera opción:
  - fin de string
  - próximo caracter

Representando strings como flujos (2)

$$\textbf{data}\ \mathit{Stream}\ s = \mathit{Stream}\ (s \to \mathit{Maybe}\ (\mathit{Word8}, s))\ s$$

Problemas

- no estamos interesados en el tipo s, sólo nos importa aplicar la función a la semilla
- por razones de eficiencia es bueno saber el largo del string
- también es bueno tener una tercera opción:
  - fin de string
  - próximo caracter
  - no hacer nada

Representando strings como flujos (3)

$$\textbf{data}\ \mathit{Stream} = \forall\ s.\mathit{Stream}\ (s \to \mathit{Step}\ s)\ s\ \mathit{Int}$$

$$\begin{aligned}\textbf{data}\ \mathit{Step}\ s = {} & \mathit{Done} \\ & \mid \mathit{Yield}\ \mathit{Word8}\ s \\ & \mid \mathit{Skip}\ s\end{aligned}$$

Representando strings como flujos (3)

$$\textbf{data}\ \mathit{Stream} = \forall\ s.\mathit{Stream}\ (s \to \mathit{Step}\ s)\ s\ \mathit{Int}$$

$$\begin{aligned}\textbf{data}\ \mathit{Step}\ s = {} & \mathit{Done} \\ & \mid \mathit{Yield}\ \mathit{Word8}\ s \\ & \mid \mathit{Skip}\ s\end{aligned}$$

Stream es un tipo **existencial**

Representando strings como flujos (3)

$$\textbf{data } \mathit{Stream} = \forall\ s.\mathit{Stream}\ (s \to \mathit{Step}\ s)\ s\ \mathit{Int}$$

$$\begin{aligned}\textbf{data } \mathit{Step}\ s = {} & \mathit{Done} \\ & \mid \mathit{Yield}\ \mathit{Word8}\ s \\ & \mid \mathit{Skip}\ s\end{aligned}$$

Stream es un tipo **existencial**

El tipo del constructor es:

$$\mathit{Stream} :: \forall\ s.(s \to \mathit{Step}\ s) \to s \to \mathit{Int} \to \mathit{Stream}$$

el tipo s no aparece en el tipo resultado

Representando strings como flujos (3)

$$\textbf{data } \mathit{Stream} = \forall\ s.\mathit{Stream}\ (s \to \mathit{Step}\ s)\ s\ \mathit{Int}$$

$$\begin{aligned}\textbf{data } \mathit{Step}\ s = {} & \mathit{Done} \\ & \mid \mathit{Yield}\ \mathit{Word8}\ s \\ & \mid \mathit{Skip}\ s\end{aligned}$$

Stream es un tipo **existencial**

El tipo del constructor es:

$$\mathit{Stream} :: \forall\ s.(s \to \mathit{Step}\ s) \to s \to \mathit{Int} \to \mathit{Stream}$$

el tipo s no aparece en el tipo resultado

El tipo *Step* representa el resultado, para cada elemento, de la función de paso

- *Done*: se terminó de procesar el flujo
- *Yield*: produce un elemento transformado de tipo a
- *Skip*: ignora ese elemento

ByteString $\rightarrow$ *Stream*

$$
\begin{aligned}
&\mathit{readUp} :: \mathit{ByteString} \rightarrow \mathit{Stream} \\
&\mathit{readUp}\ s = \mathit{Stream}\ \mathit{next}\ 0\ n \\
&\quad \textbf{where} \\
&\qquad n \qquad\qquad\qquad\quad = \mathit{length}\ s \\
&\qquad \mathit{next}\ i \mid i < n \quad = \mathit{Yield}\ (s\ !\ i)\ (i + 1) \\
&\qquad\qquad\ \mid \mathit{otherwise} = \mathit{Done}
\end{aligned}
$$

Stream → *ByteString*

```
writeUp :: Stream → ByteString
writeUp (Stream next s n) = listArray (0, n − 1)
                                      (unfoldStream next s)
  where
    unfoldStream next s =
      case next s of
        Done      → []
        Yield x r → x : unfoldStream next r
        Skip r    → unfoldStream next r
```

Modificando un flujo (ejemplo *map*)

$$
\begin{aligned}
&map :: (Word8 \to Word8) \to ByteString \to ByteString \\
&map\ f = writeUp.mapS\ f.readUp \\
&mapS :: (Word8 \to Word8) \to Stream \to Stream \\
&mapS\ f\ (Stream\ next\ s\ n) = Stream\ next'\ s\ n \\
&\quad \textbf{where} \\
&\qquad next'\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\qquad\qquad Done \to Done \\
&\qquad\qquad\qquad Yield\ x\ r \to Yield\ (f\ x)\ r \\
&\qquad\qquad\qquad Skip\ r \to Skip\ r
\end{aligned}
$$

Fusión de flujos

readUp/writeUp fusion

$$readUp.writeUp \equiv id$$

Fusión de flujos

readUp/writeUp fusion

$$readUp.writeUp \equiv id$$

En el ejemplo:

$$\begin{aligned}
&map\ f.map\ g \\
&\equiv \{ Definicion\ de\ map, dos\ veces \} \\
&writeUp.mapS\ f.readUp.writeUp.mapS\ g.readUp \\
&\equiv \{ readUp\ /\ writeUp\ fusion\ usando\ reglas\ de\ reescritura\ de\ GHC \} \\
&writeUp.mapS\ f.mapS\ g.readUp \\
&\equiv \{ unfolding\ de\ funciones\ no\ recursivas\ de\ GHC \} \\
&writeUp.mapS\ (f.g).readUp
\end{aligned}$$

Reglas de reescritura de GHC

Optimizador de GHC permite agregar reglas

Reglas de reescritura de GHC

Optimizador de GHC permite agregar reglas

La regla:

$$readUp.writeUp \equiv id$$

Reglas de reescritura de GHC

Optimizador de GHC permite agregar reglas

La regla:

$$readUp.writeUp \equiv id$$

puede ser escrita, usando pragmas:

```
{-# RULES
    "readUp/writeUp"
    forall x. (readUp (writeUp x)) = x
#-}
```

Reglas de reescritura de GHC

Optimizador de GHC permite agregar reglas

La regla:

$$readUp.writeUp \equiv id$$

puede ser escrita, usando pragmas:

```
{-# RULES
    "readUp/writeUp"
    forall x. (readUp (writeUp x)) = x
#-}
```

Se chequea el tipado de las reglas, pero el usuario es responsable por su correctitud.