Temario

Bibliografía

✓ *Razonando con Haskell. Un curso sobre programación funcional.* Blas Ruiz, Francisco Gutiérrez, Pablo Guerrero y José Gallardo. Thomson, 2004. (`http://www.lcc.uma.es/RazonandoConHaskell`)

✓ *Introduction to Functional Programming using Haskell.* Richard Bird. Prentice Hall, 1998.

✓ *The Haskell School of Expression. Learning Functional Programming through multimedia.* Paul Hudak. Cambridge University Press, 2000.

✓ *Haskell. The Craft of Functional Programming.* Simon Thompson. Addison-Wesley, 1999.

Profesor

Pepe Gallardo.

Despacho 3.2.50.

`pepeg@lcc.uma.es`

Web asignatura

`http://www.lcc.uma.es/~pepeg/declarativa`

Tema 1. Introducción y semántica operacional

1.1 Programación Funcional

- ✓ El estilo de programación tradicional (*imperativo*) está basado en sentencias
- ✓ Problemas:
  - ◇ Muy bajo nivel
  - ◇ Programas difíciles de entender
  - ◇ Difícil verificación de la corrección: *Software falla*
- ✓ El estilo funcional está basado en expresiones:
  - ◇ Un programa es un conjunto de definiciones de *funciones matemáticas*
  - ◇ La ejecución de un programa es la evaluación de una expresión
- ✓ Ventajas:
  - ◇ Permite escribir programas claros, concisos y con alto nivel de abstracción
  - ◇ Soporta componentes *Software* reusables
  - ◇ Facilita el uso de la verificación formal

Usaremos *Haskell 98*

`http://haskell.org`

1.2 El lenguaje Haskell

Haskell es un lenguaje funcional

✓ Puro:

- ◇ Una misma expresión denota siempre el mismo valor (*Transparencia referencial*).
- ◇ Verificación formal relativamente fácil.

✓ No estricto:

- ◇ El orden utilizado para reducir expresiones es normal.
- ◇ Las implementaciones de Haskell suelen usar *evaluación perezosa*.
- ◇ Permite trabajar con estructuras infinitas.

✓ Fuertemente tipado:

- ◇ Cada elemento tiene un *tipo*.
- ◇ Se usa para comprobar el uso consistente de los elementos.
- ◇ Usos inconsistentes dan lugar a errores de tipo (en tiempo de compilación).
- ◇ Muchos errores se detectan durante compilación.

1.3 La notación Currificada

Aplicación de funciones

✓ En *Matemáticas* la aplicación de funciones es denotada usando paréntesis:

$f\ (a, b)\ +\ c\ \times\ d$ aplicar la función f a los argumentos a y b

✓ En *Haskell* la aplicación de funciones es denotada usando espacios (notación *currificada*):

$f\ a\ b\ +\ c * d$ aplicar la función f a los argumentos a y b

✓ En *Haskell* la aplicación de funciones tiene prioridad máxima:

$g\ a\ +\ b$ significa $(g\ a) + b$ y NO $g\ (a + b)$

✓ En *Haskell* los argumentos compuestos van entre paréntesis:

$f\ (a\ +\ b)\ c$ aplicar la función f a dos args: $(a + b)$ y c

Ejemplos:

Matemáticas	Haskell
$g\,(x)$	$g\ x$
$f\,(x, y)$	$f\ x\ y$
$g\,(f\,(x, y))$	$g\,(f\ x\ y)$
$f\,(x, g\,(y))$	$f\ x\ (g\ y)$
$g\,(x + y)$	$g\,(x + y)$
$g\,(x) + y$	$g\ x + y$

Definición de funciones

✓ Se usa también la notación *currificada*:

$$\begin{array}{l} \text{-- Un comentario} \\ g \quad :: \; Integer \rightarrow Integer \\ g\ x = x + 1 \end{array}$$

$$\begin{array}{l} f \quad :: \; Integer \rightarrow Integer \rightarrow Integer \\ f\ x\ y = x + y + 2 \end{array}$$

$$\begin{array}{l} doble \quad :: \; Integer \rightarrow Integer \\ doble\ x = x + x \end{array}$$

$$\begin{array}{l} cuadruple \quad :: \; Integer \rightarrow Integer \\ cuadruple\ x = doble\ (doble\ x) \end{array}$$

Significado:

$$\begin{array}{l} \overbrace{f}^{nombre\ fun} \qquad\qquad\qquad \overbrace{::}^{tiene\ tipo} \overbrace{Integer}^{Tipo\ Arg\ 1^0} \rightarrow \overbrace{Integer}^{Tipo\ Arg\ 2^0} \rightarrow \overbrace{Integer}^{Tipo\ Res} \\ \underbrace{f}_{nombre\ fun} \underbrace{x}_{par.\ 1^0} \underbrace{y}_{par.\ 2^0} \underbrace{=}_{se\ define} \underbrace{x + y + 2}_{resultado} \end{array}$$

✓ Nombres de función: comienzan por minúscula

$f \qquad f' \qquad fun3 \qquad fun\_3$

✓ Nombres de parámetros: comienzan por minúscula

$x \qquad y \qquad x' \qquad x1 \qquad xs$

✓ Nombres de tipos: comienzan por mayúscula

$Integer \qquad Bool \qquad Char$

1.4 Sesiones y declaraciones

- ✓ *Programar*: especificar cómo resolver un problema.

- ✓ Un modo natural de describir programas es mediante funciones. Entradas $\longrightarrow$ Salidas

El ordenador funciona como una calculadora o *evaluador*:

?

Valores numéricos enteros:

? $1 + 2$
$3 \;::\; Integer$

Valores reales:

? $cos\ pi$
$-1.0 \;::\; Double$

Solo los argumentos compuestos van entre paréntesis:

? $cos\ (2 * pi)$
$1.0 \;::\; Double$

Ejemplo más elaborado:

? $[1..5]$
$[1, 2, 3, 4, 5] \;::\; [Integer]$

? $sum\ [1..10]$
$55 \;::\; Integer$

Sesiones y declaraciones (2)

Funciones de más de un argumento:

? $mod\ 10\ 3$
$1\ ::\ Integer$

? $mod\ 10\ (3+1)$
$2\ ::\ Integer$

✓ Haskell proporciona un rico conjunto de elementos predefinidos

✓ Este conjunto es extensible: el programador puede definir nuevas funciones, operadores y tipos de datos.

Ejemplo. función que calcula el sucesor de un número entero:

$$
\begin{aligned}
&sucesor && :: && Integer \to Integer \\
&sucesor\ x && = && x + 1
\end{aligned}
$$

Tras proporcionar la declaración de función anterior al evaluador:

? $sucesor\ 3$
$4\ ::\ Integer$

? $10 * sucesor\ 3$
$40\ ::\ Integer$

Una función de dos argumentos:

$$
\begin{aligned}
&sumaCuadrados && :: && Integer \to Integer \to Integer \\
&sumaCuadrados\ x\ y && = && x * x + y * y
\end{aligned}
$$

? $sumaCuadrados\ 2\ (sucesor\ 3)$
$20\ ::\ Integer$

? $sumaCuadrados\ (2+2)\ 3$
$25\ ::\ Integer$

1.5 Reducción de expresiones

El evaluador calcula el resultado de una expresión utilizando las definiciones de las funciones involucradas.

Ejemplo

$$
\begin{aligned}
&cuadrado && :: \ Integer \rightarrow Integer \\
&cuadrado\ x && = \ x * x
\end{aligned}
$$

$$
\begin{aligned}
& 2 + \underline{cuadrado\ 3} \\
\Longrightarrow\ & \{\text{por la definición de } cuadrado\} \\
& 2 + (\underline{3 * 3}) \\
\Longrightarrow\ & \{\text{por el operador } (*)\} \\
& \underline{2 + 9} \\
\Longrightarrow\ & \{\text{por el operador } (+)\} \\
& 11
\end{aligned}
$$

- ✓ Cada uno de los pasos efectuados es una *reducción*.
- ✓ En cada reducción, el evaluador busca una parte de la expresión que sea simplificable (*redex* o *reducto*) y la simplifica.
- ✓ Cuando una expresión no puede ser reducida más se dice que está en *forma normal*.
- ✓ Labor del evaluador: buscar un *redex* en la expresión, *reducirlo* y repetir este proceso hasta que la expresión esté en *forma normal*.

Reducción desde dentro hacia fuera

La definición del comportamiento del evaluador dada es *ambigua*.

¿Qué pasa cuando hay más de un *redex*?

Podemos reducir la expresión desde dentro hacia fuera (reducir primero aquellos reductos más anidados).

$$
\begin{aligned}
cuadrado &:: Integer \to Integer \\
cuadrado\ x &= x * x
\end{aligned}
$$

$$
\begin{aligned}
& cuadrado(\underline{cuadrado\ 3}) \\
\Longrightarrow\ & \{\text{por la definición de } cuadrado\} \\
& cuadrado(\underline{3 * 3}) \\
\Longrightarrow\ & \{\text{por el operador } (*)\} \\
& \underline{cuadrado\ 9} \\
\Longrightarrow\ & \{\text{por la definición de } cuadrado\} \\
& \underline{9 * 9} \\
\Longrightarrow\ & \{\text{por el operador } (*)\} \\
& 81
\end{aligned}
$$

Esta estrategia presenta problemas.

Reducción desde fuera hacia dentro

Reducir la expresión desde fuera hacia dentro (reducir primero los reductos menos anidados).

La definición de la función $cuadrado$

$$\begin{array}{lcl} cuadrado & :: & Integer \rightarrow Integer \\ cuadrado\ x & = & x\ *\ x \end{array}$$

puede ser vista como una *regla de reescritura*:

$$cuadrado\ \boxed{x} \Longrightarrow \boxed{x} * \boxed{x}$$

Se pasan los argumentos a las funciones como expresiones sin reducir, no como valores.

$$\begin{array}{ll}
 & \underline{cuadrado(cuadrado\ 3)} \\
\Longrightarrow & \{\text{por la definición de } cuadrado\} \\
 & \underline{(cuadrado\ 3)} * (cuadrado\ 3) \\
\Longrightarrow & \{\text{por la definición de } cuadrado\} \\
 & (\underline{3 * 3}) * (cuadrado\ 3) \\
\Longrightarrow & \{\text{por la definición de } (*)\} \\
 & 9 * (\underline{cuadrado\ 3}) \\
\Longrightarrow & \{\text{por la definición de } cuadrado\} \\
 & 9 * (\underline{3 * 3}) \\
\Longrightarrow & \{\text{por el operador } (*)\} \\
 & \underline{9 * 9} \\
\Longrightarrow & \{\text{por el operador } (*)\} \\
 & 81
\end{array}$$

Observación: los operadores aritméticos son *estrictos*.

Importancia de la estrategia de reducción

Transparencia referencial: una misma expresión denota siempre el mismo valor.

Consecuencia:

- ✓ Sea cual sea la estrategia seguida en las reducciones, el resultado final (el valor 81) coincide (si se alcanza).
- ✓ La elección de un *redex* equivocado puede hacer que no se obtenga la forma normal de una expresión.

Ejemplo:

$$\begin{array}{lll} \mathit{infinito} & :: & \mathit{Integer} \\ \mathit{infinito} & = & 1 + \mathit{infinito} \end{array}$$

$$\begin{array}{lll} \mathit{cero} & :: & \mathit{Integer} \to \mathit{Integer} \\ \mathit{cero}\ x & = & 0 \end{array}$$

Comportamiento esperado $\forall\ n :: \mathit{Integer}\ .\ \mathit{cero}\ n \Longrightarrow 0$.

Si reducimos siempre el *redex* más interno:

$$\begin{array}{l} \quad \mathit{cero}\ \underline{infinito} \\ \Longrightarrow \{\text{por definición de } \mathit{infinito}\} \\ \quad \mathit{cero}\ (1 + \underline{infinito}) \\ \Longrightarrow \{\text{por definición de } \mathit{infinito}\} \\ \quad \mathit{cero}\ (1 + (1 + \underline{infinito})) \\ \Longrightarrow \{\text{por definición de } \mathit{infinito}\} \\ \quad \ldots \end{array}$$

Si reducimos el *redex* más externo:

$$\begin{array}{l} \quad \underline{cero\ infinito} \\ \Longrightarrow \{\text{por definición de } \mathit{cero}\} \\ \quad 0 \end{array}$$

La estrategia utilizada para seleccionar el *redex* es crucial, ya que puede hacer que se obtenga o no la forma normal de la expresión.

Orden de reducción aplicativo

- ✓ Seleccionar en cada reducción el *redex* más interno (el más anidado).
- ✓ En caso de que existan varios reductos que cumplan la condición anterior, se selecciona el que aparece más a la izquierda en la expresión.

Esto significa que

- ✓ Ante una aplicación de función, se reducen primero los argumentos de la función para obtener sus correspondientes valores (*paso de parámetros por valor*).

A los evaluadores que utilizan este orden se los llama *estrictos* o *impacientes*.

Problemas:

- ✓ A veces, se efectúan reducciones que no son necesarias:

$$
\begin{array}{l}
\quad cero\ (\underline{10 * 4}) \\
\Longrightarrow \{\text{por el operador } (*)\} \\
\quad \underline{cero\ 40} \\
\Longrightarrow \{\text{por definición de } cero\} \\
\quad 0
\end{array}
$$

- ✓ No encuentra la forma normal de ciertas expresiones:

$$
\begin{array}{l}
\quad cero\ \underline{infinito} \\
\Longrightarrow \{\text{por definición de } infinito\} \\
\quad cero\ (1 + \underline{infinito}) \\
\Longrightarrow \{\text{por definición de } infinito\} \\
\quad cero\ (1 + (1 + \underline{infinito})) \\
\Longrightarrow \{\text{por definición de } infinito\} \\
\quad \ldots
\end{array}
$$

Orden de reducción normal

- ✓ Seleccionar el *redex* más externo (menos anidado)

- ✓ En caso de conflicto, de entre los más externos el que aparece más a la izquierda de la expresión.

Esto significa que

- ✓ Se pasan como argumentos expresiones sin evaluar necesariamente (*paso de parámetros por nombre*)

A los evaluadores que utilizan este orden se los llama *no estrictos*.

Ventajas:

- ✓ Es *normalizante*: si la expresión tiene forma normal, una reducción mediante este orden la alcanza. (*Teorema de estandarización*).

- ✓ Un evaluador no estricto solo reducirá aquellos reductos que son necesarios para calcular el resultado final.

Problema:

- ✓ La reducción de los argumentos puede repetirse (menor eficiencia).

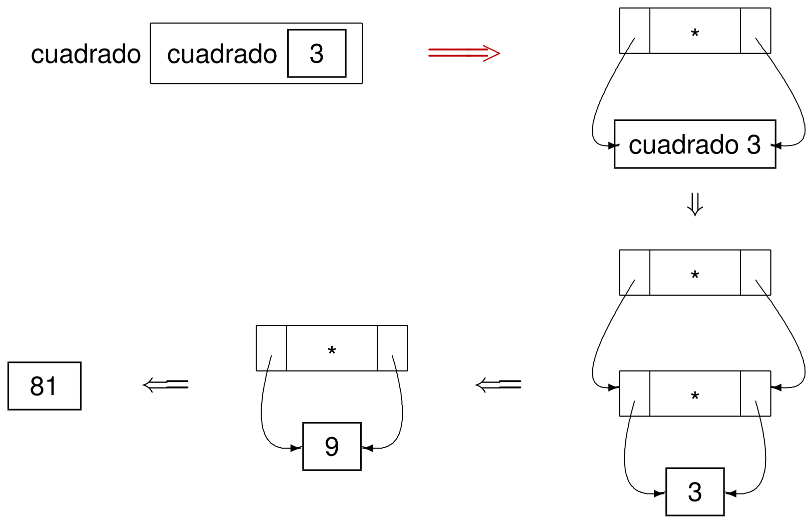
$$
\begin{array}{ll}
 & \underline{cuadrado(cuadrado\ 3)} \\
\Longrightarrow & \{\text{por la definición de } cuadrado\} \\
 & \underline{(cuadrado\ 3)} * (cuadrado\ 3) \\
\Longrightarrow & \{\text{por la definición de } cuadrado\} \\
 & (\underline{3 * 3}) * (cuadrado\ 3) \\
\Longrightarrow & \{\text{por la definición de } (*)\} \\
 & 9 * (\underline{cuadrado\ 3}) \\
\Longrightarrow & \{\text{por la definición de } cuadrado\} \\
 & 9 * (\underline{3 * 3}) \\
\Longrightarrow & \{\text{por el operador } (*)\} \\
 & \underline{9 * 9} \\
\Longrightarrow & \{\text{por el operador } (*)\} \\
 & 81
\end{array}
$$

Evaluación Perezosa

La *Evaluación perezosa* soluciona este problema.

Evaluación perezosa = *paso por nombre* + recordar los valores de los argumentos ya calculados (evita que el cálculo se repita)

Cada expresión se representa mediante un *grafo*.



La reducción de la figura la escribiremos como:

$$\underline{cuadrado\ (cuadrado\ 3)}$$
$\Longrightarrow$ {por la definición de $cuadrado$}
$$a * a \ \textbf{donde}\ a = \underline{cuadrado\ 3}$$
$\Longrightarrow$ {por la definición de $cuadrado$}
$$a * a \ \textbf{donde}\ a = \underline{b * b}\ \textbf{donde}\ b = 3$$
$\Longrightarrow$ {por el operador $(*)$}
$$\underline{a * a}\ \textbf{donde}\ a = 9$$
$\Longrightarrow$ {por el operador $(*)$}
$$81$$

No se realizarán más reducciones que utilizando *paso por valor*.

Posee las ventajas del *paso por nombre* y no es menos eficiente que el *paso por valor*.

Objetivos del tema

El alumno debe:

- ✓ Conocer las bases del estilo de programación funcional
- ✓ Conocer las principales características de Haskell
- ✓ Conocer la notación currificada de Haskell
- ✓ Conocer los principales órdenes de reducción: aplicativo, normal y evaluación perezosa
- ✓ Conocer las principales ventajas e inconvenientes de cada orden de reducción
- ✓ Saber reducir expresiones utilizando los distintos órdenes