EVALUACIÓN DE EXPRESIONES
AJUSTE DE PATRONES
FUNCIONES ESTRICTAS

Nociones básicas

Curso 2019/20

Transparencia referencial

- En los lenguajes funcionales puros como Haskell, la evaluación de una expresión **nunca** produce un **efecto colateral**.
- El resultado de evaluar una expresión e es **independiente del contexto.**
- La evaluación de e produce un valor.

- Valor =
  - Constante (constructora de datos de aridad 0) :

    True, False, -2147483648, . . ., -1, 0, 1, . . ., 2147483647, [ ], ()

  - Aplicación de una constructora de datos de aridad n a n valores :

    (True, [ ]), (0, 5, 7), [-214, 74], [[('a','A'), ('c','8')]]

Evaluación de expresiones (lo básico)

- Todo tipo τ denota un conjunto de valores $\mathcal{T}$.

  Bool = {True,False} ¿Qué denota [Int]?

- Toda expresión e :: τ sintácticamente correcta tiene un valor *v*, dentro del conjunto $\mathcal{T}$, denotado por su tipo. Notación *[[e]] = v*

*[[*3 + 1*]]* = 4 *[[* (2<3,[]) *]]* = (True,[]) *[[*suc 'a'*]]* = 'b'

[[if 2<3 then [3 + 1] else [] *]]* = [4]

*[[*let x = 2 in x * 3*]]* = *[[* (x * 3)[x/2] *]]* = 6

¿Cómo se evalúan las aplicaciones (e1 e2)? Depende de la definición de la función e1

Tipo de las aplicaciones

[[(e1 e2)*]]* = *v* ¿cómo se obtiene *v*?

e1 :: τ -> τ' e2 :: τ (e1 e2) :: τ'

El valor *v* está en $\mathcal{T}$' (conjunto denotado por el tipo τ')

Además la función puede tener varios argumentos

Función *currificada*

f :: τ1 -> τ2 ->… -> τn -> τ = (τ1 -> (τ2 ->… -> (τn -> τ)…))

f e_1 e_2 … e_n = (…((f e_1) e_2)…e_n)

*[[*f e_1*]]* : τ2 ->… -> τn -> τ e_1 :: τ1 , e_2 :: ?

…

*[[*f e_1 e_2 … e_{n-1}*]]* : $\mathcal{T}n$ -> $\mathcal{T}$

*[[*f e_1 … e_{n-1} e_n*]]* es un valor de $\mathcal{T}$.

Este valor depende de la definición de f

Definición ecuacional de una función

- Sucesión de ecuaciones con guardas o no:

f :: $\tau 1$ -> $\tau 2$ ->… -> τn -> τ declaración de tipo (aconsejable)

f p_1 … p_n = e_1 p_1…p_n patrones lineales

… (sin variables en común)

f p'_1 … p'_n

| b_1 = ex_1

| b_2 = ex_2

…

| b_k = ex_k

…

Patrones Haskell

Un patrón *p* puede tener las siguientes formas:

- *x* identificador de variable
- _ variable anónima
- *C* constructora de aridad *0* (constante)
- *p1 … pn* sucesión de patrones
- *C p1 … pn* constructora de aridad *n* aplicada a *n* patrones

(x:xs) ≅ (:) x xs (:) constructora de listas, x, xs variables

Evaluación de una expresión funcional

f :: $\tau 1$ -> $\tau 2$ ->… -> τn -> τ

f e_1 e_2 … e_n = (…((f e_1) e_2)…e_n) (asocia por la izquierda)

[[f e_1 e_2 … e_n *]]* = valor perteneciente a $\mathcal{T}$

1. Se busca la primera ecuación de la def de f cuyo lado izquierdo f p_1…p_n sea tal que los parámetros actuales e_1 … e_n "ajustan con los parámetros formales p_1…p_n" *Ajuste de patrones.* Produce una *sustitución de ajuste.*
2. Se busca la primera guarda (si las hay) para el caso f p_1…p_n anterior que se evalúe a True, tras aplicar la sustitución de ajuste.
3. Se evalúa la expresión de la derecha de la ecuación que cumple las condiciones anteriores, tras aplicar la sustitución de ajuste.

Ajuste de una expresión a un patrón

e se ajusta al patrón *p,* si *tiene la forma* de *p* al sustituir adecuadamente las variables de *p* por otras expresiones :

- *x* cualquier expresión *e* ajusta con x. Sustitución de ajuste [*x*/*e*]
- _ cualquier expresión *e* ajusta con _ No produce sustitución de ajuste
- *C* *e* tiene que ser igual a *C*. No produce sustitución de ajuste
- $p_1 \ldots p_n$ ajustan con él las expresiones de la forma $e_1 \ldots e_n$ si:

  e_i ajusta con p_i $(1 \leq i \leq n)$. Sustitución de ajuste = reunión de las sustituciones de los *n* ajustes
- $C\ p_1 \ldots p_n$ ajustan con él las expresiones de la forma $C\ e_1 \ldots e_n$ si:

  e_i ajusta con p_i $(1 \leq i \leq n)$. Sustitución de ajuste = reunión de las sustituciones de los *n* ajustes

[1,2,3] ajusta con (x:xs) y con (_:x:xs), pero no con (_:[])

Determina la sustitución de ajuste en cada caso

Ejemplo evaluación de (f e1 … en)

```
f :: Int -> Int -> Int
f 0 1 = 2
f x y
  | x > 0 = y
  | otherwise = x
```

- ¿Cuánto vale f (1-1) 1? A partir de la definición de f:
  - La evaluación de (1-1) ajusta con 0 y 1 ajusta con 1 (1ª ecuación)
  - Se evalúa la parte derecha de esta ecuación que da 2
- ¿Cuánto vale f (1-1) 3?
  - La evaluación de (1-1) ajustaría con 0, pero 3 no ajusta con 1
  - Hay ajuste de patrones con la 2ª ecuación
  - La sustitución de ajuste es [x/0, y/3]
  - La primera guarda de la 2ª ecuación que se hace cierta, trás aplicar la sustitución de ajuste es otherwise
  - Su parte derecha vale x = 0 en este caso
- ¿Cuánto vale f (1-2) (3-2)? ¿Se evalúa 3-2? NO

Ejemplo evaluación de (f' e1 … en)

```
f' :: Int -> Int -> Int
f' x y
  | x > 0 = y
  | otherwise = x
f' 0 1 = 2
```

- ¿Cuánto vale f' (1-1) (4-1)?

  (1-1) ajusta con x, (4-1) ajusta con y sin necesidad de evaluar (1ª ecuación). Sustitución de ajuste [x/(1-1), y/(4-1)]

  Se evalúa x para poder evaluar la guarda x >0, *[[*(1-1) >0*]]* = False.

  Se evalúa la siguiente guarda otherwise que es True. El resultado es 0

  No se ha evaluado el segundo argumento
- ¿Cuánto vale f' (1-1) 1?
- ¿Cuánto vale f' (2-1) (4-1)?

El valor indefinido

- Cuando una expresión e no puede evaluarse porque da un error de ejecución o es infinita se dice que está indefinida, *[[e]]* = $\perp$ (bottom)
- A las expresiones e tales que *[[e]]* = $\perp$, las identificamos con bottom, por ejemplo:
  - div 1 0
  - Undefined (símbolo predefinido)
  - head [ ]
  - error “mensaje de error”
  - c = c
- Admitimos bottom como elemento de cualquier tipo

Funciones estrictas

- f :: τ -> τ' es estricta cuando:

$$[[e]] = \bot \Rightarrow [[f\ e\ ]] = \bot$$

f :: τ1 -> τ2 ->… -> τn -> τ es estricta en el i-ésimo argumento cuando:

$$[[e_i]] = \bot \Rightarrow [[\ f\ e_1 \ldots e_{i-1}\ e_i\ e_{i+1} \ldots e_n\ ]] = \bot$$

para cualquier valor de los restantes argumentos

Con las definiciones anteriores de f y f':

1. ¿Cuál es el valor de (f 0 undefined) y de (f' 0 undefined)?
2. ¿Es f estricta en alguno de sus argumentos?
3. ¿Es f' estricta en alguno de sus argumentos?