Formalización del teorema de Church-Rosser

Juan Agustín Bongiovanni

30 de julio de 2019

Índice general

Capítulo 1

Introduccion

Capítulo 2

Cálculo Lambda

2.1. Historia

Gottfried Wilhelm Leibniz tenía como ideal conseguir lo siguiente:
1) Crear un lenguaje universal en el que todos los problemas posiblen puedan estar.
2) Encontra un método con el que se pueda resolver todos los problemas establecidos en el lenguaje universal.
El problema 1 se puede cumplir adoptando alguna forma de teoría de conjuntos en el lenguaje de la lógica predicada de primer orden. El punto 2 trae una difícil pregunta: ¿se puede resolver todos los problemas formulados en el lenguaje universal? Esto se puede ver que no, pero no es muy claro probar eso. Esta pregunta se conoce como el Entscheidungsproblem. En 1936 con el objetivo de resolver el Entscheidungsproblem, Alonzo Church y Alan Turing adoptaron una formalización de la noción 'computable'. Church y Turing presentaron dos modelos de computación diferntes:
1) Church (1936) inventó un sistema formal llamado Cálculo Lambda y definió la idea de computabilidad a través de este sistema.
2) Turing (1936/2937) inventó una clase de máquinas (luego llamadas las máquinas de turing) y definió la idea de computabilidad basada en estas máquinas.
El cálculo lambda es un lenguaje universal ya que cualquier función computable puede ser expresada y evaluada a través de él, este lenguaje es capaz de decidir que función puede ser computable o no. Este tipo de demostraciones son equivalentes a las máquinas de Turing, con la diferencia de que el cálculo lambda no hace demasiado uso de reglas de transformación (usa sustituciones de variables) y no está pensado para que pueda ser implementado en maquinas reales.
Considerado el primer lenguaje funcional, este lenguaje es el fundamento de los lenguajes funcionales. El cálculo lambda se puede considerar como el lenguaje de programación más pequeño, ya que consiste en una regla de sustitución de variables, además de un esquema simple de definición de funciones.
El primer lenguaje funcional en aparecer fue LISP, diseñado en 1958 por John

McCarthy en el entorno de la computación simbólica. A este lenguaje siguieron otros como ML, el Miranda y el Haskell.

2.2. Introducción

Un programa funcional consiste en una expresion E la cual esta sujeta a reglas de reescritura. La reduccion se basa en reemplazar una parte P de E por otra expresión P' de acuerdo a reglas de reescritura dadas. La notacion seria la siguiente

$$E[P] \rightarrow E[P']$$

Este proceso de reducción se debe repetir hasta obtener una expresion la cual no tenga mas partes que puedan ser reemplazadas. A esta ultima se la llama forma normal E* de la expresion E, y es el dato de salida del programa funcional. Se puede dar un ejemplo, en el cual las reglas de reduccion consiste en las operaciones sobre numeros:

$$\begin{aligned}(20-5)+(11+8*2) &-> 15+(11+8*2)\\ &-> 15+(11+16)\\ &-> 15+27\\ &-> 42\end{aligned}$$

El sistema de reduccion usualmente satisface la propiedad Church-Rossier que establece que la forma normal obtenida es independiente del orden en el que se evaluen los subterminos. Tomando el mismo ejemplo, lo podemos ver:

$$\begin{aligned}(20-5)+(11+8*2) &-> (20-5)+(11+16)\\ &-> (20-5)+27\\ &-> 15+27\\ &-> 42\end{aligned}$$

o evaluando varias sub-expresiones al mismo tiempo:

$$\begin{aligned}(20-5)+(11+8*2) &-> 15+(11+16)\\ &-> 15+27\\ &-> 42\end{aligned}$$

Tenemos dos operaciones básicas en el cálculo lambda: la aplicación y la abstracción. La primera de ellas se denota a través de la expresion F.A o FA, donde la F es considerada un algoritmo al cual se le aplica como entrada el dato A. Es de tipo libre, esto permite considerar expresión como por ejemplo FF, donde F es aplicada a si misma. Esto es útil para simular la recursión. La otra operación basica es la abstracción. Si M $\equiv$ M[x] es una expresión que contiene a x, entonces λx.M[x] denota la función $x \mapsto$ M[x]. Estas dos operaciones trabajan juntas. Esto lo podemos ver en la siguiente fórmula:

$$(\lambda x, 2 * x + 1)3 = 2 * 3 + 1 (= 7)$$

2.3. Definición formal

La sintaxis oficial del calculo lambda esta contenida en la siguiente definición:

Definición 1. *El alfabeto del cálculo lambda contiene los parentesis y corchetes izquiedros y derechos, el simbolo λ y un conjunto infinito de variables. Los términos de tipo lambda se definen inductivamente de la siguiente manera:*
1. Toda variable es un λ-términos
2.Si M y N son λ-términos, entonces (MN) lo es también
3.Si M es un λ-término y x es una variable, entonces (λx[M]) es un λ-término

Los términos formados a partir de la regla (2) son llamados términos de aplicación, mientras que los formados a partir de la regla (3) son término de abstracción. Para poder omitir paréntesis y ahorrarse escribir paréntesis innecesarios se adopta la convensión de asociación a la izquierda, por ejemplo cuando se juntan mas de dos términos como M1M2M3...Mn se pueden recuperar los paréntesis faltantes asociando a la izquierda y quedaría ((M1M2)M3)...Mn
Como ejemplo de expresiones lambda podemos ver como se codifican estructuras de datos conocidas:

- Números: Los números de Church son una representación de los números naturales. La función que representa al número natural n es una función que asigna cualquier función a su composición en n. Es decir, el valor del número es igual a la cantidad de veces que la función encapsula al argumento. Todos los números de Church son funciones de dos parámetros, empezando por el 0 al que no se le aplica ninguna función.

  - 0 = λf.$\lambda x.x$
  - 1 = λf.λx.fx
  - 2 = λf.λx.f(fx)
  - 3 = λf.λx.f(f(fx))

- Valores booleanos: Estos valores se pueden definir en lambda de la siguiente manera:

  - true = $\lambda x.\lambda y.x$
  - false= $\lambda x.\lambda y.y$

  Por ejemplo la sentencia IF se escribe de la siguiente forma: λp.λa.λb.pab
  Si p es True será ($\lambda x.\lambda y.x$)(ab) = a
  Si p es True será ($\lambda x.\lambda y.y$)(ab) = b

- Listas: Consisten en un par formado por la cabeza (head) de la lista y su cola (tail). A este par lo podemos representar de la siguiente manera: (λf.((fh)t)

Las variables en las expresiones del cálculo lambda pueden ser tanto libres como ligadas. Se puede definir la noción de variables libres y ligadas de la siguiente manera:

Definición 2. *Las funciones sintácticas FV and BV (variables libres y variables ligadas, respectivamente) son definididas en el conjunto de λ-términos de forma inductiva:*
Para cualquier variable x y términos M y N:
1. FV(x)=x
2. FV(MN)=FV(M)∪FV(N)
3. FV(λx[M])=FV(M)-x

1. BV(x)=/emptyset
2. BV(MN)=BV(M)∪BV(N)
3. BV(λx[M])=BV(M)∪x

A continuación, definimos sustitución:

Definición 3 (Sustitución)**.** *Escribimos M[x:=N] para denotar la sustitución de N por las ocurrencias libres de x en M. Una definición precisa por recursion en el conjunto de λ-términos es la siguiente: para todos términos A,B y M, y para todas variables x,y tenemos:*
1. x[x:=M]≡M
2. y[x:=M]≡y (y distinto a x)
3. (AB)[x:=M]≡A[x:=M]B[x:=M]
4. (λx[A])[x:=M]≡λx[A]
5. (λy[A])[x:=M]≡[A[x:=M]] (y distinto a x)

La parte (1) dice que si queremos sustituit M por x y estamos simplemente tenemos a x, entonces el resultado será M. La parte (2) dice que no pasa nada cuando queremos sustituir x y estamos tratando con una variable distinta. La cláusula (3) dice que la sustitución se distribuye en la aplicación. Las cláusulas (4) y (5) se refieren a términos de abstracción y cláusulas paralelas (1) y (2): si la variable enlazada z del término de abstracción λz[A] es idéntico a la variable x para la que debemos aplicar la sustitución entonces no realizamos ninguna sustitución. Esto es porque M[x:=N] denota la sustitución de N por las ocurrencias libres de x en M. Si M es un término de abstracción λx[A] cuya variable ligada es x, entonces x no ocurre libremente en M, por lo que no hay nada para hacer. Esto explica la cláusula (4). En cambio, en la (5) si la variable ligada de un término de abstracción difiere de x, entonces al menos tiene la posibilidad de ocurrir libremente en el término de abstracción, y la sustitución continúa en el cuerpo de este término.

Definición 4 (Cambio de variable ligadas, α-conversión)**.** *El término N es obtenido del término M mediante el cambio de variable ligada si cualquier término de asbtracción λx[A] dentro de M ha sido reemplazado por λy[A[x:=y]]*

Decimos que dos términos M y N son α-convertibles si hay una secuencia de cambios de variables ligadas empezando por M y que termina en N.

Reducción

Hay varias nociones de reducción que son válidas en el cálculo lambda, pero la principal es /*beta*-reducción

Definición 5 (Un paso de β-reducción)**.** *Para λ-términos A y B, decimos que A se β-reduce en un paso a B, si existe un subtérmino C en A, una variable x y λ-términos M y N tal que C≡(λ[M])N y B es A a expeción de que una ocurrencia de C en A sea reemplazada por M[x:=N].*

Podemos ver varios ejemplos:

1) (λx[x])a -¿β a 2) (λx[y])a -¿β y 3) (λy[y5])(λx[3*x]) -¿β (λx[3*x])5 -¿β 3*5 4) El término (λx[(λy[xy])a])b se puede reducir en un paso en dos diferentes λ-términos: (λx[(λy[xy])a])b -¿β (λy[by])a y (λx[(λy[xy])a])b] -¿β (λx[xa])b

Vemos que la reducción no es otra cosa que el reemplazo textual de un parámetro formal en el cuerpo de una función por el parámetro real dado. Si una secuencia de reducciones ha llegado a su fin y no es posible realizar mas reducciones, decimos que el término se ha reducido a su forma normal. Uno esperaría que un término después de una serie de reducciones llegue a una forma donde no sea posible aplicar más reducciones. Pero esto no siempre es posible. Se puede ver esto con este ejemplo:

(λx[xx])(λx[xx])

Este término siempre se reduce a si mismo. Este término, como muchos otros, no tiene una forma normal.

2.4. Teorema Church-Rosser

Es posible que un término ofrezca muchas oportunidades de reducción al mismo tiempo. Para que todo el cálculo tenga sentido, es necesario que el resultado de la computación sea independiente del orden de reducción. Es necesario expresar esta propiedad para todos los términos, no sólo para aquellos que tengan una formal normal. Esto es posible con el siguiente teorema:

Teorema 1 (Church-Rosser)**.** *Si un término M puede reducirse (en varios pasos) a términos N y P, entonces existe un término Q al que tanto como N y P pueden reducirse (en varios pasos).*

Imagen rombo

Capítulo 3

Agda

Capítulo 4

Sintaxis

Para poder demostrar el teorema de Church-Rosser en el lenguaje Agda, es necesario ir definiendo y creando los tipos de datos necesarios para llegar a esto.

4.1. Listas

Una estructura de datos que utilizaré, por ejemplo para la definición de variables libres, son las listas. En Agda, las defino de la siguiente manera:

```
data List (A : Set) : Set where
[ ]                 : List A
_::_                : A -> List A -> List A
```

También es necesario definir funciones sobre las listas. Una de ellas es la función $\in$ la cual toma como parámetros una variable y una lista, y devuelve true o false, según esa variable pertenezca a la lista o no, respectivamente:

```
_ ∈ _ : V -> List V -> Bool
x ∈ [] = false
x ∈ (y::ys) with x=y
... | true  = true
... | false = x \in ys
```

Otra función necesaria es la de adjuntar dos listas, la cual toma como parámetro dos listas y tiene como salida otra de ellas:

```
_ +++ _ : List V -> List V -> List V
ys       +++ [ ] = ys
(x :: xs) +++ (y :: ys) with x = y
... | true = x :: (xs +++ ys)
... | false = x :: (y :: ( xs +++ ys))
```

La última función sobre las listas es la que elimina una variable de esta estructura. Como entrada toma una lista y una variable, y la salida será una lista:

```
_ - _ : List V -> V -> List V
(x :: xs) - s with x = s
... | true = xs - s
... | false = x :: (xs - s)
```

4.2. Sintaxis abstracta y variables libres

La sintaxis abstracta del cálculo lambda en Agda, se puede definir de la siguiente manera:

```
data Expr : Set where
Var       : V
App       : Expr -> Expr -> Expr
Lamb      : V -> Expr -> Expr
```

También podemos deifnir al conjunto de variables libres de una expresión. En Agda a las variables libres las podemos expresar de dos formas, una de ellas es como una función la cual toma como argumento a una expresión lambda y devuelve una lista de sus variables libres, para esto utilizamos la definición de listas dada anteriormente:

```
FreeVList : Expr -> List V
FreeVList (Var s) = s :: [ ]
FreeVList (App e1e2) = FreeVList e1 +++ FreeVList e2
FreeVList (Lamb s e1) = FreeVList e1 - s
```

Otra forma de ver las variables libres es definiendo una relación de dos elementos, donde uno de ellos es una variable y el otro será una expresion lambda. Esta relación sólo estará definida si esa variable es una variable libre de dicha expresión. Para esto debo escribir todos los casos posibles para la expresiones lambda, con todos sus constructores. La relacion quedaría de la siguiente forma:

```
data _ FreeV _ : V -> Expr -> Set where
var : {x y : V} -> x = y ->
        x FreeV (Var y)
appl : {x : V} {e e' : Expr} -> x FreeV e ->
        x FreeV (App e e')
appr : {x : V} {e e' : Expr} -> x FreeV e' ->
        x FreeV (App e e')
abs  : {x y : V} {e : Expr} -> x FreeV e -> (x = y -> ⊥) ->
        x FreeV (Lamb y e)
```

Lo siguiente sería realizar dos pruebas con ambas definiciones. Una de completitud que demuestre que si existe v FreeV e entonces v pertenece a la lista de variables libres de e. Y luego una prueba de correccion, la cual muestra que para toda variable v perteneciente a lista de variables libres de e, existe la relacion v FreeV e. Las demostraciones también las puedo realizar en Agda:

FALTA LAS DEMOSTRACIONES

Capítulo 5

Teorema Church-Rosser

Capítulo 6

Conclusiones