Paradigmas de Programación (EIF-400) Nociones cálculo λ

CARLOS LORÍA-SÁENZ LORIACARLOS@GMAIL.COM

AGOSTO-SETIEMBRE 2017

EIF/UNA

Objetivos

- Presentar las nociones principales del cálculo λ
- Estudiarlo como modelo teórico de la FP
- Relacionarlo con la herramienta de trabajo (JS)

Objetivos Específicos

- Nociones y sintaxis
- Reducciones y propiedades
- Representación de aritmética y lógica booleana
- Recursión
- Computación por manipulación simbólica

Tema del programa

10. Introducción al cálculo lambda (λ-calculus) (Objetivos 2, 5, 7)

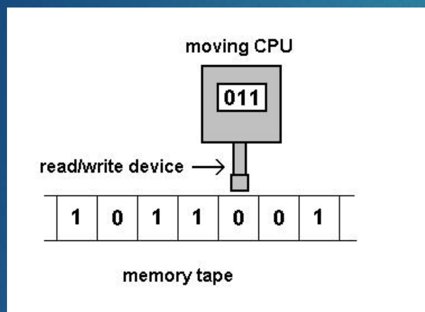
a. Historia y descripción.
b. Evaluación por conversión y reducción.
c. Aritmética y Lógica.
d. Recursividad.
e. Funciones computables.
f. Evaluación por sustitución.

Nociones

- Teoría de funciones matemáticas
- Desarrollado alrededor de 1930 principalmente por Alonso Church
- Estudiar qué significa *computabilidad* (análogo al estudio de Allan Turing pero con otro enfoque)
- Abstrae el concepto de función a objeto matématico y sus propiedades de "computar"
- Usos en semántica de lenguajes (naturales y artificiales)
- Dio origen a LISP (alrededor de 1960) el primer lenguaje de FP

Modelos de computación

Máquina de Turing



Cálculo Lambda

$$\lambda x. f x = f$$

$$\begin{aligned} f(0, \boldsymbol{y}) &= g(\boldsymbol{y}) \\ f(x+1, \boldsymbol{y}) &= h(x, f(x, \boldsymbol{y}), \boldsymbol{y}) \end{aligned}$$

Funciones
recursivas

¿Qué es “*computar*”?

- En este enfoque: manipular sistemáticamente símbolos
- **Hipótesis (Turing-Church)**: algo es computable sii lo puedo computar con el cálculo lambda (máquina de Turing ó Función recursiva)
- “Corolario”: La computación es alcanzable únicamente con la manipulación de símbolos
- Leer acá http://web.archive.org/web/20090925082414/http://www.claudiogutierrez.com/bid-fod-uned/Newell.html

Evaluación simbólica == computar

- Definir símbolos permitidos
- Reglas de combinación: reemplazo (sustitución) igual por igual
- Estrategias de "simplificación" ("normalización")
- Se pueden "probar" propiedades matemáticas de funciones ("formal software verification")

Ejercicio

- Usando `simplificar.js`
- Pruebe formalmente:
  - $\forall f: id(f) = f$
  - $\forall f: id(f) = id1(f)$
  - $\forall a: Array\ \forall i 0 \leq i < a.length$:
    - $addOneToList(a)[i] = a[i] + 1$

```
PP:node
> let {id, id1, add, addOneToList} = require('./simplificar')
undefined
> var a = [0, 1, 2, 3, 4]
undefined
> let aOne = addOneToList( a )
undefined
> a.every( (_, i) => a[i] + 1 == aOne[i])
true
>
```

Ideas y sintaxis

- Analogía con JS
- Función es un objeto
- En el cálculo $\lambda x.\lambda y.((add\ x)y)$
- Se llama abstracción
- Hay expresiones : variables, abstracciones y aplicaciones

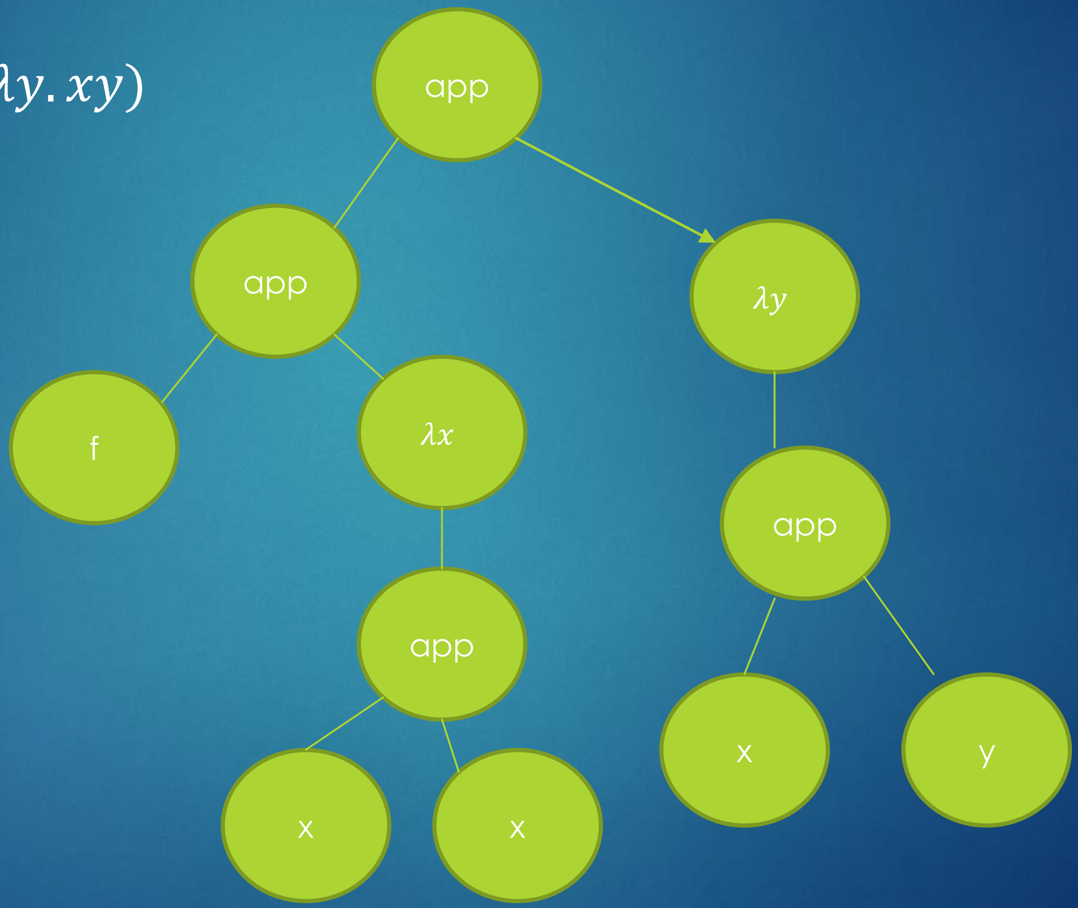
```
let add = x => y => x+y

let f = x => y => add(x)(y)
```

Expresiones: sintaxis

- Variables $x, y, z, \ldots$
- Si M y N son expresiones entonces (MN) es una aplicación
- Si M es una expresión $\lambda x.(M)$ es una abstracción o expresión lambda. M se llama el cuerpo. Es el alcance (scope) de x.
- Omitimos los paréntesis si no hay ambigüedad: el cuerpo tiene más precedencia. Ahorramos lambdas apiladas en una sóla. Se asume asociación a la izquierda:
- $((MN)P) = MNP$
- $\lambda x.\left(\lambda y.(MN)\right) = \lambda x.\lambda y.MN = \lambda xy.MN$

AST de una expresión λ

- $f(\lambda x.xx)(\lambda y.xy)$



Propiedad: Forma de Curry

- Basta trabajar con lambdas unarias (de un solo argumento)
- Curry de una función ("*curied*" form)

```
let noCurried = (x, y) => x + y

let curried = x => y => x + y
```

Ejercicio

- Escriba una versión de Curry de Math.pow

```
let powCurried = ... => Math.pow(x, n)
```

Variable libre

- Se dice que x es libre en M si en el camino desde la raíz hasta M x no es abstraída por una λx
- De otra forma se dice atada en M.
- Note que una variable puede ocurrir libre y atada en una expresión: Ejemplo $\lambda x.[x,\ \lambda x.x]$

```
x => [x, x => x]
```

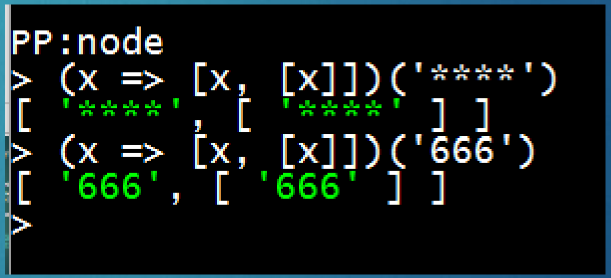
x ocurre libre y atada

Sustitución

- Informalmente: Si se tiene que $x = M$ en un y x es libre ahí se puede reemplazar x por M
- A veces escribimos $[M/x]N$ la sustitución en todo N de cada ocurrencia libre de x por M

Primera regla

- Si se pone una abstracción $\lambda x. M$ a la izquierda de una expresión N la lambda se "come" a N : la desaparece y se sustituye x por N en todo M.
- Se llama reducción β



Ejercicio

- Usando estas definiones calcule simbólicamente $f(10)(20)$

```
let add = x => y => x+y

let f = x => y => add(x)(y)

f(10)(20) = ...
```

Segunda Regla

- Se vale renombrar una variable x atada en $\lambda x.M$ por otra variable z (que no ocurra libre en M) y se cambian las ocurrencias libres de x por z en todo M.
- $\lambda x.\lambda y.xyx\lambda x.yx = \lambda z.\lambda y.zyz\lambda x.yx$
- Se llama reducción α

```
x => [x, x => x] es lo mismo que z => [z, x => x]
```

Tercera regla

- Se puede eliminar las abstracción de $\lambda x.M$ si x no ocurre libre en M
- Se llama reducción η (eta)
- Ejemplo

$$\lambda x.(\lambda y.yy)x = (\lambda y.yy)$$

```
let f = x => g(x)
let g = x => x + 1

f y g son eta-equivalentes
```

Forma normal

- M está en beta forma normal sino se le puede aplicar ninguna reducción beta
- Similar para eta.
- Nos concentraremos a reducciones beta para calcular formas normales

Estrategias de reducción

- *Left-most-inner-most-first*: bajamos por el árbol prefiriendo bajar por la izquierda y primero en profundidad (aplicativa, eager, by-value).
- La misma que usan mayoría de lenguajes de programación
- *Left-most-outer-most-first*: damos prioridad a reducciones izquierdas lo más arriba en el árbol (normal-order, lazy, by-need)
- $(\lambda a.(\lambda x.xa)\lambda y.y)((\lambda z.z)c)$
- ¿Qué pasa con $\lambda y.((\lambda x.x)\lambda x.x)$?
- Aplicativa no es siempre normalizante, normal-order sí

Forma normal débil (wnf)

- Se le dice a $(\lambda x. M)N$ un beta redex
- En la raíz del árbol no hay un beta-redex, a lo más en los nodos interiores
- Ejemplo
- $f((\lambda x. x)y)$ está en WNF pero no en forma normal

Teorema

- SI M tiene dos formas normales M_1 y M_2 estas son α-equivalentes (sólo difieren en las variables atadas, a lo más).
- Es decir las formas normales son únicas ni importa por cuál camino (estrategia) se busquen

Representación de teorías

- El cálculo permite codificar (representar) naturales y su aritmética
- Igualmente la lógica booleana
- Y veremos que acepta recursión
- Es un resultado muy interesante que a partir de reglas tan simples se pueda construir tales teorías

Representación de naturales

- Codificamos los números naturales con lambdas
- Sea s^n la composición de s consigo misma n veces
- Ver `selfCompose` en `lambda.js`
- $\hat{n} = \lambda sz.s^n(z)$
- $SUCC = \lambda nsz.s(nsz)$
- Propiedad: $\hat{n}yx = y^n x$
- Probar que $SUCC(\hat{n}) = \widehat{n+1}$

Ejercicio: numerales de Church

- Estudie la implementación de en `naturals.js`

```
PP:node
> let {test1} = require('./naturales')
undefined
> test1()
0 = 0
s(0) = 1
s(s(0)) = 2
s(s(s(0))) = 3
undefined
>
```

Suma, multiplicación

- $PLUS = \lambda mn.m\ SUCC\ n$ (ya está en `naturales.js`)
- $MULT = \lambda mny.m(ny)$
- Pruebe <u>formalmente</u> que funcionan bien
- Implemente $MULT$ en naturales.js.

Ejercicio avanzado

- Implemente PRED el predecesor de un natural en `naturals.js`.
- Wikipedia da esta:
- $PRED = \lambda nfx.n(Tf)(Kx)$
- Donde $T = \lambda fgh.fh(gf), K = \lambda xu.x, I = \lambda u.u$
- Probar por inducción que $T^{n+1}f(Kx) = \lambda h.h(f^n x)$
- Pruebe formalmente que PRED está correcta

Recursión

- Veremos el combinador Y que permite hacer recursión con lo cuál es posible escribir algoritmos como el factorial.

Combinador `Y`

- 'Y' encuentra puntos fijos de funciones
- Un punto fijo de una función f es un a tal que $f(a) = a$. Por ejemplo 0 y 1 son puntos fijos de $f(x) = x^2$
- Existe una expresión Y que dada una función f encuentra un punto fijo. Es decir $f(Y(f)) = Y(f)$
- Se cumple:$f(f(Y(f))) = f(Y(f)) = Y(f)$ Es decir permite llamar a f recursivamente.
- $Y = \lambda f.(Uf)(Uf)$ donde $U = \lambda fx.f(x\ x)$
- Pruebe que $Yf\ =\ f(Yf)$

Ejercicio

- Estudie `Y.js`
- Calcule usando `Y` en `Y.js` el punto fijo de la siguiente función `gmisterio`
- Trate de deducir qué función es

```
let gmisterio = f => n => (n == 0) ? 0 : 2 + f(n - 1);
```

```
PP:node
> let {Y} = require('./Y')
undefined
> let gmisterio = f => n =>  (n == 0) ? 0 : 2 + f( n - 1 );
undefined
> let h = Y(gmisterio) // Y-ificamos gmisterio
undefined
> [0, 1, 2, 3, 4].forEach( i => console.log(`h(${i}) = ${h(i)}`) )
h(0) = 0
h(1) = 2
h(2) = 4
h(3) = 6
h(4) = 8
undefined
>
```

Representación de Lógica

- $TRUE = \lambda xy.x$
- $FALSE = ZERO = \lambda xy.y$
- $NOT = \lambda pxy.pyx$
- Probar formalmente que sirven
- $ITE\ p\ t\ e = p\ t\ e$

```
PP:node
> let {ITE, TRUE, FALSE} = require('./logica')
undefined
> ITE(TRUE)('YES')('NO')
'YES'
> ITE(FALSE)('YES')('NO')
'NO'
>
```

Ejercicios

- Implemente `AND` y `OR` usando `ITE` en `lógica.js`
- Usando `Y` de `Y.js` defina una función g tal que:
  - Si definimos `h = Y(g)`
  - Entonces `h(a) = máximo(a)` si `a` es un arreglo no vacio de números

```
PP:node
> let {Y} = require('./Y')
undefined
> let g = f => a =>                ?
undefined
> let h = Y(g)
undefined
> h([2, 4, 1, 10, 3])
10
>
```