Lenguajes de Programación
Nota de clase 3: Cálculo λ

Karla Ramírez Pulido Manuel Soto Romero Javier Enríquez Mendoza

14 de octubre de 2021
Facultad de Ciencias UNAM

3.1. Introducción

El Cálculo λ, es una de las teorías más importantes del área de Lenguajes de Programación. Fue introducida por Alonzo Church en la década de 1930, con el fin de formalizar el concepto de computabilidad[1]. Es considerado por algunos, el lenguaje de programación universal más pequeño del mundo y consiste de un lenguaje formal para describir términos y definir funciones, usando una única regla de transformación llamada β-reducción que permite sustituir variables.

Este sistema es el fundamento de los lenguajes de programación funcionales y engloba la noción atómica tanto de composición como de abstracción de funciones de un argumento. Hoy en día existen muchas variantes de éste, sin embargo en esta nota estudiaremos su variante más simple.

3.2. Sintaxis

El Cálculo λ consiste de una notación para representar expresiones, llamados *términos* λ. Existen tres posibles términos: *variables*, *abstracciones lambda* y *aplicaciones de función*.

Para construir dichos términos se tiene la siguiente gramática:

```
<expr> ::= <var>
         | (λ<var>.<expr>)
         | (<expr> <expr>)

<var> ::= x | y | z | ...
```

3.2.1. Variables

Las variables o identificadores son representadas mediante letras minúsculas. Algunos ejemplos de variables:

$$x \quad y \quad z \quad w$$

[1] Estudia los problemas de decisión que se pueden resolver mediante un algoritmo.

3.2.2. Abstracciones λ

Las funciones, abstracciones λ o simplemente abstracciones, representan la definición de funciones *anónimas,* esto es, no requieren de un nombre explícitamente para ser definidas o usadas. Tales abstracciones se componen de un encabezado y un cuerpo. El encabezado contiene el nombre del parámetro formal de la función y el cuerpo de ésta se compone otras subexpresiones que son por supuesto términos λ. Todo lo que se encuentra antes del punto es el encabezado y lo que se encuentra después es el cuerpo.

$$\underbrace{\lambda x}_{\text{Encabezado}}.\ \underbrace{x}_{\text{Cuerpo}}$$

Otros ejemplos de abstracciones son:

1. $\lambda x.\lambda y.xy$
2. $\lambda z.\lambda w.u$
3. $\lambda u.\lambda v.vvv$

Las expresiones anteriores, tienen como encabezado λx, λz y λu respectivamente, mientras que $\lambda y.xy$, $\lambda w.u$ y $\lambda v.vvv$ conforman el cuerpo de las mismas.

La primera abstracción de los ejemplos anteriores, es equivalente a la función de dos parámetros $\lambda xy.xy$ sin embargo, la especificación de términos λ del Cálculo λ no permite definir funciones con más de un parámetro. Una solución que se da a esta situación es modificar las definiciones de función de manera reciban un único parámetro (en este ejemplo x) que tendrán como cuerpo otra función de igualmente un parámetro (y), a esta técnica se le conoce como *currificación* . A continuación se exponen algunos ejemplos de abstracciones con su currificación:

1. $\lambda xy.xy \equiv_{curry} \lambda x.\lambda y.xy$
2. $\lambda zw.u \equiv_{curry} \lambda z.\lambda w.u$
3. $\lambda abc.cba \equiv_{curry} \lambda a.\lambda b.\lambda c.cba$

3.2.3. Aplicaciones de función

La regla de producción **<expr> <expr>** de la gramática para términos λ representa la aplicación de funciones. Dada la expresión de una aplicación de función $e_1\ e_2$, se dice que e_1 está en posición de función y que e_2 está en la posición de su argumento. De esta forma, una aplicación representa el proceso de evaluar una función con su respectivo argumento (también llamado parámetro real). Por ejemplo la siguiente función:

$$\lambda x.x\ \ \lambda y.y$$

representa la aplicación de la función $\lambda x.x$ con la función $\lambda y.y$, sin embargo, es posible incluir paréntesis que faciliten la lectura o ayuden a entender el orden de evaluación. Algunos ejemplos:

1. $(\lambda x.x)\,(\lambda y.y)$
2. xy
3. $(\lambda w.ws)\,(\lambda y.yy)\,u$

En caso de que no haya paréntesis explícitos, la aplicación de funciones asocia a la izquierda.

3.2.4. Alcance

El alcance que toma un identificador es la región de un programa en la cual éstos alcanzan su valor. En el Cálculo λ el alcance de todo identificador se da en el cuerpo de las abstracciones.

En la expresión $\lambda x.x$ se dice que el identificador x está *ligado* o *acotado*, pues se encuentra en el cuerpo de la función que toma como parámetro el mismo nombre del identificador x (*de ligado*). Si un identificador no es precedido por una λ con el mismo parámetro, se dice que está *libre* en la expresión. Por ejemplo, en la siguiente expresión:

$$\lambda x.xy$$

El identificador x está ligado, mientras que el identificador y se encuentra libre. Otro ejemplo es:

$$(\lambda x.x)\,(\lambda y.x)$$

Se observa que en la subexpresión $(\lambda x.x)$ el identificador o la variable x se encuentra ligada, mientras que la subexpresión $(\lambda y.x)$ la variable x está libre. Es importante observar que la x de la segunda subexpresión es independiente a la x de la primera.

Se define formalmente a los identificadores libres como sigue:

- x está libre en y, con y cualquier variable (incluyendo a x).
- x está libre en $(\lambda y.e)$ si y sólo si $x \neq y$ y x se encuentra libre en e, siendo e una expresión cualquiera.
- x está libre en e_1e_2 si y sólo si x está libre en e_1 o en e_2, con e_1y e_2 expresiones cualquiera.

Los identificadores ligados se definen formalmente como sigue:

- x se encuentra ligado en $(\lambda x.e)$
- x se encuentra ligado en $(\lambda y.e)$ con $x \neq y$ si y sólo si x está ligado en e.
- x está ligado en e_1e_2 si y sólo sí, x está ligado en e_1 o e_2.

Se observa que gracias a la tercera regla de ambas definiciones, estas no son excluyentes entre sí, esto es, un identificador puede estar libre y ligado al mismo tiempo. Una expresión e tal que no tiene variables libres, se dice que es una expresión *cerrada* también conocida como *combinador*.

3.2.5. α-equivalencias

Los nombres de los identificadores ligados no son importantes en el contexto del Cálculo λ. Por ejemplo $\lambda x.x$ y $\lambda y.y$ representan a la misma función y lo único que cambia es el nombre del identificador. A esto se le conoce como relación de α-*equivalencia.* Sin embargo, la regla tiene ciertas restricciones:

$$\lambda V.E \equiv_\alpha \lambda W.E\,[V := W] \text{ si } W \text{ no está en las variables libres de } E$$

Algunos ejemplos:

- $\lambda x.x \equiv_\alpha \lambda x.x$
- $\lambda x.x \equiv_\alpha \lambda y.y$
- $\lambda x.\,(\lambda x.x)\,x \equiv_\alpha \lambda y.\,(\lambda x.x)\,y$

3.3. Semántica

3.3.1. β-reducciones

La regla de β-*reducción* expresa la idea de aplicación de función y ésta es la regla semántica del Cálculo λ, es decir, la forma de evaluar una expresión. Esta regla se define como sigue:

$$(\lambda x.e)\,s \to_\beta e\,[x := s]$$

Es decir, se deben sustituir todas las apariciones de x por s en e. Lo anterior expresa la idea de una función a la cual se le asigna el parámetro real a su parámetro formal. A la expresión $(\lambda x.e)$ se le llama *redex*[2], mientras que a la expresión $e\,[x := s]$ se le llama *reducto*.

Algunos ejemplos:

- $(\lambda x.x)\,x \to_\beta x$
- $(\lambda x.x)\,(\lambda x.x) \to_\beta \lambda x.x$
- $(\lambda x.x)\,(\lambda y.y) \to_\beta \lambda y.y$
- $(\lambda x.\lambda y.\,(xy))\,(\lambda x.x) \to_\beta \lambda y.\,(\lambda x.x)\,y \to_\beta \lambda y.y$

Formas Normales

Dada una expresión e, si no existe e' tal que $e \to_\beta e'$, se dice que e se encuentra en Forma Normal. Es decir, dada una expresión, si ésta no puede reducirse más mediante β-reducciones, se dice entonces que esta expresión se encuentra en Forma Normal. Por ejemplo, todas las reducciones anteriores, están en Forma Normal.

[2]Del inglés *reducible expression.*

Divergencia

También existen expresiones, que no tienen una Forma Normal pues sin importar cuantas reducciones se apliquen, siempre se obtiene una nueva expresión que puede ser reducida, en este caso se dice que la expresión *diverge* y por lo tanto no tiene Formal Normal. Por ejemplo, sean $\omega =_{def} \lambda x.xx$ y $\Omega =_{def} \omega\omega$, entonces:

$$\Omega =_{def} \omega\omega =_{def} (\lambda x.xx)\,\omega \to_\beta \omega\omega =_{def} \Omega$$

De manera que Ω se reduce a sí mismo, por lo tanto diverge:

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \Omega \to_\beta ...$$

Confluencia

El Cálculo λ tiene una natural no determinista, por ejemplo la expresión $(\lambda x.\,(\lambda y.yx)\,z)\,v$ tiene dos reductos distintos:

- $(\lambda x.\,(\lambda y.yx)\,z)\,v \to_\beta (\lambda y.yv)$
- $(\lambda x.\,(\lambda y.yx)\,z)\,v \to_\beta (\lambda x.zx)\,v$

En este caso ambos caminos llevan a la misma Forma Normal (zv) sin embargo, debemos preguntarnos si esto siempre sucede. Si queremos usar el Cálculo λ como un sistema de cómputo, el resultado final debe ser el mismo independientemente del camino utilizado. Si esta propiedad llamada *confluencia* fuera falsa para nuestra regla de β-reducción, cualquier intento de usar el Cálculo λ como lenguaje de programación sería fallido.

Teorema 1. *(Confluencia o propiedad de Church-Rosser) Si $e \to^*_\beta r$ y $e \to^*_\beta s$ entonces existe un término t tal que $r \to^*_\beta t$ y $s \to^*_\beta t$.*

Corolario 1. *(Unicidad de las Formas Normales) Si e tiene una Forma Normal entonces es única. Es decir si $e \to^*_\beta e_f$ y $e \to^*_\beta e'_f$ entonces $e_f = e'_f$ (salvo α-equivalencia).*

La demostración de estos resultados queda fuera de los alcances de este curso. El corolario garantiza que el Cálculo λ puede usarse como un sistema de cómputo. Sin embargo el uso de β-reducciones sin restricciones o estrategias adicionales podría prohibir la obtención de una Forma Normal. Por lo anterior, si se usara directamente como lenguaje de programación habría algunos programas que, teniendo un valor bien definido, podrían ciclarse infinitamente.

3.4. Aritmética

El Cálculo λ permite modelar operaciones aritméticas. Para poder usar estas operaciones, añadiremos la representación de números (Numerales de Church).

3.4.1. Numerales de Church

Para representar números naturales, se tiene una definición para el cero y el resto de naturales, se construyen a partir de la función sucesor. De esta forma, podemos representar números con funciones que reciben una función sucesor (s) y una función cero (z), se definen cada uno de estos números aplicando s tantas veces como sea necesario a z. Se conoce a esta representación como *Numerales de Church*.

- $0 =_{def} \lambda s.\lambda z.z$
- $1 =_{def} \lambda s.\lambda z.sz$
- $2 =_{def} \lambda s.\lambda z.s\,(sz)$

  ...
- $n =_{def} \lambda s.\lambda z.\underbrace{s(...(s}_{n\ veces}\ z)...)$

Por supuesto, todo depende de la definición de sucesor que demos. Una definición de esta función se muestra a continuación:

$$S =_{def} \lambda n.\lambda a.\lambda b.a\,(nab)$$

Por ejemplo, para calcular el sucesor de cero, tenemos:

$S\ 0$
$=_{def} (\lambda n.\lambda a.\lambda b.a\,(nab))\,0$
$\rightarrow_\beta \lambda a.\lambda b.a\,(0ab)$
$=_{def} \lambda a.\lambda b.a\,((\lambda s.\lambda z.z)\,ab)$
$\rightarrow_\beta \lambda a.\lambda b.a\,((\lambda z.z)\,b)$
$\rightarrow_\beta \lambda a.\lambda b.ab \equiv_\alpha \lambda s.\lambda z.sz =_{def} 1$

3.4.2. Funciones aritméticas

Una vez definidas estas funciones, podemos definir operaciones entre números naturales.

Sumas

Una suma en realidad es una aplicación de la función sucesor, por ejemplo, si se desea sumar los números $n + b$, se debe aplicar la función sucesor n veces a m. Por ejemplo, para sumar $2 + 3$ tenemos:

$2S3$
$=_{def} (\lambda s.\lambda z.s\,(sz))\,S3$
$\rightarrow_\beta (\lambda z.S\,(Sz))\,3$
$\rightarrow_\beta S\,(S3) \rightarrow_\beta ... \rightarrow_\beta 5$

De esta forma, podemos definir la función suma que recibe dos números n y m como sigue:

$$A =_{def} \lambda n.\lambda m.nSm$$

Debido a la conmutatividad de la suma, tenemos también la definición:

$$A' =_{def} \lambda n.\lambda m.mSn$$

Se deja como ejercicio al lector, demostrar la equivalencia entre estas dos funciones.

Multiplicación

Similarmente se define la operación multiplicación como sigue:

$$\lambda x.\lambda y.\lambda a.x\,(ya)$$

Se deja como ejercicio al lector el cálculo del producto 3×3, es decir:

$$(\lambda x.\lambda y.\lambda a.x\,(ya))\,33$$

3.5. Álgebra Booleana

Al igual que con los números, es posible representar expresiones booleanas y operaciones entre éstas. Las constantes lógicas *verdadero* y *falso* se representan mediante:

$$\begin{array}{lll} T & =_{def} & \lambda x.\lambda y.x \\ F & =_{def} & \lambda x.\lambda y.y \end{array}$$

La función T, dados dos valores para verdadero y falso (x y y respectivamente), regresa x, mientras que la función F regresa y.

Operaciones con booleanos

Con estas definiciones es posible definir algunas operaciones del Álgebra Booleano, a continuación se muestran las definiciones de las funciones not, or y and junto con su tabla de verdad.

Negación

$$\begin{array}{lll} \neg & =_{def} & \lambda x.xFT \end{array}$$

$\neg F$
$=_{def}\ (\lambda x.xFT)\,F$
$\rightarrow_\beta\ FFT$
$=_{def}\ (\lambda x.\lambda y.y)\,FT$
$\rightarrow_\beta\ (\lambda y.y)\,T$
$\rightarrow_\beta\ T$

$\neg T$
$=_{def}\ (\lambda x.xFT)\,T$
$\to_\beta\ TFT$

$=_{def}\ (\lambda x.\lambda y.x)\,FT$
$\to_\beta\ (\lambda y.F)\,T$
$\to_\beta\ F$

Disyunción

$$\vee \quad =_{def} \quad \lambda x.\lambda y.xTy$$

$\vee FF$
$=_{def}\ (\lambda x.\lambda y.xTy)\,FF$
$\to_\beta\ (\lambda y.FTy)\,F$
$\to_\beta\ FTF$
$=_{def}\ (\lambda x.\lambda y.y)\,TF$
$\to_\beta\ (\lambda y.y)\,F$
$\to_\beta\ F$

$\vee FT$
$=_{def}\ (\lambda x.\lambda y.xTy)\,FT$
$\to_\beta\ (\lambda y.FTy)\,T$
$\to_\beta\ FTT$
$=_{def}\ (\lambda x.\lambda y.y)\,TT$
$\to_\beta\ (\lambda y.y)\,T$
$\to_\beta\ T$

$\vee TF$
$=_{def}\ (\lambda x.\lambda y.xTy)\,TF$
$\to_\beta\ (\lambda y.TTy)\,F$
$\to_\beta\ TTF$
$=_{def}\ (\lambda x.\lambda y.x)\,TF$
$\to_\beta\ (\lambda y.T)\,F$
$\to_\beta\ T$

$\vee TT$
$=_{def}\ (\lambda x.\lambda y.xTy)\,TT$
$\to_\beta\ (\lambda y.TTy)\,T$
$\to_\beta\ TTT$
$=_{def}\ (\lambda x.\lambda y.x)\,TT$
$\to_\beta\ (\lambda y.T)\,T$
$\to_\beta\ T$

Conjunción

$$\wedge \quad =_{def} \quad \lambda x.\lambda y.xyF$$

Se deja como ejercicio al lector desarrollar la tabla de verdad de la conjunción.

Condicional if0

También es posible definir condicionales. Por ejemplo, se define el condicional $if0$. Si se aplica a un número n y éste resulta ser cero, se obtiene T; en caso contrario, se obtiene F. La definición de esta función se muestra a continuación:

$$if0 =_{def} \lambda x.xF\neg F$$

Veamos la reducción de $if00$ e $if01$:

$if00$
$=_{def}\ (\lambda x.xF\neg F)\,0$
$\to_\beta\ 0F\neg F$
$=_{def}\ (\lambda s.\lambda z.z)\,F\neg F$
$\to_\beta\ (\lambda z.z)\,\neg F$
$\to_\beta\ \neg F$
$\to_\beta\ T$

$if01$
$=_{def}\ (\lambda x.xF\neg F)\,1$
$\to_\beta\ 1F\neg F$
$=_{def}\ (\lambda s.\lambda z.sz)\,F\neg F$
$\to_\beta\ (\lambda z.Fz)\,\neg F$
$\to_\beta\ F\neg F$
$=_{def}\ (\lambda x.\lambda y.y)\,\neg F$
$\to_\beta\ (\lambda y.y)\,F$
$\to_\beta\ F$

Se deja como ejercicio al lector la definición de la función if.

3.6. Integrando números y booleanos

Con los naturales y booleanos definidos podemos definir otras funciones de utilidad.

Función predecesor

Para definir esta función construiremos un par de la forma $(n, n-1)$ y tomaremos el segundo elemento del par como resultado. Un par puede representarse mediante la función:

$$\lambda z.zab$$

La z permite realizar operaciones con el par. Por ejemplo, podemos obtener el primer y segundo elemento auxiliandonos de las funciones T (verdadero) y F (falso).

$$(\lambda z.zab)\, T \rightarrow_\beta Tab \rightarrow_\beta a$$

$$(\lambda z.zab)\, F \rightarrow_\beta Fab \rightarrow_\beta b$$

Podemos definir el par $(n+1, n)$ a partir del par $(n, n-1)$, representado por el parámetro p y extrayendo el primer elemento y aplicando la función sucesor. De esta forma definimos la función Φ como sigue:

$$\Phi =_{def} \lambda p.\lambda z.z\,(S\,(pT))\,(pT)$$

De esta forma el predecesor de un número se obtiene aplicando n veces la función Φ al par $\lambda z.00$ y seleccionando el segundo elemento del par usando la función F.

$$P =_{def} \lambda n.(n\Phi\,(\lambda z.z00))F$$

Es importante notar que bajo esta definición, el predecesor de cero es cero, con lo cual se evita la generación de números negativos. Se deja como ejercicio al lector obtener el predecesor de 1.

Operadores relacionales

Una vez definida la función predecesor, podemos definir una función que verifique si un número es mayor o igual que otro. Por ejemplo. para verificar $2 \geq 1$, basta con aplicar la función predecesor 2 veces a 1.

$$2P1 \rightarrow_\beta ... \rightarrow_\beta 0$$

Con la función $if0$ se verifica si esto se cumple. De esta forma, definimos la función $\geq$ como sigue:

$$\geq =_{def} \lambda x.\lambda y.Z\,(xPy)$$

De la misma forma, podemos definir la función igualdad $x = y$. Si se cumple $x \geq y$ y $y \geq x$, por lo tanto $x = y$. De esta forma, definimos la función $=$ que usa la operación $\wedge$ definida anteriormente.

$$==_{def} (\lambda x.\lambda y. \wedge (\geq xy) (\geq yx))$$

De forma similar se pueden definir otras operaciones relacionales.

3.7. ISWIM

El Cálculo λ fue diseñado por Church con el fin de dar un sistema formal que permitiera modelar las matemáticas. En los años 50 y 60 se descubrió la conexión entre los lenguajes de programación y varios aspectos del Cálculo λ gracias a un intento de dar la especificación formal del lenguaje de programación Algol 60. Sin embargo, el Cálculo λ resulta ser demasiado primitivo para considerarlo un lenguaje de programación, al igual que las Máquinas de Turing son demasiado primitivas para considerarlas computadoras reales.

Peter J. Landin definió el lenguaje de programación **ISWIM**[3]. Los fundamentos de su diseño fueron tomados del Cálculo λ como un punto de partida. **ISWIM** se define mediante la siguiente gramática similar a la del Cálculo λ:

```
<expr> ::= <var>
         | <cte>
         | <op> (<expr>+)
         | (λ<var>.<expr>)
         | (<expr> <expr>)
```

Por ejemplo, para el lenguaje **WAE** estudiado en la Nota de Clase 2:

```
<cte> ::= 1 | 2 | 3 | ...

<op> ::= + | -
```

Aunque son similares sintácticamente, la principal diferencia entre **ISWIM** y el Cálculo λ es la forma en que evalúan funciones. Por ejemplo, la siguiente expresión del Cálculo λ:

$$(\lambda x.1) (sub1\ \lambda y.y)$$

se reduce a 1. Esto ocurre debido a que las β-reducciones ignoran el parámetro real (no lo evalúan) y únicamente realizan la sustitución correspondiente. Esta forma de evaluación es llamada *paso de parámetros por nombre*[4] principalmente asociado con la evaluación perezosa. Sin embargo, muchos lenguajes de programación en la actualidad no hacen este tipo de evaluación y en su lugar usan una estrategia de evaluación llamada *paso de parámetros por valor*[5] asociada con la evaluación glotona, es decir, evalúan el parámetro real antes de pasarlo a la función correspondiente. **ISWIM** implementa paso por valor. Más adelante, en otras Notas se estudian estos conceptos.

En conclusión, aunque **ISWIM** nunca fue implementado como tal, éste ha servido de base para estudiar otras aplicaciones mediante la definición de *máquinas abstractas* o *virtuales*. La filosofía de este lenguaje de programación es la que siguen algunos lenguaje de programación hoy en día como son SCHEME y ML.

[3] Del inglés *If you See What I mean*.
[4] *call-by-name*.
[5] *call-by-value*.

3.8. Ejercicios

3.8.1. Sintaxis

1. Dados los siguientes términos λ, da una expresión equivalente por medio de currificación.

   (a) $\lambda xy.xy$

   (b) $\lambda xy.\lambda zw.xyzw$

   (c) $\lambda ab.\,(\lambda cd.ac)\,(\lambda ef.bf\,(\lambda xyz.e))$

2. Dadas las siguientes expresiones, da una α-equivalencia para cada una de forma tal que todas las variables de ligado sean distintas. Indica en cada caso las variables libres y ligadas.

   (a) $\lambda x.\lambda y.\lambda x.xy$

   (b) $(\lambda x.\lambda y.xy)\,(\lambda y.\lambda x.yx)$

   (c) $\lambda a.\lambda b.\,(\lambda x.\lambda y.ay)\,(\lambda y.\lambda x.by)$

3.8.2. Semántica

1. Obtén la β-reducción de las siguientes expresiones. Muestra en cada paso el redex y reducto correspondiente. En caso de que la expresión diverja, explica por qué.

   (a) $(\lambda x.x)\,(\lambda y.y)$

   (b) $(\lambda x.x)\,(\lambda y.y)\,z$

   (c) $(\lambda x.xx)\,(\lambda x.xx)$

   (d) $(\lambda x.\lambda y.xyy)\,(\lambda a.a)\,b$

   (e) $(\lambda z.zz)\,(\lambda y.yy)$

3.8.3. Representación de números y booleanos

1. Usando las definiciones para los numerales de Church:

   (a) Obtén la β-reducción de $S4$.

   (b) Obtén la β-reducción de $(2+3)\times 2$

2. Usando las definiciones para los booleanos:

   (a) Obtén la β-reducción de $\vee\,(\neg T)\,F$.

   (b) Obtén la β-reducción de $\wedge T\,(\vee TF)$

   (c) Obtén la β-reducción de $if_0\,(P\,(P2))$.

Referencias

[1] Lambda Calculi. A guide for computer scientists. Chris Hankin

[2] A Tutorial Introduction to the Lambda Calculus. Raúl Rojas. Web. 22 julio 2019 `https://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`

[3] Notas para el curso *Lenguajes de programación* impartido en la Facultad de Ciencias, UNAM. Favio E. Miranda Perea y Lourdes del Carmen Gonzalez Huesca.

[4] Programming Languages and Lambda Calculi, Matthias Felleisen, Matthew Flatt, 2003.