Programación Funcional

Curso 2019-20

TEORÍA DE TIPOS

Haskell y los tipos

- Lenguaje estáticamente, fuertemente tipado
- Basado en el sistema de Hindley-Milner (creado para ML)
  ☞ Proporciona polimorfismo paramétrico
- Extendido mediante un sistema de clases de tipos
  ☞ Proporciona polimorfismo *ad-hoc* o sobrecarga
- Otras extensiones: laboratorio de ideas muy activo!

⋆ Robin Milner (1934-2010): Premio Turing 1991, realizó contribuciones esenciales en demostradores automáticos (LCF), lenguajes de programación (ML) y sistemas concurrentes (π-cálculo)
⋆ Roger Hindley: contribuciones a la lógica y el λ-cálculo

Sistema de tipos Hindley-Milner

- Tipo $\equiv$ colección de valores
- Una expresión tiene el tipo de su valor

  `e::T` expresa que `e` tiene o admite el tipo `T`

  En Haskell: `> :t` *e* muestra el tipo de *e*

En Hindley-Milner

- Cada expresión bien tipada puede admitir varios tipos, pero un solo *tipo principal*, que es el más general de todos ellos.
- Los tipos principales pueden ser inferidos, sin necesidad de que el programador declare los tipos.
- Una propiedad esencial (*preservación de tipos*): el tipo de una expresión no cambia durante el proceso de su evaluación
  - O sea: $e :: T \wedge e \rightarrow e' \;\Rightarrow\; e' :: T$
    donde $e \rightarrow e'$ indica un paso de evaluación
  - $e :: T \;\Rightarrow\; [\![\mathrm{e}]\!] \in \mathcal{T}$ aceptando que $\perp :\in \mathcal{T}$

Anatomía de los tipos Hindley-Milner
Tipos monomórficos

`T ::=`		`TP`	Tipo primitivo: Char, Bool, Int,…
	`|`	`(T1,...,Tn)`	Producto de tipos: T1$\times \ldots \times$Tn
	`|`	`[T]`	Listas de elementos de tipo T
	`|`	`T -> T'`	Tipo funcional

`Añadiremos tipos definidos por el usuario`

Algunos tipos monomórficos

`Char` `(Char,Int)` `[Int]` `[[Int]]` `Int -> Int`

`(Char->Int , Int , Bool)` `(Char->Int)->([Char]->[Int])`

Anatomía de los tipos Hindley-Milner
Tipos polimórficos

Tipos simples

`TS ::=`		`TP`	Tipo primitivo: Char, Bool, Int,...
	`|`	`(TS1,...,TSn)`	Producto de tipos: TS1$\times \ldots \times$TSn
	`|`	`[TS]`	Listas de elementos de tipo TS
	`|`	`TS -> TS'`	Tipo funcional
	`|`	`a`	Variable de tipo: `a,a',b,...`

Tipos genéricos o esquemas de tipo

`T ::=`		`TS`	Tipo simple
	`|`	$\forall$ `a.T`	Tipo cuantificado o polimórfico

O sea: `T` $\equiv \forall a_1 \ldots \forall a_n$.`TS` $(n \geq 0)$

`T` es *cerrado* si todas las variables de `TS` están cuantificadas

Algunos tipos simples

`Char` `(a,b)` `[b]` `Int -> Int` `Int -> a`
`(a , Bool , a')` `(a->b)->([a]->[b])`

Algunos esquemas de tipo

`Char` `∀ b.(a,b)` `[Int->Int]` `∀ a.Int -> a`
`(a , Bool , a')` `∀ a.∀ b.(a->b)->([a]->[b])`

Estos son cerrados

Tipos de algunas expresiones (algunas son funciones)

`'a'::Char` `(True,'z')::(Bool,Char)`

`[['b','\n'],[]]::[[Char]]` `not::Bool -> Bool`

`length::∀a.[a]->Int` `fst::∀a.∀b.(a,b)->a`

Polimorfismo paramétrico

`fst::∀a.∀b.(a,b)->a` se infiere de `fst (x,y) = x`

- `x,y` valores cualesquiera
- `a,b` tipos cualesquiera
- `fst` definida de modo uniforme para todos los tipos

Con las opciones por defecto, Haskell no muestra los ∀

Funciones *currificadas*

¿Qué pasa con las funciones de más de un argumento?

¿Cuál es el tipo de `(&&)` , `(++)` , `take` , `elem` , ...?

`(&&)::`	`Bool->Bool->Bool`	$\equiv$ `Bool->(Bool->Bool)`
`(++)::`	$\forall a.$`[a]->[a]->[a]`	$\equiv$ $\forall a.$`[a]->([a]->[a])`
`take::`	$\forall a.$`Int->[a]->[a]`	$\equiv$ $\forall a.$`Int->([a]->[a])`
`elem::`	$\forall a.$`a->[a]->Bool`	$\equiv$ $\forall a.$`a->([a]->Bool)`

☞ Los argumentos vienen de uno en uno
Visión currificada de las funciones

Visión currificada de las funciones

Sea f una función de tres argumentos x, y, z que se toman de tres conjuntos (tipos) A, B, C, y que devuelve un resultado r de un conjunto D.

- Visión matemática usual: tres argumentos en una terna

  $f : A \times B \times C \;\rightarrow\; D$

  $f(x, y, z) = r$

  f, aplicada a la terna (x, y, z), devuelve r

- Visión currificada: los argumentos de uno en uno

  $f : A \rightarrow (B \rightarrow (C \rightarrow D))$

  $((f\ x)\ y)\ z = r$

  f, aplicada a x, devuelve la función que, aplicada a y, devuelve la función que, aplicada a z, devuelve r

Dos azúcares sintácticos de uso **constante**

La flecha de los tipos (`->`) asocia por la derecha

```
A -> B -> C -> D  ≡  A -> (B -> (C -> D))
                  ≡  A -> B -> (C -> D)
                  ≡  A -> (B -> C -> D)
                  ≢  (A -> B) -> C -> D
                  ≢  (A -> B -> C) -> D
                  ≢  A -> (B -> C) -> D
```

La aplicación en expresiones asocia por la izquierda

```
f x y z  ≡  ((f x) y) z
         ≡  (f x y) z
         ≡  (f x) y z
         ≢  f (x y z)
         ≢  f (x y) z
         ≢  f x (y z)
```

Efectos de la currificación

Un lema famoso de la programación funcional

Las funciones son *ciudadanos de primera clase*

- Una función puede ser argumento o resultado de otra
- Las expresiones de tipo funcional son evaluables
  Si $e :: T \to T'$, entonces $[\![\mathrm{e}]\!]$ es una función de $\mathcal{T}$ en $\mathcal{T}'$

Efectos de la currificación (II)

Aplicaciones parciales

- Si $f :: T_1 \to T_2 \to \ldots \to T_n \to T$ y $m < n$ entonces $f\ e_1\ \ldots\ e_m$ es una aplicación parcial de f.
  Debe tenerse: $e_1 :: T_1, \ldots, e_m :: T_m$
  Se tendrá: $f\ e_1\ \ldots\ e_m :: T_{m+1} \to \ldots \to T_n \to T$
- `take 2::[a] -> [a]` `take:: Int -> [a] -> [a]`
  `elem::a -> [a] -> Bool`
  `elem (True || False)::[Bool] -> Bool`
  `(&&) False::Bool -> Bool`

  Veremos un azúcar para aplicaciones parciales de operadores

Secciones de operadores infijos

Dado un operador infijo $\bigoplus$, hay sendas notaciones especiales para escribir su aplicación parcial a sus argumentos izquierdo o derecho.

(x $\bigoplus$)

Denota la función definida como `(x ⨁) y = x ⨁ y`

```
> (2 ^) 3
8
```

```
> map (2 ^) [1,2,3]
[2,4,8]
```

Es un azúcar para la aplicación parcial `((⨁) x)`

`map f xs` es el resultado de aplicar f a todos los elementos de xs

`map f [x1,...,xn] = [f x1,...,f xn]`

($\bigoplus$ y)

Denota la función definida como `(⨁ y) x = x ⨁ y`

```
> (^ 2) 3
```

```
> map (:[]) [1,2,3]
```