Tesis de la Licenciatura en Ciencias de la computación

# El cálculo lambda y fundamentos de la computación

2016


Universidad de Sonora
División de Ciencias Exactas y Naturales
Departamento de Matemáticas

# El cálculo lambda y fundamentos de la computación

Tesis que para obtener el título de

Licenciado en Ciencias de la Computación

presenta

**Eduardo Acuña Yeomans**

Director de tesis: Martín Eduardo Frías Armenta.

Hermosillo, Sonora, 15 de marzo de 2016.

# Resumen

Falta escribir el resumen.

*Falta escribir la dedicatoria.*

# Agradecimientos

Falta escribir los agradecimientos.

# Índice general

**Introducción** **1**

Descripción del trabajo . . . . . 1

Aportaciones principales . . . . . 1

Organización del trabajo . . . . . 1

**1. El cálculo lambda en el desarrollo de la computación** **3**

1.1. Introducción . . . . . 3

1.2. Revisión histórica . . . . . 4

1.3. Enfoques del cálculo lambda . . . . . 7

1.3.1. Fundamentos de matemáticas . . . . . 7

1.3.2. Cómputo . . . . . 8

1.3.3. Cálculo lambda puro . . . . . 8

**2. El cálculo lambda sin tipos** **9**

2.1. Introducción . . . . . 9

2.2. Notación . . . . . 10

2.2.1. Variables . . . . . 11

2.2.2. Aplicaciones anidadas . . . . . 12

2.2.3. Aplicaciones sin sentido aparente . . . . . 12

2.2.4. Aplicación de función a función . . . . . 12

2.2.5. Funciones de varias variables . . . . . 12

2.3. Equivalencias y operaciones . . . . . 13

2.3.1. Sustitución . . . . . 13

2.3.2. Equivalencias . . . . . 13
2.3.3. Transformación de expresiones . . . . . 18
2.4. Formalización de la teoría $\boldsymbol{\lambda}$ . . . . . 18

**3. Tipos en el cálculo lambda** **19**

**4. Teoría de la computación** **21**

**5. Lógica constructivista** **23**

**6. Lenguajes de programación** **25**

**Conclusiones** **27**

**A. Composición tipográfica de términos lambda en LaTeX** **29**
A.1. Código . . . . . 29
A.2. Documentación . . . . . 30

**B. Compilación del cálculo lambda** **35**

**C. Intérprete del cálculo lambda** **37**

**Bibliografía** **48**

# Introducción

## Descripción del trabajo

## Aportaciones principales

## Organización del trabajo

Capítulo 1

# El cálculo lambda en el desarrollo de la computación

## 1.1. Introducción

El cálculo lambda es un sistema formal inventado en la década de 1920 por el matemático y lógico Alonzo Church con la finalidad de describir las propiedades mas básicas de la abstracción, aplicación y sustitución de funciones en un contexto muy general [2].

Cardone y Hindley mencionan en su artículo publicado en "Handbook of the History of Logic" que el cálculo lambda es utilizado ampliamente en el cómputo y la lógica de orden superior. Versiones modificadas del cálculo lambda fundamentan importantes sistemas lógicos y lenguajes de programación. Ejemplos de esto se puede encontrar en el desarrollo de los lenguajes Lisp [6], Scheme [12] y Haskell [8]; así como en demostradores automáticos de teoremas como LCF [7].

El concepto de *función* en el cálculo lambda es tratado como *reglas* en lugar de como gráficas, la noción de funciones como reglas se refiere, en este sistema formal, al proceso de obtener un valor a partir de un argumento, sin imponer restricciones sobre éste. En la época en la que Church formuló el cálculo lambda, sistemas formales con la noción de función como la teoría de conjuntos Zermelo-Fraenkel impedían que algunas funciones pudieran ser definidas [1], un ejemplo de esto es la función *identidad*, es usual restringir el dominio de esta función dependiendo de el contexto en la que es utilizada, sin embargo, en principio pudiera tener como dominio el conjunto universal, lo cual no es permitido en la teoría de conjuntos tradicional ya que dicho conjunto sería elemento de si mismo.

## 1.2. Revisión histórica

La siguiente revisión histórica resume el desarrollo del cálculo lambda de acuerdo a una compilación de artículos sobre la historia de la lógica matemática [2].

El sistema formal conocido hoy en día como el *cálculo lambda* fue inventado por el matemático y lógico Alonzo Church en 1928, sin embargo el primer trabajo publicado en donde se hace mención de este sistema fue en el artículo titulado *A set of postulates for the foundation of logic* [3] en 1932, en donde presentó el desarrollo de un sistema lógico a partir del cálculo lambda.

De 1931 a 1934 en la universidad de Princeton, Alonzo Church recibió ayuda de dos estudiantes: Stephen Kleene y Barkley Rosser. Entre ellos colaboraron e hicieron varias aportaciones, tanto en la lógica de Church como en el sistema detrás de ella, el cual era el cálculo lambda. Uno de estos descubrimientos fue la inconsistencia de la lógica propuesta por Church en su trabajo de 1932.

En años siguientes se desarrolló el estudio de la teoría de reducción en el cálculo lambda, lo cual llevó a la demostración del teorema de confluencia de Church-Rosser en 1936 y aseguró la consistencia del cálculo lambda puro, es decir, el sistema simple del cálculo lambda no contenía inconsistencias. En este año también se encontró la equivalencia del sistema con la *lógica combinatoria*, un sistema inventado por Haskell Curry basado en ideas similares a las de Church.

Trabajos sobre la representación de los números naturales en el cálculo lambda se desarrollaron y se descubrió que las funciones del sistema que operaban sobre estas representaciones eran mas poderosas que lo anticipado, debido a una demostración de equivalencia entre esta clase de funciones, las funciones recursivas de Herbrand-Gödel y las funciones computables de Turing las cuales intentaban formalizar el concepto informal de cómputo. Alonzo Church conjeturó en la tesis de Church-Turing que las funciones definibles en el cálculo lambda capturaban exactamente el concepto informal de cómputo. El clímax de esta serie de trabajos sobre lo que es posible definir en el cálculo lambda fue cuando Church dio una solución negativa al problema de decisión planteado por Hilbert para la lógica de primer orden.

Por muchos años el cálculo lambda falló en atraer a otros estudiosos lógicos y a pesar de haber servido para la primer demostración para el problema de decisión, Turing poco tiempo después consiguió desarrollar otra demostración utilizando un modelo mas transparente al cómputo, incluso fuertes personajes que contribuyeron al desarrollo del cálculo

lambda como Kleene parecían favorecer otros sistemas. Sin embargo Alonzo Church escribió un libro introductorio titulado *Los cálculos de la conversión lambda* dirigido a una demografía menos especializada lo cual amplió el estudio del sistema a otras ramas de las matemáticas.

Durante la década de los cuarentas y cincuentas, los avances relacionados con el cálculo lambda se enfocaron en cuatro temas: teoría de tipos simples (simple type theory), teoría abstracta de reducción (abstract reduction theory), reducciones en la lógica combinatoria y en el cálculo lambda y sistemas ilativos (illative systems). En su extensa revisión de la historia del cálculo lambda, Cardone y Hindley expresan que los avances de estas cuatro vertientes de estudio fueron fundamentales para el desarrollo del cálculo lambda en las siguientes décadas.

En los primeros sistemas basados en el cálculo lambda la aplicación de funciones no tenía restricciones, una función podía ser evaluada en si misma o en cualquier otra expresión válida del cálculo lambda. Versiones del cálculo lambda en donde se restringe la aplicación de funciones utilizando tipos son llamados cálculo lambda con tipos. Estos tipos son incrustados incrustados en la definición de funciones como anotaciones para restringir el rango de dominio.

Basándose en las ideas de Frank Ramsey y Leon Chwistek, Church desarrolló una teoría de tipos basada en funciones para simplificar la teoría de tipos de Russel y Whitehead. Su sistema de tipos fue analizada y extendida por una serie de estudiantes de doctorado de Princeton, cuyas contribuciones mas relevantes fueron dar definiciones de estructuras algebraicas para el cálculo lambda con tipos y la extensión del sistema para tipos transfinitos.

Alonzo Church no se involucró mucho en las teorías abstractas de reducción, sin embargo, fuertes contribuciones fueron realizadas en la década de los sesenta. El concepto central detrás de estas teorías fue el de la propiedad de confluencia asociada a ciertos procedimientos para la manipulación de expresiones en el cálculo lambda. Con el pasar de los años el interés por estas teorías abstractas siguió desarrollándose y llegó a extenderse mas allá de los sistemas formales del cálculo lambda y la lógica combinatoria.

Los primeros trabajos de Kleene y Rosser con el cálculo lambda involucraban un procedimiento de manipulación de expresiones llamado $\beta$-reducción. Haskell Curry estudió mas a fondo esta reducción, sin embargo su trabajo (como al igual que muchos científicos de la época) se vio interrumpido por la segunda guerra mundial. Fue hasta después de 1950

que Curry retomó esta línea de investigación y por 20 años trabajó tanto con el cálculo lambda como con la lógica combinatoria vaciando una gran cantidad de resultados y teoremas en dos monografías que se convirtieron en referencias bibliográficas clave para futuros matemáticos.

A partir de la década de los sesenta, el interés por el cálculo lambda llegó a varios científicos trabajando en ciencias de la computación, en particular en el área de lenguajes de programación.

Hubo tres figuras clave en el desarrollo del cálculo lambda en el área de la programación: John McCarthy con el desarrollo del lenguaje de programación LISP cuya notación de abstracción tiene muchas similitudes con la del cálculo lambda; Peter Landin el cual propuso varias construcciones para el lenguaje ALGOL utilizando expresiones del cálculo lambda y subsecuentemente desarrollando la máquina abstracta SECD la cual trataba expresiones del cálculo lambda como programas de computadora y finalmente Corrado Böhm quien desarrolló el lenguaje CUCH en donde se utilizaban los combinadores de Haskell Curry y el cálculo lambda.

A pesar de que los trabajos de McCarthy y Landin llegaron a ser mas populares en la comunidad de programación. El trabajo que realizó Böhm y los estudiantes que formó fue el que contribuyó mas al cálculo lambda desde una perspectiva de ciencias de la computación. En sus años de estudio del sistema puro del cálculo lambda, ellos plantearon y resolvieron varios problemas asociados a la sintaxis del cálculo lambda y la lógica combinatoria.

El cálculo lambda ha sido reciclado por varias generaciones convirtiéndose en tema de interés para personas estudiando otras ramas de las matemáticas. A pesar de que estos trabajos se han alejado de los objetivos originales de Church, las nociones básicas desarrolladas por el trabajo previo se han convertido en pilares fundamentales de las ciencias de la computación.

El ganador del premio Turing de 1976 Dana Scott menciona en una presentación de la celebración del centenario de Alan Turing [11] algunos trabajos que han sido relevantes para el cálculo lambda en este milenio, incluyo aquí algunos títulos de artículos que a pesar de tratar temas mas avanzados que los presentados en este trabajo, pueden proveer una perspectiva moderna del estudio del cálculo lambda.

- Moerdijk-Palmgren (2000) Predicative topos.
- Ehrhard, Regnier (2003) Differential $\lambda$-calculus.

- Mosses (2004) Modular Structural Operational Semantics.
- Taylor (2005) A $\lambda$-calculus for real analysis.
- Awodey-Warren (2006) Homotopy type theory.
- Hudak, etal. (2010) Functional Reactive Programming.

## 1.3. Enfoques del cálculo lambda

De acuerdo a Barendregt [1] el estudio del cálculo lambda tiene tres enfoques:

I Fundamentos de matemáticas.

II Cómputo.

III Cálculo lambda puro.

### 1.3.1. Fundamentos de matemáticas

Los objetivos que se tenían cuando se dio origen al cálculo lambda eran desarrollar una teoría general de funciones y extender esta teoría con nociones lógicas para proveer fundamentos a la lógica y a partes de las matemáticas [1].

El primer objetivo es expresado por Church en [4], en donde dice

> El estudio de las propiedades generales de funciones, independientemente de su aparición en cualquier dominio matemático, pertenece a la lógica formal o se encuentra en la línea divisoria entre la lógica y la matemática. Este estudio es la motivación original para los cálculos — pero estos están formulados de tal manera que es posible abstraer a partir del significado pretendido y considerarlos simplemente como sistemas formales.

Desde la publicación del artículo [3] se intentó lograr el segundo objetivo, sin embargo, todos los intentos de proveer un fundamento para las matemáticas fallaron. Casi inmediatamente después de la publicación de Church se encontró una contradicción. Kleene y Rosser, dos estudiantes de Church, formularon una variante de la paradoja de Richard para el sistema lógico planteado [10].

Después del descubrimiento de la paradoja Kleene-Rosser Church se sintió desalentado en el estudio de los fundamentos de las matemáticas. En 1941 Church publicó una teoría que corregía las inconsistencias observadas en [10], ésta fue llamada cálculo-$\lambda I$ y era menos ambiciosa y mas limitada que sus anteriores sistemas. Otros matemáticos y lógicos continúan utilizando variaciones del cálculo lambda en sus investigaciones sobre los fundamentos de las matemáticas.

### 1.3.2. Cómputo

La parte del cálculo lambda que solo trata con funciones resultó ser bastante exitosa en el estudio del cómputo. Utilizando esta teoría, Church propuso una formalización de la noción "efectivamente computable" la cual es llamada *lambda definibilidad*. Esta formalización resultó ser equivalente a el concepto de *computabilidad* propuesto por Turing.

Debido a que el cálculo lambda fue la inspiración de muchos lenguajes de programación, investigación en la semántica de los lenguajes se realizó utilizando este sistema. Peter Landin en la década de 1960 realizó una traducción del lenguaje Algol al cálculo lambda, lo cuál le permitió expresar de manera formal la semántica operacional del lenguaje, es decir, cómo debía ejecutarse un programa escrito en Algol.

En 1969 Dana Scott logró expresar la semántica denotacional de los lenguajes de programación utilizando el cálculo lambda, esto permitió poder expresar de manera formal el significado de los programas escritos en un lenguaje de programación.

### 1.3.3. Cálculo lambda puro

Los anteriores dos enfoques del cálculo lambda se desarrollaron extendiendo la teoría básica con sistemas lógicos y computacionales para utilizar al cálculo lambda como herramienta o como un lenguaje básico. Otro enfoque es que el objeto de estudio sea el sistema en sí.

El estudio del cálculo lambda en sí consiste principalmente en estudiar las funciones como reglas partiendo de la identificación de las fórmulas bien formadas o *términos* que pueden ser expresados en el sistema. Las investigaciones en el estudio del cálculo lambda permiten resolver problemas como la identificación de funciones sobre términos que se pueden definir como términos en el cálculo lambda o mecanismos para poder manipular un término para obtener otro y la relación entre ellos.

Capítulo 2

# El cálculo lambda sin tipos

## 2.1. Introducción

El cálculo lambda es un sistema matemático el cuál permite expresar, manipular y estudiar funciones. La manera en como se trabaja con funciones en este sistema es un poco diferente a como es usual en otras áreas de las matemáticas.

En general se considera que una función es una regla de correspondencia que asocia una fórmula bien formada del cálculo lambda a otra. El procedimiento para expresar y aplicar funciones es mecánico, lo cuál permite concebir a las funciones como programas que tienen una entrada y una salida que puede ser descrita en términos de la entrada.

Una función no está limitada a operar con argumentos arbitrarios, para una función en particular se pueden identificar clases o rangos de posibles argumentos y valores de la función. No es necesario especificar la clase de argumentos y la clase de valores de una función ya que podemos tratar la aplicación de una función a algo fuera de la clase de argumentos asociada a ella como un valor sin significado.

Por ejemplo, la función $f(x) = 3 \times x$ tiene como rango de argumentos y rango de valores a números. Al aplicar la función $f$ a un número en particular como el 2 se obtiene otro número, en este caso $f(2) = 3 \times 2 = 6$. Los valores de $f$ pudieran o no perder sentido si se aplica $f$ a otra clase de argumentos: En el caso de considerar las palabras en español como argumentos de $f$, no tienen sentido los valores que puede tomar $f$ debido a que no hay una definición precisa de multiplicación que involucre a palabras en el español y números, por otro lado pudiéramos ampliar la clase de argumentos de $f$ a que incluya matrices, debido a que si hay una operación definida de multiplicación entre un número

(en este caso el 3) y una matriz.

Es posible considerar funciones cuya clase de argumentos sean funciones, por ejemplo en el cálculo diferencial la derivada es una función cuyas clases de argumentos y valores son a su vez funciones. En general existen funciones que siempre tienen sentido sin importar en que argumentos sea aplicada, una de estas funciones es la función identidad $I$ la cual es definida como $I(x) = x$.

## 2.2. Notación

La notación utilizada previamente para escribir la definición y aplicación de funciones en los párrafos de arriba es ampliamente usada en matemáticas, sin embargo el cálculo lambda favorece una notación alternativa.

Retomando un ejemplo previo, la expresión para la definición de la función $f$ es $f(x) = 3 \times x$ y se compone de tres elementos: primero se encuentra el nombre de la función seguido de una letra que representa una variable, el símbolo = establece una relación del nombre de la función y el argumento con otra expresión en donde puede o no aparecer la variable $x$, en este caso esta otra expresión es $3 \times x$ la cual representa una multiplicación. En el cálculo lambda no se especifica el nombre de la función, considerando una definición como anónima. La manera en como se escribiría la definición de $f$, considerando que $m$ es la operación de multiplicación y $y$ es el número 3 es: $(\lambda x.\,((m\ x)\ \ y))$, el nombre de la función es omitido, el argumento se encuentra después de el símbolo $\lambda$ y lo que antes era el lado derecho de la función se encuentra después del punto.

Debido a que en el cálculo lambda el nombre de la función no se encuentra en su definición la aplicación de una función se escribe especificando textualmente la función a la que se hace referencia. La manera en como de denota la aplicación de $f$ en el número 2 es:

$$((\lambda x.\,((m\ x)\ \ y))\ \ z) \text{ en lugar de } f(2)$$

Considerando que $z$ es el número 2.

A pesar de aparentar ser una notación mas inconveniente debido a que se requiere escribir mas texto para expresar la misma idea, esta notación nos permite ser mas explícitos en las operaciones que se realizan con las funciones y proveen de una mayor uniformidad en la sintáxis del lenguaje utilizado. Un inconveniente de utilizar la notación mas popular

es que la notación es mas basta que la mostrada en el ejemplo, en el caso que la función se defina por casos se introduce nueva sintaxis, por ejemplo la definición de una función que compute el factorial de un número entero no negativo se escribiría:

$$g(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \times g(n-1) & \text{en otro caso} \end{cases}$$

Si involucramos operaciones mas complejas que la multiplicación nos encontraremos con aún mas sintaxis, por ejemplo $\Sigma$ para una sumatoria, $\Pi$ para productos, $\frac{df}{dx}$ para la derivadas, $\int$ para integrales e incluso la función $g$ definida previamente sería escrita como $n!$.

En el cálculo lambda, toda la notación se compone de definición de funciones, aplicación de funciones y variables. Es por eso que para representar la función $f$ en el cálculo lambda tuvimos que escribir a la operación de multiplicación como una función y a los números como variables. Como se verá mas adelánte, es posible definir con precisión los números naturales y las operaciones aritméticas en el cálculo lambda.

Cabe mencionar que es usual escribir términos lambda precisando únicamente el significado operacional de sus partes. Por ejemplo:

| | | |
|---|---|---|
| a. | $x$ | Es una variable |
| b. | $(\lambda x.\,(x\ \ (x\ y)))$ | Es una función |
| c. | $(y\ \ (\lambda x.\,(\lambda x.x)))$ | Es una aplicación |
| d. | $((\lambda y.y)\ \ (\lambda x.\,(x\ w)))$ | Es una aplicación de una función en otra |
| e. | $(\lambda f.\,(\lambda x.\,(f\ \ (f\ x))))$ | Es una función con rango de valores de funciones |

### 2.2.1. Variables

En el cálculo lambda las variables por si solas son expresiones válidas, en el inciso $a$ aparece la variable $x$ la cual no es ni una definición de función ni una aplicación de una expresión a una función.

### 2.2.2. Aplicaciones anidadas

Así como en la notación tradicional se pueden escribir expresiones como $\text{sen}(\ln(x))$ en el cálculo lambda también se puede anidar la aplicación de funciones. En el inciso $b$ se muestra un ejemplo de esto, $(\lambda x.\,(x\ \ (x\ y)))$ consiste en aplicar $x$ al resultado de la aplicación $(x\ y)$.

### 2.2.3. Aplicaciones sin sentido aparente

El inciso $c$ muestra una expresión que no tiene sentido si consideramos la aplicación como se abordó previamente. Sin embargo, $(y\ \ (\lambda x.\,(\lambda x.x)))$ es una expresión válida en el cálculo lambda a pesar de ser poco usual tratar la aplicación de una expresión en una variable tanto en la teoría como en las aplicaciones del sistema.

### 2.2.4. Aplicación de función a función

Como se describió previamente, el cálculo lambda permite expresar la aplicación de una función a otra función para definir operaciones interesantes que manipulen expresiones (como lo son la diferenciación y la integración). En el inciso $d$ se muestra como se aplica la función identidad $(\lambda y.y)$ a la función $(\lambda x.\,(x\ w))$ (esta segunda función no tiene un significado en particular).

### 2.2.5. Funciones de varias variables

En la notación tradicional se denotan las funciones de varias variables como $f(x_1,\ldots,x_n) = M$ (donde $M$ es una expresión) y cuando se aplican estas funciones a valores particulares se escribe $f(v_1,\ldots,v_n)$ (donde $v_1,\ldots,v_n$ son valores en el dominio de $f$). En el cálculo lambda solo se pueden definir funciones en una sola variable, sin embargo, como el sistema nos permite tener funciones con rango de valores de funciones, se puede simular tener funciones de varias variables. Si consideramos como ejemplo la función $f(x,y) = \sin(x) + \cos(y)$ su equivalente en el cálculo lambda sería $(\lambda x.\,(\lambda y.\,((a\ \ (s\ x))\ \ (c\ y))))$, donde $a$, $s$ y $c$ son las operaciones de adición, seno y coseno. De tal manera que si se desee evaluar $f(\pi, \frac{1}{2}\pi)$, en el cálculo lambda se escribiría $(((\lambda x.\,(\lambda y.\,((a\ \ (s\ x))\ \ (c\ y))))\ \ p1)\ \ p2)$ (donde $p1$ y $p2$ corresponden a los valores de $\pi$ y $\frac{1}{2}\pi$).

En el cálculo lambda, la evaluación parcial de una función como esta es muy útil ya que permite tratar con los argumentos de la función de manera independiente.

## 2.3. Equivalencias y operaciones

Hay una serie de operaciones elementales en el cálculo lambda que permiten la manipulación de expresiones. La operación mas básica y de la cual dependen el resto es la de *sustitución* de una variable en una expresión por otra expresión. Las otras tres operaciones tratan con el concepto de igualdad entre dos expresiones y como se puede obtener una expresión equivalente a partir de otra.

### 2.3.1. Sustitución

La operación de sustitución se encarga de cambiar la manera en como se escribe una expresión, usualmente se sustituyen variables por expresiones mas complejas. Para denotar la sustitución de la variable $y$ por $x$ en la expresión $(y\ w)$ se escribe:

$$(y\ w)\,[y := x] = (x\ y)$$

En general se puede sustituír cualquier variable dentro de una expresión por otra expresión, de tal manera que si $M$ y $N$ son expresiones del cálculo lambda y $x$ es una variable, el sustituír a la $x$ en $M$ por $N$ se denota $M[x := N]$.

Otros ejemplos de sustitución son

§ $x[x := (\lambda y.y)] = (\lambda y.y)$

§ $(\lambda f.\,(f\ \ (f\ x)))\,[x := (f\ x)] = (\lambda f.\,(f\ \ (f\ \ (f\ x))))$

### 2.3.2. Equivalencias

Como se abordó previamente se pueden construír expresiones en el cálculo lambda cuyo significado no esté definido con claridad o que el significado de una expresión dependa del contexto en el que es utilizada.

Una pregunta que se puede responder independientemente del contexto en el que se trabaje con el cálculo lambda es, si $M$ y $N$ son dos expresiones del cálculo lambda, ¿Son

equivalentes?. Pera responder esta pregunta se tiene que explorar a que nos podemos referir con *equivalente*.

Podemos afirmar que dos expresiones son equivalentes cuando entendemos el contexto y el nivel de abstracción en el que se está refiriendo a ellas. Por ejemplo, podemos considerar dos números equivalentes si se escriben igual, en cuyo caso 3, $\frac{6}{2}$ y 3.0 no serían equivalentes. Sin embargo si consideramos una equivalencia de valores, tanto 3, como $\frac{6}{2}$, como 3.0 hacen referencia al mísmo elemento en $\mathbb{R}$. Quizá pueda parecer demasiado detallista precisar a que tipo de equivalencia se está refiriendo cada vez que se comparan dos expresiones, sin embargo, si consideramos a los algorítmos como los objetos que comparamos, se suele tomar en cuenta la cantidad de operaciones que realizan en función al tamaño del las entradas, por lo que dos algoritmos con las mismas entradas y las mismas salidas no sean equivalentes desde un punto de vista de complejidad computacional.

Ya que es importante precisar con exactitud el concepto de equivalencia, abordaremos algunas maneras en las que podemos comparar dos expresiones del cálculo lambda.

### Equivalencia sintáctica

Esta equivalencia es aquella que nos permite distinguir si dos expresiones son lo mismo de acuerdo a la manera en como están escritas. Si $M$ y $N$ están escritas de la misma manera, símbolo por símbolo, se dice que son equivalentes sintácticamente y se denota $M \equiv N$.

### Equivalencia estructural

Podemos ver las expresiones del cálculo lambda como definición de funciones, aplicación de funciones, variables y combinaciones de estas. La posición en donde se encuentran las partes de una expresión determinan su estructura, sin embargo, los nombres de varaibles que se utilizan no influyen su significado estructural, de manera similar al concepto de la equivalencia en gráficas isomorfas.

Consideranndo la función identidad $(\lambda x.x)$ se puede observar que tiene la misma estructura que $(\lambda y.y)$ la cual también representa la función identidad. A pesar de no estar escritas exactamente igual, la correspondencia que hay de la posición de la variable $x$ en la primera expresión con la posición de la variable $y$ en la segunda y el hecho de que ambas tienen la misma estructura nos permite decir que son equivalente.

Considerando dos expresiones un poco mas complejas como $(\lambda f.\,(\lambda x.\,(f\ x)))$ y $(\lambda g.\,(\lambda y.\,(g\ y)))$ podemos notar que también son equivalentes en este sentido. Una manera intuitiva de identificar esta equivalencia es comprobar que las operaciones que representan dos expresiones son leídas de la misma manera pero en lugar de leer explícitamente el nombre de la variable, se lee la posición de la primera aparición de dicha variable. En el ejemplo dado se lee "Una función cuyo cuerpo es una función cuyo cuerpo es la aplicación de la primer variable en la segunda". Si dos expresiones $M$ y $N$ son estructuralmente equivalentes, se dice que $M$ es $\alpha$-congruente a $N$.

Una notación utilizada para corroborar la equivalencia estructural es el *índice de De Bruijn*, esta notación evita la aparición de variables en las expresiones y en su lugar utiliza números que representan la distancia de una variable a la $\lambda$ de la función en donde aparece como argumento. De tal manera que una expresión como

$$(\lambda z.\,((\lambda y.\,(y\ \ (\lambda x.x)))\ \ (\lambda x.\,(z\ x)))) \tag{2.1}$$

se escribe usando el índice de De Bruijn como

$$\lambda(\lambda 1(\lambda 1))(\lambda 21) \tag{2.2}$$

En la figura 2.1 se puede observar de manera gráfica la transformación de una notación a otra para este ejemplo en particular.

Una desventaja de utilizar la notación de De Bruijn es que ciertas expresiones del cálculo lambda no pueden ser escritas, en particular, toda variable tiene que estar asociada a una $\lambda$ para que esta notación pueda ser utilizada. Sin embargo como veremos más adelante, la mayoría de los usos del cálculo lambda asocian a todas las variables en las expresiones.

En este trabajo no se utilizará la notación de De Bruijn, sin embargo es importante mencionarla ya que dos expresiones que sean $\alpha$-congruente van a ser sintácticamente equivalentes utilizando los índices de De Bruijn.

### Equivalencia de aplicación

Otra equivalencia que podemos encontrar en las expresiones es la de aplicación, esta hace referencia a que la aplicación de una función a una expresión es equivalente al resultado de evaluar la función con dicha expresión como argumento. Para entender mejor

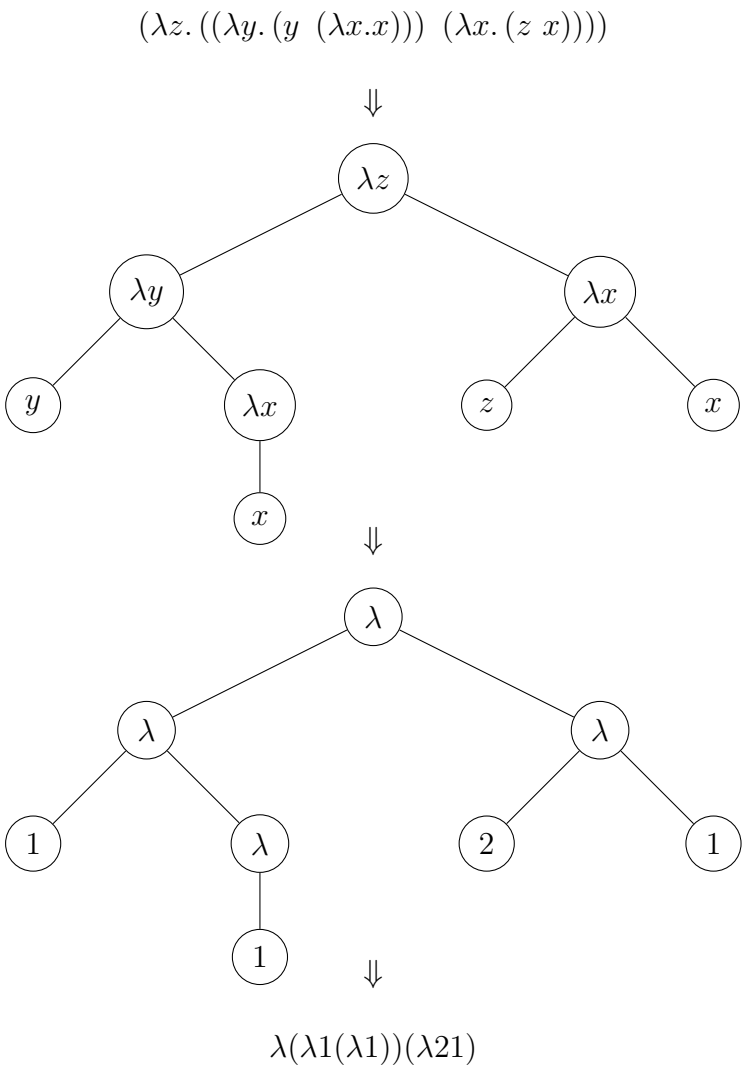


Figura 2.1: Transformación de (2.1) a (2.2).

este concepto, consideramos la función en notación tradicional $f(x) = x^2$, si se evalúa $f(3)$ el resultado es 8, por lo tanto podemos decir que $f(3)$ y 8 son equivalentes.

Si consideramos la expresión de la función identidad $(\lambda x.x)$ podemos afirmar que para cualquier expresión $M$, $((\lambda x.x)\ \ M)$ es equivalente a $M$, en el cálculo lambda a esta equivalencia se le llama *$\beta$-convertibilidad*.

En la notación tradicional, estas tres equivalencias se denotan con el mísmo símbolo $=$, de tal manera que si dos expresiones son equivalentes ya sea sintácticamente, estructuralmente o aplicativamente, entonces serán consideradas iguales. En el cálculo lambda es importante diferenciar estas equivalencias ya que el manejo de las funciones no se aborda

desde el punto de vista de una relación entre el dominio y codominio, si no como una expresión que puede ser manipulada y transformada de manera mecáncia.

### Equivalencia de redundancia

Otro tipo de equivalencia es la de redundancia, se considera en un caso muy particular pero más adelante se verá que las implicaciones de considerarla son fuertes. Consideremos la expresión $(\lambda x.\,((\lambda y.y)\ x))$, el papel que puede jugar es el de ser aplicada en otra expresión $M$, la cual resultaría igual a aplicar la expresión interna $(\lambda y.y)$ en $M$. Por las equivalencias descritas previamente podemos observar que $(\lambda x.\,((\lambda y.y)\ x))$ y $(\lambda y.y)$ no son sintácticamente equivalentes, ni estructuralmente equivalentes, ni siquiera aplicativamente equivalentes. El trabajar con la función que envuelve a $(\lambda y.y)$ resulta redundante al momento de aplicar las funciones en expresiones, ésto nos permite considerar un criterio de equivalencia.

En el cálculo lambda, la equivalencia de redundancia se denomina *eta-equivalencia* y nos permite considerar como iguales las expresiones de la forma $(\lambda x.\,(M\ x))$ y $M$.

### Equivalencia computacional

En el estudio de la lógica, se hace la distinción que una equivalencia puede ser extensional o intensional. La equivalencia extensional hace referencia a las propiedades externas de los objetos, mientras que la equivalencia intensional hace referencia a la definición o representación interna de los objetos.

Las equivalencias sintáctica y estructural son equivalencias intencionales, mientras que las equivalencias de aplicación y redundancia son equivalencias extensionales, debido a que se jusgan dos objetos a partir de su evaluación. Sin embargo, la equivalencia de aplicación no comprende el caso mencionado al inicio de esta subsección. Suponiendo que tenemos dos expresiones $M$ y $N$ que describen el mismo algoritmo o la misma función, la equivalencia de aplicación no los considera equivalente.

En la notación tradicional, la igualdad de funciones es una equivalencia extensional, por ejemplo $f(x) = e^{i\pi} \times x$ y $g(x) = x$ describen la función identidad y podemos aseverar que $f = g$ sin necesidad de evaluar ambas funciones con un argumento en particular.

En el cálculo lambda se puede hablar de este tipo de igualdad funcional si consideramos que para toda expresión del cálculo lambda $P$, si $(M\ P)$ es equivalente a $(N\ P)$, entonces las expresiones $M$ y $N$ son extensionalmente equivalentes.

### 2.3.3. Transformación de expresiones

A cada equivalencia diferente a la sintáctica se le puede asociar una operación de transformación la cual nos permita pasar de una expresión $M$ a otra expresión $N$ de tal manera que una equivalencia entre estas dos expresiones se cumpla.

En el caso de la $\alpha$-congruente la operación correspondiente consiste en cambiar nombres de variables, en la $\beta$-*convertibilidad* la operación consiste en realizar una secuencia de sustituciones de las variables de una función por expresiones a las que la función es aplicada y en la *eta-equivalencia* la operación consiste en la eliminación de funciones redundantes.

Estas operaciones se definirán de manera formal más adelante y aunque puedan parecer operaciones sencillas de definir a partir de la operación de sustitución, se tiene que tener mucho cuidado en no obtener expresiones que rompan la equivalencia asociada.

## 2.4. Formalización de la teoría $\boldsymbol{\lambda}$

Capítulo 3

# Tipos en el cálculo lambda

Formulación e introducción a aplicaciones.

Capítulo 4

# Teoría de la computación

Cálculo lambda en la teoría de la computación, computabilidad.

Capítulo 5

# Lógica constructivista

Cálculo lambda en lógica, demostradores automáticos de teoremoas, isomorfismo de Curry-Howard.

Capítulo 6

# Lenguajes de programación

Cálculo lambda y lenguajes de programación, semántica operacional, ordenes de evaluación, máquinas abstractas.

# Conclusiones

Apéndice A

# Composición tipográfica de términos lambda en LaTeX

Para escribir términos lambda en LaTeX se escribió el paquete TeX-$\mathcal{LAMBDA}$ , se encarga normalizar términos lambda, aplicar el abuso de notación utilizado en el trabajo y agregarle estilos a diferentes partes de un término lambda de forma automática con un simple comando en el modo matemáticas. También contiene diversos comandos para abreviar la escritura de terminología y operaciones del cálculo lambda.

## A.1. Código

El siguiente archivo es nombrado `texlambda.sty` y contiene las definiciones del paquete de LaTeX.

El siguiente archivo es nombrado `texlambda.rkt` y contiene el parser de términos lambda, el algoritmo de normalización, desnormalización y los procedimientos para vaciar en la salida estándar código de LaTeX.

# A.2. Documentación

## Introducción

Este paquete permite escribir en el modo de matemáticas de LaTeX términos bien formados del cálculo lambda sin tipos.

Los comandos principales son `\lc{x}` y `\lc*{x}`. El primero traduce el término lambda `x` en código para el modo matemáticas de LaTeX sin abusos de notación, el segundo traduce a `x` de la manera mas concisa posible, utilizando los abusos de notación estándar en la literatura del cálculo lambda. En ambos casos, `x` puede ser escrito con o sin abuso de notación o una mescolanza de ambos estilos.

La sintáxis aceptada es:

§ Si `x` es una secuencia de caracteres sin espacios, entonces `x` es un término lambda aceptado (*átomo*).

§ Si `x` y `y` son términos lambda, entonces `(x y)` es un término lambda aceptado (*aplicación*).

§ Si `x` es una secuencia de caracteres sin espacios y `y` es un término lambda aceptado, entonces `(\x.y)` es un término lambda aceptado (*abstracción*).

Los abusos de notación son:

§ Si `x` es una aplicación o una abstracción, se pueden ignorar los paréntesis.

§ Si `x` es una abstracción cuyo cuerpo es otra abstracción, se pueden agrupar los argumentos de ambas abstracciones, e.g `(\x.(\y.M))` es equivalente a `(\x y.M)`.

§ Si `x` es una aplicación anidada con asociación a la izquierda, se pueden escribir los términos en las aplicaciones de manera consecutiva, e.g `(((a b)c)d)` es equivalente a `(a b c d)`.

## Ejemplos

### Átomos

Escribiendo `\lc{x}` se obtiene $x$.

Escribiendo `\lc*{x}` se obtiene $x$.

### Abstracción lambda

Escribiendo `\lc{\x.x}` se obtiene $(\lambda x.x)$.
Escribiendo `\lc*{\x.x}` se obtiene $\lambda x.x$.

### Aplicación lambda

Escribiendo `\lc{x y z}` se obtiene $((x\ y)\ z)$.
Escribiendo `\lc*{x y z}` se obtiene $x\ y\ z$.

### Numerales de Church

Escribiendo `\lc{\f x.f(f(f(f(f x))))}` se obtiene $(\lambda f.\,(\lambda x.\,(f\ \ (f\ \ (f\ \ (f\ \ (f\ x)))))))$.
Escribiendo `\lc*{\f x.f(f(f(f(f x))))}` se obtiene $\lambda f\ x.f\ \ (f\ \ (f\ \ (f\ \ (f\ x))))$.

### Términos lambda variados

Escribiendo `\lc{x y z (y x)}` se obtiene $(((x\ y)\ \ z)\ \ (y\ x))$.
Escribiendo `\lc*{(((x y) z) (y x))}` se obtiene $x\ y\ z\ \ (y\ x)$.

Escribiendo `\lc{\x.u x y}` se obtiene $(\lambda x.\,((u\ x)\ \ y))$.
Escribiendo `\lc*{(\x.((u x) y))}` se obtiene $\lambda x.u\ x\ y$.

Escribiendo `\lc{\u.u(\x.y)}` se obtiene $(\lambda u.\,(u\ \ (\lambda x.y)))$.
Escribiendo `\lc*{(\u.(u (\x.y)))}` se obtiene $\lambda u.u\ \ (\lambda x.y)$.

Escribiendo `\lc{(\u.v u u)z y}` se obtiene $(((\lambda u.\,((v\ u)\ \ u))\ \ z)\ \ y)$.
Escribiendo `\lc*{(((\u.((v u) u)) z) y)}` se obtiene $(\lambda u.v\ u\ u)\ \ z\ y$.

Escribiendo `\lc{u x(y z)(\v.v y)}` se obtiene $(((u\ x)\ \ (y\ z))\ \ (\lambda v.\,(v\ y)))$.
Escribiendo `\lc*{(((u x)(y z))(\v.(v y)))}` se obtiene $u\ x\ \ (y\ z)\ \ (\lambda v.v\ y)$.

Escribiendo `\lc{(\x y z.x z(y z))u v w}` se obtiene $((((\lambda x.\,(\lambda y.\,(\lambda z.\,((x\ z)\ \ (y\ z)))))\ \ u)\ \ v)\ \ w$
Escribiendo `\lc*{(((((\x.(\y.(\z.((x z)(y z)))))u)v)w)}` se obtiene $(\lambda x\ y\ z.x\ z\ \ (y\ z))\ \ u\ v\ $

## Estilos

Para obtener diferentes estilos de términos, se puede utilizar el comando `\lc{x}` con argumentos extras: `\lc[args]{x}`, donde `x` es un término lambda como en los anteriores comandos y `args` son las banderas (o *flags*) que determinan el formato del término.

Las banderas admitidas son `s`, `v`, `l`, `d` y `p`. Si ejecutas el comando `./texlambda --help` obtendrás la siguiente descripción de las banderas:

```
TeX-LaMbDa [ <option> ... ] <str>
 where <option> is one of
  -s, --spaced : Spaced terms mode - Introduces spacing
  -v, --bold-variables : Bold variables mode - Make variable names bold
  -l, --bold-lambdas : Bold lambdas mode - Makes lambdas bold
  -d, --bold-dots : Bold dots mode - Makes dots bold
  -p, --bold-parentheses : Bold parentheses mode - Makes parentheses bold
  -e, --explicit : Explicit mode - Removes abuse of notation
  --help, -h : Show this help
  -- : Do not treat any remaining argument as a switch (at this level)
 Multiple single-letter switches can be combined after one '-'; for
  example: '-h-' is the same as '-h --'
```

El modo explícito es controlado por el modificador estrella en el comando `lc`, así que no debes utilizar la bandera `e`.

## Ejemplos de modificación de estilos

Por ejemplo, para obtener "*negritas*" en las lambdas y puntos, se utiliza el comando `\lc[ld]{x}`, también sirve usar como banderas `dl` ya que el orden no importa:

$$((((\boldsymbol{\lambda} x\boldsymbol{.}\ (\boldsymbol{\lambda} y\boldsymbol{.}\ (\boldsymbol{\lambda} z\boldsymbol{.}\ ((x\ z)\ \ (y\ z)))))\ \ u)\ \ v)\ \ w)$$

Si queremos tener únicamente los átomos en "*negritas*" se escribe `\lc[v]{x}`:

$$((((\lambda \boldsymbol{x}.(\lambda \boldsymbol{y}.(\lambda \boldsymbol{z}.((\boldsymbol{x}\ \boldsymbol{z})\ (\boldsymbol{y}\ \boldsymbol{z})))))\ \boldsymbol{u})\ \boldsymbol{v})\ \boldsymbol{w})$$

La versión no explícita de este término sería `\lc*[v]{x}`:

$$(\lambda \boldsymbol{x}\ \boldsymbol{y}\ \boldsymbol{z}.\boldsymbol{x}\ \boldsymbol{z}\ (\boldsymbol{y}\ \boldsymbol{z}))\ \boldsymbol{u}\ \boldsymbol{v}\ \boldsymbol{w}$$

Y si deseamos un término lambda mas espaciado se puede utilizar `\lc*[vs]{x}`:

$$(\ \lambda\ \boldsymbol{x}\ \boldsymbol{y}\ \boldsymbol{z}\ .\ \boldsymbol{x}\ \boldsymbol{z}\ (\ \boldsymbol{y}\ \boldsymbol{z}\ )\ )\ \boldsymbol{u}\ \boldsymbol{v}\ \boldsymbol{w}$$

Si queremos tener todo en "*negritas*" excepto las variables, utilizamos `\lc[pdl]{x}`:

$$((((\boldsymbol{\lambda} x.(\boldsymbol{\lambda} y.(\boldsymbol{\lambda} z.((x\ z)\ (y\ z)))))\ u)\ v)\ w)$$

Y con `\lc*[pdl]{x}`:

$$(\boldsymbol{\lambda} x\ y\ z.x\ z\ (y\ z))\ u\ v\ w$$

Es posible asignar las banderas por defecto utilizando el comando `\lcflags{args}`, de tal manera que si se asignan banderas utilizando este comando, todos los términos lambda escritos con `\lc` o `\lc*` sin argumentos extra, utilizarán estas banderas. Por ejemplo al escribir `\lcflags{pld}`:

§ Escribiendo `\lc{\f x.f(f(f(f x)))}` se obtiene $(\boldsymbol{\lambda} f.(\boldsymbol{\lambda} x.(f\ (f\ (f\ (f\ x))))))$

§ Escribiendo `\lc*{\f x.f(f(f(f x)))}` se obtiene $\boldsymbol{\lambda} f\ x.f\ (f\ (f\ (f\ x)))$

Con este comando se pueden redefinir las banderas y escribiendo `\lcflags{}` se eliminan, regresando a los valores por defecto originales.

Por el momento es lo único que puede estilizar el programa `texlambda`, sin embargo estoy trabajando en poder realizar con comandos de latex y de manera declarativa $\beta$-reducción y $\alpha$-conversión.

## Operaciones y equivalencias

El paquete TEX-$\mathcal{LAMBDA}$ también tiene algunos comandos para escribir operaciones y equivalencias utilizadas frecuentemente en la literatura. Por el momento son:

§ `\betaredu` → $\beta$-reducción.

§ `\alphaconv` → $\alpha$-conversión.

§ `$\synteq$` → $\equiv$.

§ `$\termlen{\lc{\x.x}}$` → $\|(\lambda x.x)\|$.

§ `\alphacong` → $\alpha$-congruente.

Hacen falta muchas para completar las que utilizaré en mi tesis de licenciatura, sin embargo, aún no establezco una notación fija para las operaciones.

Apéndice B

# Compilación del cálculo lambda

Máquina virtual SECD, implementación y compilación del cálculo lambda.

Apéndice C

# Intérprete del cálculo lambda

Intérprete del cálculo lambda.

# Índice de figuras

2.1. Transformación de (2.1) a (2.2). . . . . 16

# Índice de cuadros

# Índice de algoritmos

# Índice de códigos

# Bibliografía


[1] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, *Studies in Logic and the Foundations of Mathematics*, tomo 103. North Holland, revised ed edición, 1984.

[2] F. Cardone y J. R. Hindley. «Lambda-calculus and combinators in the 20th century». En D. M. Gabbay y J. Woods, editores, «Logic from Russell to Church», *Handbook of the History of Logic*, tomo 5, 723 – 817. North-Holland, 2009.

[3] A. Church. «A set of postulates for the foundation of logic». *Annals of Mathematics*, 1932. 33(2):346–366.

[4] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jork, 1941.

[5] J. R. Hindley y J. P. Seldin. *Lambda-Calculus and Combinators*. Cambridge University Press, 2 edición, 2008.

[6] J. McCarthy. «Recursive functions of symbolic expressions and their computation by machine, part i». *Commun. ACM*, 1960. 3(4):184–195.

[7] R. Milner. «Logic for computable functions – description of a machine implementation». Informe Técnico CS-TR-72-288, Stanford University, Department of Computer Science, 1972.

[8] S. Peyton Jones et al. «The Haskell 98 language and libraries: The revised report». *Journal of Functional Programming*, 2003. 13(1).

[9] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[10] J. B. R. S. C. Kleene. «The inconsistency of certain formal logics». *Annals of Mathematics*, 1935. 36(3):630–636.

[11] D. S. Scott. «Lambda calculus then and now». En «ACM Turing Centenary Celebration», ACM-TURING '12. ACM, 2012 .

[12] G. J. Sussman y G. L. Steele Jr. «SCHEME : an interpreter for extended lambda calculus».

Informe Técnico AI 349, Massachussetts Institute of Technology (MIT). Cambridge (MA US), 1975.