Una Formalización del Cálculo Lambda en Lean

Edwin Daniel Patiño Osorio

2025-06-26

Universidad de Antioquia

Índice

1. Introducción al Cálculo Lambda 2
   - 1.1 ¿Que es el cálculo Lambda? 3
   - 1.2 ¿Una generalización de las funciones? 4
   - 1.3 Definición formal 5
   - 1.4 Variables libres 6
   - 1.5 Sustitución 7
   - 1.6 Beta reducción 8
   - 1.7 Esquema de la formalización del Church-Rosser 10
2. Sobre Formalizar 11
   - 2.1 El motivo 12
   - 2.2 Ejemplo 13
3. Indices de Bruijn 14
   - 3.1 Definición 15
   - 3.2 Shift y Unshift 17
   - 3.3 Sustitución 19
   - 3.4 Beta reducción 21
   - 3.5 Rango 24
   - 3.6 Objetivos y cosas por hacer 25
   - 3.7 Repositorio 26

Bibliografía 27

Índice

1. Introducción al Cálculo Lambda 2
   - 1.1 ¿Que es el cálculo Lambda? 3
   - 1.2 ¿Una generalización de las funciones? 4
   - 1.3 Definición formal 5
   - 1.4 Variables libres 6
   - 1.5 Sustitución 7
   - 1.6 Beta reducción 8
   - 1.7 Esquema de la formalización del Church-Rosser 10
2. Sobre Formalizar 11
   - 2.1 El motivo 12
   - 2.2 Ejemplo 13
3. Indices de Bruijn 14
   - 3.1 Definición 15
   - 3.2 Shift y Unshift 17
   - 3.3 Sustitución 19
   - 3.4 Beta reducción 21
   - 3.5 Rango 24
   - 3.6 Objetivos y cosas por hacer 25
   - 3.7 Repositorio 26

Bibliografía 27

1.1 ¿Que es el cálculo Lambda?

El cálculo Lambda es un modelo de computación creado por el matemático Alonzo Church, el cual posee las siguientes particularidades:

- Es declarativo,
- Generaliza las nociones de función.
- Su idea principal está en las aplicaciones, sustituciones, y abstracciones.

Figura 1: Foto de Alonzo Church

¿Una generalización de las funciones?

1.2 ¿Una generalización de las funciones?

1.2 ¿Una generalización de las funciones?

- Las funciones no tienen nombre.

1.2 ¿Una generalización de las funciones?

- Las funciones no tienen nombre.

Una función como $f(x) = 3x + 2$ se denota por $\lambda x.3x + 2$.

1.2 ¿Una generalización de las funciones?

- Las funciones no tienen nombre.
- Todas las funciones son de una sola variable

1.2 ¿Una generalización de las funciones?

- Las funciones no tienen nombre.
- Todas las funciones son de una sola variable

En lugar de escribir $f(x, y) = x + y$ escribimos $\lambda x.\lambda y.x + y$, y se trata de una función que al ser evaluada en x retorna una función que depende solo de y, a esto se le conoce como currificación.

1.2 ¿Una generalización de las funciones?

- Las funciones no tienen nombre.
- Todas las funciones son de una sola variable
- Las funciones pueden ser evaluadas con cualquier término (es decir no hay restricción sobre el dominio).

1.2 ¿Una generalización de las funciones?

- Las funciones no tienen nombre.
- Todas las funciones son de una sola variable
- Las funciones pueden ser evaluadas con cualquier término (es decir no hay restricción sobre el dominio).

Esto tiene como consecuencia que toda función tiene un punto fijo,
es decir funciones como $\lambda x.x + 1$ tiene un punto fijo.
(Combinador de punto fijo Y)

1.2 ¿Una generalización de las funciones?

- Las funciones no tienen nombre.
- Todas las funciones son de una sola variable
- Las funciones pueden ser evaluadas con cualquier término (es decir no hay restricción sobre el dominio).

> **Observación**
>
> **Es una generalización del concepto de función**, pero no podemos decir que todas las funciones son de esta forma, pues el cálculo lambda es un modelo de computación, por lo que hay funciones que no son λ-definibles.

1.3 Definición formal

Definición 1.3.1 (Lambda Términos) Sea $\mathbb{V} = \{x, y, z, \ldots\}$ un conjunto de variables infinito, denotamos por Λ al conjunto de todos los lambda términos, el cual se define como:

1. (Variable) Si $x \in \mathbb{V}$, entonces $x \in \Lambda$,
2. (Aplicación) Si $M, N \in \Lambda$, entonces $(MN) \in \Lambda$,
3. (Abstracción) Si $x \in \mathbb{V}$ y $M \in \Lambda$, entonces $(\lambda x.M) \in \Lambda$

1.3 Definición formal

Definición 1.3.1 (Lambda Términos) Sea $\mathbb{V} = \{x, y, z, \ldots\}$ un conjunto de variables infinito, denotamos por Λ al conjunto de todos los lambda términos, el cual se define como:

1. (Variable) Si $x \in \mathbb{V}$, entonces $x \in \Lambda$,
2. (Aplicación) Si $M, N \in \Lambda$, entonces $(MN) \in \Lambda$,
3. (Abstracción) Si $x \in \mathbb{V}$ y $M \in \Lambda$, entonces $(\lambda x.M) \in \Lambda$

Ejemplo. $(\lambda x.x)x$, $(\lambda x.\lambda y.xyz)(\lambda x.y)$, $(\lambda x.xx)(\lambda x.xx)$, $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

1.3 Definición formal

Definición 1.3.1 (Lambda Términos) Sea $\mathbb{V} = \{x, y, z, \ldots\}$ un conjunto de variables infinito, denotamos por Λ al conjunto de todos los lambda términos, el cual se define como:

1. (Variable) Si $x \in \mathbb{V}$, entonces $x \in \Lambda$,
2. (Aplicación) Si $M, N \in \Lambda$, entonces $(MN) \in \Lambda$,
3. (Abstracción) Si $x \in \mathbb{V}$ y $M \in \Lambda$, entonces $(\lambda x.M) \in \Lambda$

Observación

Términos como $\lambda x.\lambda y.x + y$ no son validos, pues $+$ no pertenece al lenguaje, pero eso no significa que no podamos definir un λ–término correspondiente a la suma.

1.4 Variables libres

Definición 1.4.1 (Variables libres) Se define $\mathrm{FV}(M)$ como el conjunto de variables libres de M como sigue:

1.4 Variables libres

Definición 1.4.1 (Variables libres) Se define $\mathrm{FV}(M)$ como el conjunto de variables libres de M como sigue:

- $\mathrm{FV}(x) = \{x\}$,

1.4 Variables libres

Definición 1.4.1 (Variables libres) Se define $\mathrm{FV}(M)$ como el conjunto de variables libres de M como sigue:

- $\mathrm{FV}(x) = \{x\}$,
- $\mathrm{FV}(\lambda x.P) = \mathrm{FV}(P) \setminus \{x\}$,
- $\mathrm{FV}(PQ) = \mathrm{FV}(P) \cup \mathrm{FV}(Q)$.

1.5 Sustitución

Definición 1.5.1 (Sustitución) Sean $M, N \in \Lambda$ y $x \in \mathbb{V}$, denotamos $M[x := N]$, por la operación de reemplazar todas las ocurrencias libres de x en M por N. Formalmente hablando:

1.5 Sustitución

Definición 1.5.1 (Sustitución) Sean $M, N \in \Lambda$ y $x \in \mathbb{V}$, denotamos $M[x := N]$, por la operación de reemplazar todas las ocurrencias libres de x en M por N. Formalmente hablando:

- $x[x := N] \equiv N$, $y[x := N] \equiv y$,

1.5 Sustitución

Definición 1.5.1 (Sustitución) Sean $M, N \in \Lambda$ y $x \in \mathbb{V}$, denotamos $M[x := N]$, por la operación de reemplazar todas las ocurrencias libres de x en M por N. Formalmente hablando:

- $x[x := N] \equiv N$, $y[x := N] \equiv y$,
- $PQ[x := N] \equiv (P[x := N])(Q[x := N])$,

1.5 Sustitución

Definición 1.5.1 (Sustitución) Sean $M, N \in \Lambda$ y $x \in \mathbb{V}$, denotamos $M[x := N]$, por la operación de reemplazar todas las ocurrencias libres de x en M por N. Formalmente hablando:

- $x[x := N] \equiv N$, $y[x := N] \equiv y$,
- $PQ[x := N] \equiv (P[x := N])(Q[x := N])$,
- $(\lambda x.P)[x := N] \equiv \lambda x.P$,

1.5 Sustitución

Definición 1.5.1 (Sustitución) Sean $M, N \in \Lambda$ y $x \in \mathbb{V}$, denotamos $M[x := N]$, por la operación de reemplazar todas las ocurrencias libres de x en M por N. Formalmente hablando:

- $x[x := N] \equiv N$, $y[x := N] \equiv y$,
- $PQ[x := N] \equiv (P[x := N])(Q[x := N])$,
- $(\lambda x.P)[x := N] \equiv \lambda x.P$,
- $(\lambda y.P)[x := N] \equiv \lambda x.P[x := N]$

1.5 Sustitución

Definición 1.5.1 (Sustitución) Sean $M, N \in \Lambda$ y $x \in \mathbb{V}$, denotamos $M[x := N]$, por la operación de reemplazar todas las ocurrencias libres de x en M por N. Formalmente hablando:

- $x[x := N] \equiv N$, $y[x := N] \equiv y$,
- $PQ[x := N] \equiv (P[x := N])(Q[x := N])$,
- $(\lambda x.P)[x := N] \equiv \lambda x.P$,
- $(\lambda y.P)[x := N] \equiv \lambda x.P[x := N]$

Ejemplo.

$$(\lambda x.\lambda y.xyz)[z := \lambda w.w] \equiv \lambda x.\lambda y.xy(\lambda w.w)$$

1.6 Beta reducción

La **beta-reducción** es la regla esencial del cálculo lambda: es lo que convierte a este formalismo en un verdadero modelo de computación. Gracias a ella podemos evaluar (reducir) términos lambda, lo que nos permite implementar operaciones aritméticas, definir funciones y, en última instancia, programar. Sin la beta-reducción no habría mecanismo para "hacer correr" un programa en forma de términos lambda; por eso constituye el corazón del poder computacional del cálculo lambda.

1.6 Beta reducción

Definición 1.6.1 (Beta reducción de un solo paso)

- $(\lambda x.P)Q \to_\beta P[x := Q]$,
- Si $P \to_\beta Q$ y $R \in \Lambda$, entonces $PR \to_\beta QR$ y $RP \to_\beta RQ$,
- Si $x \in \mathbb{V}$ y $P \to_\beta Q$, entonces $\lambda x.P \to_\beta \lambda x.Q$

1.6 Beta reducción

Definición 1.6.1 (Beta reducción de un solo paso)

- $(\lambda x.P)Q \to_\beta P[x := Q]$,
- Si $P \to_\beta Q$ y $R \in \Lambda$, entonces $PR \to_\beta QR$ y $RP \to_\beta RQ$,
- Si $x \in \mathbb{V}$ y $P \to_\beta Q$, entonces $\lambda x.P \to_\beta \lambda x.Q$

$$(\lambda x.x)N \rightarrow_\beta N \text{ (Función identidad)}$$

$$(\lambda x.\lambda y.x)NM \to_\beta N \text{ (Proyección)}$$

Definimos

$$0 = \lambda f.\lambda x.x$$

Definimos

$$0 = \lambda f.\lambda x.x$$

$$1 = \lambda f.\lambda x.f\,x$$

Definimos

$$0 = \lambda f.\lambda x.x$$

$$1 = \lambda f.\lambda x.f\,x$$

$$2 = \lambda f.\lambda x.f\,(f\,x)$$

Definimos

$$0 = \lambda f.\lambda x.x$$

$$1 = \lambda f.\lambda x.f\,x$$

$$2 = \lambda f.\lambda x.f(f\,x)$$

$$n = \lambda f.\lambda x.f^n(x)$$

Definimos

$0 = \lambda f.\lambda x.x$

$1 = \lambda f.\lambda x.fx$

$2 = \lambda f.\lambda x.f(fx)$

$n = \lambda f.\lambda x.f^n(x)$

Y con ello definimos

$\text{Succ} = \lambda nfx.f(nfx)$

$\text{Add} = \lambda mnfx.mf(nfx)$

$\text{Mult} = \lambda mnfx.m(nf)x$

$\text{Exp} = \lambda mnfx.mnfx$

1.6 Beta reducción

Definición 1.6.2 (Beta reducción de más de un paso) Decimos que $M \twoheadrightarrow_\beta N$ si existe un $n \geq 0$ y términos M_i, con $0 \leq i \leq n$, tal que

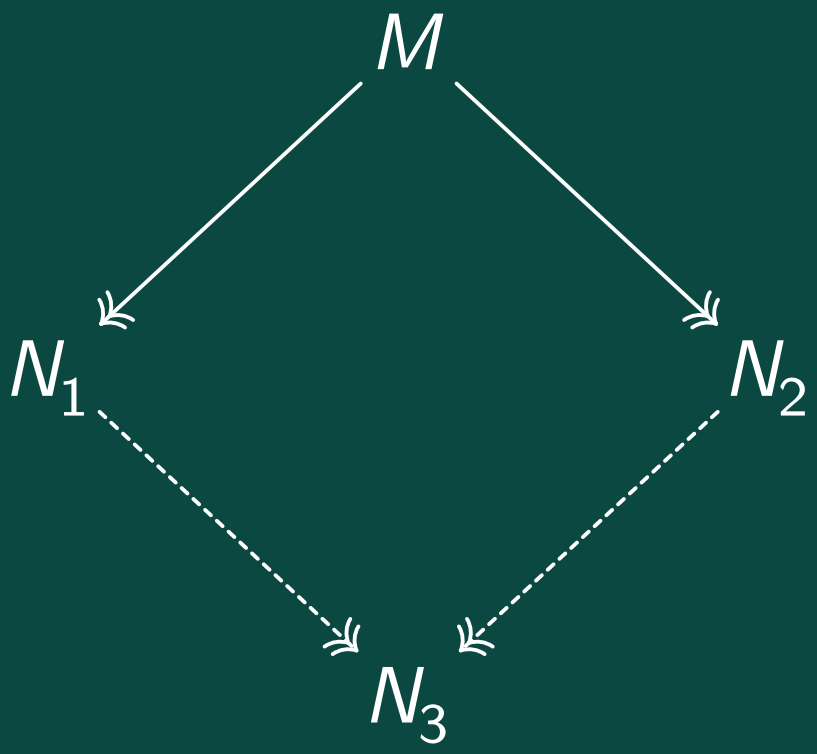
$$M \equiv M_0 \to_\beta M_1 \to_\beta \ldots \to_\beta M_{n-1} \to_\beta M_n \equiv N.$$

1.6 Beta reducción

Definición 1.6.2 (Beta reducción de más de un paso) Decimos que $M \twoheadrightarrow_\beta N$ si existe un $n \geq 0$ y términos M_i, con $0 \leq i \leq n$, tal que

$$M \equiv M_0 \to_\beta M_1 \to_\beta \ldots \to_\beta M_{n-1} \to_\beta M_n \equiv N.$$

Teorema 1.6.3 (Church-Rosser, Confluencia) Sea $M \in \Lambda$, tal que $M \twoheadrightarrow_\beta N_1$ y $M \twoheadrightarrow_\beta N_2$, entonces existe un N_3 tal que $N_1 \twoheadrightarrow_\beta N_3$ y $N_2 \twoheadrightarrow_\beta N_3$.



Este teorema asegura que no importa como hagamos las beta reducciones, siempre obtendremos un mismo resultado, esto es fundamental, pues queremos que nuestras funciones sean deterministas.

1.7 Esquema de la formalización del Church-Rosser

1.7 Esquema de la formalización del Church-Rosser

1. Si $\to_1$ y $\to_2$ son dos relaciones, tal que $\to_1 \subseteq \to_2$ y $\to_2$ es la extensión reflexiva transitiva de $\to_1$, y $\to_1$ es confluente entonces $\to_2$ es confluente.

1.7 Esquema de la formalización del Church-Rosser

1. Si $\to_1$ y $\to_2$ son dos relaciones, tal que $\to_1 \subseteq \to_2$ y $\to_2$ es la extensión reflexiva transitiva de $\to_1$, y $\to_1$ es confluente entonces $\to_2$ es confluente.
2. Se define una relación $\Rightarrow_\beta$ sobre Λ llamada reducción beta paralela, tal que $\to_\beta \subsetneq \Rightarrow_\beta \subsetneq \twoheadrightarrow_\beta$, y de modo que $\twoheadrightarrow_\beta$ sea la extensión transitiva reflexiva de $\Rightarrow_\beta$.

1.7 Esquema de la formalización del Church-Rosser

1. Si $\to_1$ y $\to_2$ son dos relaciones, tal que $\to_1 \subseteq \to_2$ y $\to_2$ es la extensión reflexiva transitiva de $\to_1$, y $\to_1$ es confluente entonces $\to_2$ es confluente.
2. Se define una relación $\Rightarrow_\beta$ sobre Λ llamada reducción beta paralela, tal que $\to_\beta \subsetneq \Rightarrow_\beta \subsetneq \twoheadrightarrow_\beta$, y de modo que $\twoheadrightarrow_\beta$ sea la extensión transitiva reflexiva de $\Rightarrow_\beta$.
3. Se muestra que $\Rightarrow_\beta$ es confluente, para concluir que $\twoheadrightarrow_\beta$ es confluente. [1]

1.7 Esquema de la formalización del Church-Rosser

1. Si $\to_1$ y $\to_2$ son dos relaciones, tal que $\to_1 \subseteq \to_2$ y $\to_2$ es la extensión reflexiva transitiva de $\to_1$, y $\to_1$ es confluente entonces $\to_2$ es confluente.
2. Se define una relación $\Rightarrow_\beta$ sobre Λ llamada reducción beta paralela, tal que $\to_\beta \subsetneq \Rightarrow_\beta \subsetneq \twoheadrightarrow_\beta$, y de modo que $\twoheadrightarrow_\beta$ sea la extensión transitiva reflexiva de $\Rightarrow_\beta$.
3. Se muestra que $\Rightarrow_\beta$ es confluente, para concluir que $\twoheadrightarrow_\beta$ es confluente. [1]

> **Observación**
>
> $\to_\beta$ (beta reducción de un solo paso) no es confluente, tome el término $(\lambda x.xx)((\lambda x.x)R)$.

Índice

1. Introducción al Cálculo Lambda 2
   - 1.1 ¿Que es el cálculo Lambda? 3
   - 1.2 ¿Una generalización de las funciones? 4
   - 1.3 Definición formal 5
   - 1.4 Variables libres 6
   - 1.5 Sustitución 7
   - 1.6 Beta reducción 8
   - 1.7 Esquema de la formalización del Church-Rosser 10
2. Sobre Formalizar 11
   - 2.1 El motivo 12
   - 2.2 Ejemplo 13
3. Indices de Bruijn 14
   - 3.1 Definición 15
   - 3.2 Shift y Unshift 17
   - 3.3 Sustitución 19
   - 3.4 Beta reducción 21
   - 3.5 Rango 24
   - 3.6 Objetivos y cosas por hacer 25
   - 3.7 Repositorio 26

Bibliografía 27

2.1 El motivo

- Hay proposiciones muy difíciles, por lo que intentar demostrarlas se vuelve un reto, y cometer errores es sencillo.

2.1 El motivo

- Hay proposiciones muy difíciles, por lo que intentar demostrarlas se vuelve un reto, y cometer errores es sencillo.
- Se han encontrado errores en demostraciones modernas y de hace más de 100 años.

2.1 El motivo

- Hay proposiciones muy difíciles, por lo que intentar demostrarlas se vuelve un reto, y cometer errores es sencillo.
- Se han encontrado errores en demostraciones modernas y de hace más de 100 años.
- Las matemáticas cada vez son más complejas, por lo que se convierte en una labor difícil.

2.1 El motivo

- Hay proposiciones muy difíciles, por lo que intentar demostrarlas se vuelve un reto, y cometer errores es sencillo.
- Se han encontrado errores en demostraciones modernas y de hace más de 100 años.
- Las matemáticas cada vez son más complejas, por lo que se convierte en una labor difícil.
- Los compiladores de los lenguajes de programación son muy buenos encontrando errores, ¿Porque no aprovecharnos de esto para verificar demostraciones?

2.1 El motivo

- Hay proposiciones muy difíciles, por lo que intentar demostrarlas se vuelve un reto, y cometer errores es sencillo.
- Se han encontrado errores en demostraciones modernas y de hace más de 100 años.
- Las matemáticas cada vez son más complejas, por lo que se convierte en una labor difícil.
- Los compiladores de los lenguajes de programación son muy buenos encontrando errores, ¿Porque no aprovecharnos de esto para verificar demostraciones?

Formalizar las matemáticas consiste en expresar definiciones y demostraciones mediante lenguajes de programación especializados, como Lean o Coq, que permiten verificar automáticamente su corrección. Esto garantiza rigor, elimina ambigüedades y facilita la reutilización de resultados.

2.2 Ejemplo

Teorema 2.2.1 Un natural n es par, si existe un $k \in \mathbb{N}$ tal que $n = 2k$. Si n y m son pares, entonces $n + m$ es par.

2.2 Ejemplo

Teorema 2.2.1 Un natural n es par, si existe un $k \in \mathbb{N}$ tal que $n = 2k$. Si n y m son pares, entonces $n + m$ es par.

Demostración. Como n, m son pares, entonces existen $k_1, k_2 \in \mathbb{N}$ tal que $n = 2k_1$ y $m = 2k_2$, luego $m + n = 2(k_1 + k_2)$. □

2.2 Ejemplo

Teorema 2.2.1 Un natural n es par, si existe un $k \in \mathbb{N}$ tal que $n = 2k$. Si n y m son pares, entonces $n + m$ es par.

Demostración. Como n, m son pares, entonces existen $k_1, k_2 \in \mathbb{N}$ tal que $n = 2k_1$ y $m = 2k_2$, luego $m + n = 2(k_1 + k_2)$. □

```lean
def Par (n : Nat) := ∃ k : Nat, n = 2 * k
example (n m : Nat) (nPar : Par n) (mPar : Par m) : Par (n + m) := by
  obtain ⟨k₁, (hn : n = 2 * k₁)⟩ := nPar
  obtain ⟨k₂, (hm : m = 2 * k₂)⟩ := mPar
  exists k₁ + k₂
  rw [hn, hm, ← Nat.mul_add]
```

¿Entonces formalizar solo es traducir demostraciones ya hechas?

¿Entonces formalizar solo es traducir demostraciones ya hechas?

No, en el proceso de formalizar se pueden crear nuevas matemáticas.

Índice

1. Introducción al Cálculo Lambda 2
   - 1.1 ¿Que es el cálculo Lambda? 3
   - 1.2 ¿Una generalización de las funciones? 4
   - 1.3 Definición formal 5
   - 1.4 Variables libres 6
   - 1.5 Sustitución 7
   - 1.6 Beta reducción 8
   - 1.7 Esquema de la formalización del Church-Rosser 10
2. Sobre Formalizar 11
   - 2.1 El motivo 12
   - 2.2 Ejemplo 13
3. Indices de Bruijn 14
   - 3.1 Definición 15
   - 3.2 Shift y Unshift 17
   - 3.3 Sustitución 19
   - 3.4 Beta reducción 21
   - 3.5 Rango 24
   - 3.6 Objetivos y cosas por hacer 25
   - 3.7 Repositorio 26

Bibliografía 27

3.1 Definición

Definición 3.1.1

- Si $n \in \mathbb{N}$ entonces $n \in \Lambda$,
- Si M y N son términos, entonces $(MN) \in \Lambda$,
- Si M es un término, entonces $(\lambda M) \in \Lambda$.

3.1 Definición

Definición 3.1.1

- Si $n \in \mathbb{N}$ entonces $n \in \Lambda$,
- Si M y N son términos, entonces $(MN) \in \Lambda$,
- Si M es un término, entonces $(\lambda M) \in \Lambda$.

λ-término convencional	λ-término con indices
$\lambda x.x$	$\lambda 0$
$\lambda x.\lambda y.x$	$\lambda\lambda 1$
$\lambda x.x(\lambda y.x)$	$\lambda 0(\lambda 1)$
$\lambda x.\lambda y.xyz(\lambda w.wx)$	$\lambda\lambda 102(\lambda 02)$

3.1 Definición

Observación

- Surgió para aliviar la complejidad de formalizar operaciones sobre términos del cálculo λ (por ejemplo, la sustitución de variables), que exige un manejo delicado de variables libres y ligadas.
- Al reemplazar nombres por números que indican la profundidad de su λ-binder, se evita toda ambigüedad y colisión de nombres, simplificando la α-conversión a una simple comparación numérica de estructuras.

3.1 Definición

> **Observación**
>
> - Surgió para aliviar la complejidad de formalizar operaciones sobre términos del cálculo λ (por ejemplo, la sustitución de variables), que exige un manejo delicado de variables libres y ligadas.
> - Al reemplazar nombres por números que indican la profundidad de su λ-binder, se evita toda ambigüedad y colisión de nombres, simplificando la α-conversión a una simple comparación numérica de estructuras.

```lean
inductive Lambda
  | var : Nat → Lambda
  | abs : Lambda → Lambda
  | app : Lambda → Lambda → Lambda
```

3.2 Shift y Unshift

Definición 3.2.1 (Shift)

$$\uparrow_c^i (n) = \begin{cases} n & n < c \\ n + i & n \geq c \end{cases}$$

$$\uparrow_c^i (MN) = (\uparrow_c^i M)(\uparrow_c^i N)$$

$$\uparrow_c^i (\lambda M) = \lambda(\uparrow_{c+1}^i M)$$

```lean
def shift (c i : Nat) : Lambda → Lambda
  | var n =>
    if n < c then var n
    else var (n + i)
  | app e₁ e₂ =>
    app (e₁.shift c i) (e₂.shift c i)
  | abs e => abs (e.shift (c + 1) i)
```

3.2 Shift y Unshift

Definición 3.2.1 (Shift)

$$\uparrow_c^i (n) = \begin{cases} n & n < c \\ n + i & n \geq c \end{cases}$$

$$\uparrow_c^i (MN) = (\uparrow_c^i M)(\uparrow_c^i N)$$

$$\uparrow_c^i (\lambda M) = \lambda(\uparrow_{c+1}^i M)$$

```lean
def shift (c i : Nat) : Lambda → Lambda
  | var n =>
    if n < c then var n
    else var (n + i)
  | app e₁ e₂ =>
    app (e₁.shift c i) (e₂.shift c i)
  | abs e => abs (e.shift (c + 1) i)
```

La función shift, suma i a cada variable libre que sea mayor o igual que c.

3.2 Shift y Unshift

Definición 3.2.2 (Unshift)

$$\downarrow_c^i (n) = \begin{cases} n & n < c \\ n \dot{-} i & n \geq c \end{cases}$$

$$\downarrow_c^i (MN) = (\downarrow_c^i M)(\downarrow_c^i N)$$

$$\downarrow_c^i (\lambda M) = \lambda(\downarrow_{c+1}^i M)$$

```lean
def unshift (c i : Nat) : Lambda → Lambda
| var n =>
  if n < c then var n
  else var (n - i)
| app e₁ e₂ =>
  app (e₁.unshift c i) (e₂.unshift c i)
| abs e => abs (e.unshift (c + 1) i)
```

3.2 Shift y Unshift

Definición 3.2.2 (Unshift)

$$\downarrow_c^i (n) = \begin{cases} n & n < c \\ n \dot{-} i & n \geq c \end{cases}$$

$$\downarrow_c^i (MN) = (\downarrow_c^i M)(\downarrow_c^i N)$$

$$\downarrow_c^i (\lambda M) = \lambda(\downarrow_{c+1}^i M)$$

```lean
def unshift (c i : Nat) : Lambda → Lambda
| var n =>
  if n < c then var n
  else var (n - i)
| app e₁ e₂ =>
  app (e₁.unshift c i) (e₂.unshift c i)
| abs e => abs (e.unshift (c + 1) i)
```

⚠ Advertencia

Esta función es peligrosa, pues si el i es grande podemos terminar creando «variables ligadas» sin querer. $\downarrow_0^1 (\lambda 1) = \lambda \downarrow_1^1 1 = \lambda 0$

3.3 Sustitución

Definición 3.3.1 (Sustitución)

$$n[m := R] = \begin{cases} R & n = m \\ n & n \neq m \end{cases}$$

$$(MN)[m := R] = (M[m := R])(N[m := R])$$

$$(\lambda M)[m := R] = \lambda\big(M\big[m+1 :=\uparrow_0^1 R\big]\big)$$

3.3 Sustitución

Definición 3.3.1 (Sustitución)

$$n[m := R] = \begin{cases} R & n = m \\ n & n \neq m \end{cases}$$

$$(MN)[m := R] = (M[m := R])(N[m := R])$$

$$(\lambda M)[m := R] = \lambda(M[m+1 :=\uparrow_0^1 R])$$

Observación

Se hace $\uparrow_0^1 R$, porque queremos que al reemplazar en M, R no me genere nuevas variables ligadas, y se reemplazara por $m+1$, para evitar reemplazar variables ligadas.

3.3 Sustitución

```lean
def subst (v : Lambda) (n : Nat) (e : Lambda) :=
  match v with
  | var m =>
    if m = n then e
    else var m
  | app v₁ v₂ => app (v₁.subst n e) (v₂.subst n e)
  | abs v₁ => abs (v₁.subst (n + 1) (e.shift 0 1))
```

3.4 Beta reducción

Definición 3.4.1 (Beta en redex)

$$(\lambda M)N \to_\beta \downarrow_0^1 (M[0 := \uparrow_0^1 N])$$

```lean
def beta (M N : Lambda) :=
  (↓) 0 1 (M[0 ↦ (↑) 0 1 N])
```

3.4 Beta reducción

Definición 3.4.1 (Beta en redex)

$$(\lambda M)N \to_\beta \downarrow_0^1 (M[0 := \uparrow_0^1 N])$$

```lean
def beta (M N : Lambda) :=
  (↓) 0 1 (M[0 ↦ (↑) 0 1 N])
```

Observación

- Se hace $\uparrow_0^1 N$ para que las «variables libres» de N no se combinen con las de M.
- Al terminar de reemplazar, todos las «variables ligadas» están incrementados por 1, puesto que despareció un λ, es por eso que hacemos un unshift.

3.4 Beta reducción

Ejemplo.

$$(\lambda\lambda 01)1$$

3.4 Beta reducción

Ejemplo.

$$(\lambda\lambda 01)1 \to_\beta \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1]$$

3.4 Beta reducción

Ejemplo.

$$(\lambda\lambda 01)1 \to_\beta \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1]$$
$$= \downarrow_0^1 (\lambda 01)[0 := 2]$$

3.4 Beta reducción

Ejemplo.

$$
\begin{aligned}
(\lambda\lambda 01)1 \to_\beta \ & \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1] \\
= & \downarrow_0^1 (\lambda 01)[0 := 2] \\
= & \downarrow_0^1 \lambda((01)[0+1 := \uparrow_0^1 2])
\end{aligned}
$$

3.4 Beta reducción

Ejemplo.

$$
\begin{aligned}
(\lambda\lambda 01)1 \rightarrow_\beta \ & \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1] \\
= & \downarrow_0^1 (\lambda 01)[0 := 2] \\
= & \downarrow_0^1 \lambda((01)[0+1 := \uparrow_0^1 2]) \\
= & \downarrow_0^1 \lambda((01)[1 := 3])
\end{aligned}
$$

3.4 Beta reducción

Ejemplo.

$$
\begin{aligned}
(\lambda\lambda 01)1 \rightarrow_\beta \ & \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1] \\
= & \downarrow_0^1 (\lambda 01)[0 := 2] \\
= & \downarrow_0^1 \lambda((01)[0+1 := \uparrow_0^1 2]) \\
= & \downarrow_0^1 \lambda((01)[1 := 3]) \\
= & \downarrow_0^1 \lambda(0[1 := 3]1[1 := 3])
\end{aligned}
$$

3.4 Beta reducción

Ejemplo.

$$
\begin{aligned}
(\lambda\lambda 01)1 \rightarrow_\beta & \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1] \\
= & \downarrow_0^1 (\lambda 01)[0 := 2] \\
= & \downarrow_0^1 \lambda((01)[0+1 := \uparrow_0^1 2]) \\
= & \downarrow_0^1 \lambda((01)[1 := 3]) \\
= & \downarrow_0^1 \lambda(0[1 := 3]1[1 := 3]) \\
= & \downarrow_0^1 \lambda 03
\end{aligned}
$$

3.4 Beta reducción

Ejemplo.

$$
\begin{aligned}
(\lambda\lambda 01)1 \rightarrow_\beta \ & \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1] \\
= & \downarrow_0^1 (\lambda 01)[0 := 2] \\
= & \downarrow_0^1 \lambda((01)[0+1 := \uparrow_0^1 2]) \\
= & \downarrow_0^1 \lambda((01)[1 := 3]) \\
= & \downarrow_0^1 \lambda(0[1 := 3]1[1 := 3]) \\
= & \downarrow_0^1 \lambda 03 \\
= & \ \lambda(\downarrow_1^1 03)
\end{aligned}
$$

3.4 Beta reducción

Ejemplo.

$$
\begin{aligned}
(\lambda\lambda 01)1 \rightarrow_\beta \ & \downarrow_0^1 (\lambda 01)[0 := \uparrow_0^1 1] \\
= & \downarrow_0^1 (\lambda 01)[0 := 2] \\
= & \downarrow_0^1 \lambda((01)[0+1 := \uparrow_0^1 2]) \\
= & \downarrow_0^1 \lambda((01)[1 := 3]) \\
= & \downarrow_0^1 \lambda(0[1 := 3]1[1 := 3]) \\
= & \downarrow_0^1 \lambda 03 \\
= & \ \lambda(\downarrow_1^1 03) \\
= & \ \lambda 02
\end{aligned}
$$

3.4 Beta reducción

```lean
inductive Beta : Lambda → Lambda → Prop
  | basis (M N : Lambda) : Beta ((λ M).app N) (Lambda.beta M N)
  | appr (M N L : Lambda) : Beta N M → Beta (N.app L) (M.app L)
  | appl (M N L : Lambda): Beta N M → Beta (L.app N) (L.app M)
  | abs (N M) : Beta N M → Beta (λ N) (λ M)
```

```lean
inductive BetaTR : Lambda → Lambda → Prop
  | refl (M) : BetaTR M M
  | beta (M N) : M →β N → BetaTR M N
  | trans (M N L) : BetaTR M N → BetaTR N L → BetaTR M L
```

3.5 Rango

Definición 3.5.1 (Rango) El rango es la mayor variable en el término

$$\begin{aligned} \text{range } n &= n \\ \text{range } (MN) &= \max\{\text{range } M, \text{range } N\} \\ \text{range } (\lambda M) &= \text{range } M \end{aligned}$$

```lean
def range : Lambda → Nat
| var n => n
| app e₁ e₂ =>
  max (e₁.range) (e₂.range)
| abs e => e.range
```

3.5 Rango

Definición 3.5.1 (Rango) El rango es la mayor variable en el término

$$\begin{aligned} \text{range } n &= n \\ \text{range } (MN) &= \max\{\text{range } M, \text{range } N\} \\ \text{range } (\lambda M) &= \text{range } M \end{aligned}$$

```lean
def range : Lambda → Nat
| var n => n
| app e₁ e₂ =>
  max (e₁.range) (e₂.range)
| abs e => e.range
```

Observación

Esta función fue creada con el objetivo de simular en algunas pruebas la hipótesis de que $x \notin \text{FV}(M)$, ya que resulta muy conveniente.

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela 🟢

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela 🟢
- Demostrar que la beta reducción paralela es confluente

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela 🟢
- Demostrar que la beta reducción paralela es confluente 🟠

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela 🟢
- Demostrar que la beta reducción paralela es confluente 🟠
- Demostrar el lema de sustitución

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela 🟢
- Demostrar que la beta reducción paralela es confluente 🟠
- Demostrar el lema de sustitución 🟢

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela 🟢
- Demostrar que la beta reducción paralela es confluente 🟠
- Demostrar el lema de sustitución 🟢
- Demostrar el teorema de Church-Rosser

3.6 Objetivos y cosas por hacer

- Definir los términos lambda y sus operaciones 🟢
- Definir una notación en el código, para hacer la lectura más sencilla 🟢
- Definir la beta reducción paralela 🟢
- Demostrar que la beta reducción paralela es confluente 🟠
- Demostrar el lema de sustitución 🟢
- Demostrar el teorema de Church-Rosser 🔴

3.7 Repositorio

Figura 2: https://github.com/Danelnov/Lambda-Calculus-Formalization-

Bibliografía

Bibliografía

[1] M. Takahashi, «Parallel Reductions in λ-Calculus», *Information and Computation*, vol. 118, n.º 1, pp. 120–127, 1995, doi: https://doi.org/10.1006/inco.1995.1057.

[2] N. de Bruijn, «Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem», *Indagationes Mathematicae (Proceedings)*, vol. 75, n.º 5, pp. 381–392, 1972, doi: https://doi.org/10.1016/1385-7258(72)90034-0.

[3] A. Sampson, «Lecture #15: De Bruijn, Combinators, Encodings». [En línea]. Disponible en: https://www.cs.cornell.edu/courses/cs4110/2018fa/lectures/lecture15.pdf