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, 29 de marzo de 2016.

Resumen

Falta escribir el resumen.

Falta escribir la dedicatoria.

Agradecimientos

Falta escribir los agradecimientos.

Índice general

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 [3].

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 [7], Scheme [13] y Haskell [9]; así como en demostradores automáticos de teoremas como LCF [8].

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 [3].

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* [4] 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 β-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 [12] 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 λ-calculus.
- Mosses (2004) Modular Structural Operational Semantics.
- Taylor (2005) A λ-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 [5], 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 [4] 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 [11].

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 [11], ésta fue llamada cálculo-λI 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

El cálculo lambda es un sistema matemático creado con la finalidad de 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 y para mitigar las posibles confusiones que se puedan presentar en la lectura de este trabajo se presentará en este capítulo una introducción que servirá como transición de la manera en como usalmente se trabaja con funciones a como se trabajará con funciones en el resto de este documento.

El presente capítulo se divide en dos secciones: en la primer sección se abordará una introducción informal al cálculo lambda, en donde se presentará la terminología utilizada en las explicaciones y ejemplos presentados, se establecerán las diferencias tanto conceptuales como de notación entre las funciones en el cálculo lambda y las funciones en otras áreas de las matemáticas; en la segunda parte se presenta una definición formal del cálculo lambda y en base a esta se definirán con precisión los términos y conceptos abordados en la primer sección.

2.1. Introducción

La definición usual de función es el de una relación entre un conjunto de entradas, llamado el dominio y un conjunto de salidas llamado codominio. Esta relación tiene además la propiedad de que cada elemento del dominio se relaciona exactamente con un elemento del codominio, escrito formalmente: Sean A y B dos conjuntos, una función f con dominio

A y codominio B es un subconjunto de $A \times B$, tal que $\forall a \in A$, si $(a,\ b) \in f$ y $(a,\ b') \in f$ con $b,\ b' \in B$, entonces $b = b'$.

Las funciones tienen varias maneras de ser representadas. En la definición anterior la representación es la de pares ordenados en donde el primer elemento del par es un elemento del dominio y el segundo es un elemento del codominio. Dependiendo del contexto en el que se trabaja con las funciones puede ser conveniente representarlas simbolicamente como fórmulas, graficamente como dibujos, numéricamente como tablas o incluso verbalmente con palabras.

Es posible utilizar esta definición para *describir* las funciones en el cálculo lambda, para esto tendríamos que establecer cuál es el dominio y codominio de las funciones; después presentar una representación conveniente para las reglas de correspondencia en el sistema. Sin embargo, hay algunas propiedades del cálculo lambda que hacen que esta definición no pueda ser directamente aplicada. En particular, el cálculo lambda como sistema formal es una *teoría ecuacional*, lo cuál significa que toda la teoría matemática del cálculo lambda está basada en la manera en como se escriben simbolicamente fórmulas en el sistema y como se verá mas adelante, estas fórmulas no se componen de conjuntos, colecciones, clases o secuencias ni de conectivos o cuantificadores lógicos.

Las fórmulas en el cálculo lambda sólo nos permiten representar variables, funciones y el acto de *aplicar* una fórmula a otra. El concepto de aplicar hace alusión a la aplicación de funciones, es decir, el acto de aplicar una función a un argumento (el cuál es elemento de su dominio), para obtener el valor correspondiente (elemento del codominio). Por ejemplo, considerando la función $f(x) = x^2$, aplicar f a 4 es $f(4) = 16$. Este concepto, así como la representación de funciones, es en algunos aspectos mas restrictivo que en otras áreas de las matemáticas, ya que las fórmulas no contienen números como el 2, ni operaciones como la exponenciación, por lo tanto, la función $f(x) = x^2$ no puede ser escrita directamente como una fórmula del cálculo lambda. Por otro lado, la aplicación en el cálculo lambda es menos restrictiva que la aplicación de funciones de otras áreas de las matemáticas, por ejemplo, una fórmula válida en el cálculo lambda es la aplicación de una variable en otra.

En general, las fórmulas asociadas a los conceptos de función y de aplicación de fun-

ciones se pueden definir únicamente en términos de otras fórmulas, las cuales a su vez pueden ser sólo variables, funciones o aplicaciones. Esto no significa que al trabajar con este sistema formal no podamos trabajar también con teoría de conjuntos, números o lógica, ya que al igual que las palabras en el español, estos objetos y herramientas matemáticas forman parte del *metalenguaje* que utilizaremos en este trabajo, es decir, son parte del lenguaje utilizado cuando se examina o estudia el cálculo lambda.

Al tratar con funciones en el cálculo lambda, se omite hablar del dominio o codominio de la las funciones, esto es debido a que si consideramos el conjunto de todas las fórmulas expresables del cálculo lambda, todas las funciones tienen a este conjunto como dominio y codominio. Esto después debe ser tratado con cuidado, ya que abordaremos diferentes objetos matemáticos que se pueden representar como fórmulas del cálculo lambda y las fórmulas que definamos como funciones para operar con estas representaciones seguirán teniendo el mismo dominio y codominio. Por ejemplo, es posible representar cualquier número natural solo con fórmulas del cálculo lambda y también es posible tener una representación de la operación de exponenciación, con estas dos componentes se puede *emular* la función $f : \mathbb{N} \to \mathbb{N}$, $f(x, y) = x^y$, mas sin embargo la fórmula que representa esta operación seguirá teniendo como dominio y codominio el conjunto de todas las fórmulas del cálculo lambda, esto significa que será valido aplicar esta representación de exponenciación a fórmulas que no sean representaciones de números naturales y el resultado de dicha aplicación no necesariamente será una fórmula que represente a un número natural.

El hecho de tener estas limitaciones en las fórmulas nos permite poder entender de manera muy clara y precisa todos los procesos que tranformen la estructura de una fórmula, a tal grado que todas las operaciones que se realizan sobre las fórmulas pueden reproducirse paso a paso de manera mecánica, manipulando los símbolos que componen a las fórmulas.

A continuación se abordará una introducción informal a la notación usada para expresar fórmulas del cálculo lambda y se enfatizarán algunas diferencias mencionadas previamente.

2.1.1. Notación

La notación utilizada para escribir la definición y aplicación de funciones en los ejemplos anteriores fué la de expresar la regla de correspondencia como una fórmula simbólica. En el cálculo lambda, también se aborda ésta representación, pero los símbolos utilizados para escribir las fórmulas son definidos con precisión de antemano, en contraste con otras áreas de las matemáticas, en donde la notación de las reglas de correspondencia puede ser extendida de manera arbitraria ya sea para incluír operaciones sobre distintos objetos matemáticos (como el producto vectorial), abstraer repetición de operaciones (como $\sum_{i=0}^{n}$) o incluso incrustar en la notación procesos no finitos (como límites al infinito $\lim_{x\to\infty}$).

Para introducir la notación del cálculo lambda, consideramos la función identidad $I : \mathbb{N} \to \mathbb{N}$ definida como $I(x) = x$.

En esta notación, I se compone de la especificación de argumentos y valores, en donde aseveramos que a lo que nos referiremos como I es una función cuyo dominio y codominio es $\mathbb{N}$, después se establece la regla de correspondencia en donde establecemos que cuando I sea aplicada a un argumento $x \in \mathbb{N}$ el elemento en el codominio será equivalente a la fórmula del lado derecho de la igualdad, en donde toda aparición de la x en la fórmula hace referencia al argumento particular al que le fué aplicado I.

En el cálculo lambda, no se considera el dominio ni el codominio de I, e incluso, no se considera el nombre con el que nos referimos a la función. La manera en como se escribe esta función como una fórmula en el cálculo lambda es

$$(\lambda x.x)$$

el símbolo "λ" nos indica que la fórmula es una función, y el símbolo "." separa la variable que hace referencia al argumento al que la función será aplicada y la fórmula de la regla de correspondencia.

La aplicación de fórmulas se denota de manera diferente también, mientras que en la notación tradicional se escribe $I(y)$ considerando que $y \in \mathbb{N}$, en el cálculo lambda, debido a que no se nombran las funciones, se escribe explícitamente la función a la que hacemos

referencia

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

en ambos casos, *realizar* la aplicación consiste en sustituír el argumento concreto en la fórmula de la regla de correspondencia y el resultado en ambos casos sería y. Sin embargo, en el cálculo lambda no podemos expresar que $((\lambda x.x)\ \ y) = y$ sin antes mencionar de manera explícita el significado que se le dará a la igualdad entre dos fórmulas.

Como se mencionó anteriormente, lo único que se puede escribir en el cálculo lambda son fórmulas, por lo tanto, en el ejemplo anterior "x", "y", "$(\lambda x.x)$" y "$((\lambda x.x)\ \ y)$" son fórmulas del cálculo lambda y representan las tres diferentes clases de fórmulas: variables, funciones y aplicaciones.

A pesar de ser aparentemente una notación mas inconveniente debido a que se requiere escribir mas símbolos para expresar la misma idea y no se puede extender su notación como es usual en matemáticas, 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. En futuras secciones se abordarán técnicas para representar diversos objetos matemáticos y operaciones. Este incremento en la capacidad de expresar ideas y conceptos seguirá manteniendo la misma uniformidad en la manera en como se escriben las fórmulas y seguirá utilizando únicamente variables, funciones y aplicaciones.

La notación utilizada nos permitirá también, estudiar y tener mas control sobre la manera en cómo las fórmulas son transformadas y podremos asociarles un significado preciso, tanto en lo que las fórmulas representan conceptualmente hablando, como en las maneras en las que las fórmulas pueden ser manipuladas, aplicadas y simplificadas.

Un ejemplo de la importancia de la asignación explícita del significado operacional de las fórmulas es el de los posibles problemas que se pueden encontrar con algunos modelos de sustitución en la definición de funciones usual en otras áreas de las matemáticas. Consideremos la función factorial $f : \mathbb{N} \to \mathbb{N}$

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

Sabemos que para obtener el resultado deseado, al momento de aplicar f en 5, primero verificamos si $5 = 0$, en donde, si fuera el caso, el resultado sería 1, pero ya que $5 \neq 0$, el resultado es $5 \times f(4)$, el proceso mecánico de sustituír el argumento en la fórmula de la regla de correspondencia consiste en primero verificar si la condición es cierta antes de proceder en sustituír el valor del argumento en el consecuente correspondiente. Si este modelo de sustitución no se especifica para el uso de la notación de casos presente en el ejemplo, se pudieran contemplar otras maneras de sustituír al 5 en la fórmula, por ejemplo, sustituyendo el argumento en todas las apariciones de la variable n, luego expandir el valor de la aplicación de funciones y posteriormente decidir el resultado final verificando si el argumento cumple la condición. Sin embargo, decidir usar este modelo de sustitución resultaría en realizar una infinidad de sustituciones debido a la naturaleza recursiva de la definición.

A pesar de que se puede definir el factorial de un número de tal manera que no sea tan importante el modelo de sustitución, en el ejemplo anterior como $f(n) = \prod_{i=1}^{n} i$, estas definiciones ocultan el orden en el que se realizan las transformaciones mecánicas al aplicar f a un número y mas aún, hacen que sea difícil asignar el orden debido a la gran variedad de notaciones que se pueden utilizar para referirse a la misma operación.

En el estudio cotidiano del cálculo lambda, es usual referirse a una fórmula en particular por lo que sus partes representan, en el resto de esta sección nos referiremos a una variable entre la λ y el punto de una función como *argumento* y a la fórmula después del punto y antes del paréntesis cerrado como el *cuerpo* de la función. A continuación se muestran algunos ejemplos de fórmulas junto con la descripción textual de su estructura:

a.	x	Variable;
b.	$(\lambda x.x)$	Función cuyo cuerpo es su argumento;
c.	$(y\ \ (\lambda x.x))$	Aplicación de una variable en una función;
d.	$((\lambda y.\,(y\ \ (\lambda x.x)))\ \ (\lambda w.w))$	Aplicación de una función en otra función;
e.	$(\lambda x.\,(x\ x))$	Función cuyo cuerpo es la aplicación de su argumento sobre sí mismo;
f.	$(\lambda f.\,(\lambda x.\,(f\ x)))$	Función cuyo cuerpo es una función.

Las variables en el cálculo lambda son fórmulas por si solas, en el inciso a aparece la variable x la cual no es ni una función ni una aplicación; las variables por si solas en el cálculo lambda casi no tienen utilidad, pero al ser partes de una fórmula se enriquece su uso: en el caso del inciso b la misma variable x es el cuerpo de la función y como es también el argumento, esta variable obtiene el potencial de convertirse en cualquier otra fórmula a partir de la aplicación de $(\lambda x.x)$.

En el inciso c se tiene una aplicación inusual, es la variable y siendo aplicada a una función. Lo mas usual es tener fórmulas en donde lo que se aplica es una función, sin embargo si $(y\ \ (\lambda x.x))$ fuera el cuerpo de una función, entonces y jugaría un papel mas relevante. Esto se puede apreciar en el inciso d, en donde la fórmula del inciso c es el cuerpo de una función con argumento y y esta función está siendo aplicada a otra función. Este ejemplo nos permite abordar dos ideas importantes, primero, las funciones pueden ser aplicadas a funciones y segundo el realizar la aplicación del ejemplo d, la variable y tomará el valor de $(\lambda w.w)$ y podrá ser ahora aplicada a la función $(\lambda x.x)$:

1. $((\lambda y.\,(y\ \ (\lambda x.x)))\ \ (\lambda w.w))$	fórmula del inciso d
2. $((\lambda w.w)\ \ (\lambda x.x))$	al aplicar $(\lambda y.\,(y\ \ (\lambda x.x)))$ a $(\lambda w.w)$
3. $(\lambda x.x)$	al aplicar $(\lambda w.w)$ a $(\lambda x.x)$

En este último ejemplo se describió una secuencia de transformaciones mecánicas sobre los símbolos de la fórmula, este procedimiento tiene algunos detalles que son importantes recalcar pero se abordarán cuando se describa la formalización del cálculo lambda en la siguiente sección. Por el momento se describen los últimos dos incisos los cuales presentan dos conceptos interesantes.

En el inciso e se tiene una función cuyo cuerpo es la aplicación de su argumento sobre sí mismo. Lo interesante de esta fórmula es que encapsula la idea de replicar cualquier fórmula a la que se aplique. Por ejemplo, si aplicamos la fórmula a la variable y y realizamos el proceso de aplicación similar al mostrado con el anterior ejemplo, obtendremos $(y\ y)$ como resultado, lo cuál no es un resultado muy interesante. Sin embargo si aplicamos la fórmula a sí misma obtendremos lo que Douglas Hofstadter acuñó como "*quine*" en una

de las tantas exploraciones sobre la autoreferencia entre la Tortuga y Aquiles en [**?**]:

1. $((\lambda x.(x\ x))\ \ (\lambda x.(x\ x)))$	fórmula del inciso e aplicada a si misma
2. $x \leftarrow (\lambda x.(x\ x))$	valor que tomará x en el cuerpo de $(\lambda x.(x\ x))$
3. $(x\ x)$	fórmula en donde se sustituirá x
4. $((\lambda x.(x\ x))\ \ (\lambda x.(x\ x)))$	al completar la sustitución

Como podemos observar, el resultado de la aplicación es la fórmula inicial, a pesar de que el término quine se asoció originalmente a una paradoja sobre valores de verdad [**?**], hoy en día hace referencia a un programa que tiene como resultado el código fuente de él mismo.

El inciso f es una función cuyo cuerpo es una función, en donde el cuerpo de esta última es la aplicación del argumento de la primer función al argumento de la segunda. El concepto interesante que ilustra esta fórmula es el de funciones de varias variables: Aplicar esta fórmula a una segunda fórmula y posteriormente aplicar el resultado de la aplicación a una tercera fórmula produce el mismo resultado a que si tuvieramos una función de dos argumentos f y x cuyo cuerpo es $(f\ x)$ y aplicaramos esta fórmula hipotética a la misma segunda y tercera fórmula. Siguiendo una notación similar a la que hemos estado utilizando, el resultado de la operación

$$((\lambda f\ x.(f\ x))\ g\ y)$$

sería equivalente a

$$(((\lambda f.(\lambda x.(f\ x)))\ g)\ y)$$

Otra manera de representar funciones de varias variables sería la de representar a tuplas en el cálculo lambda y tener fórmulas para obtener cada elemento de una tupla. Sin embargo, representar tuplas es un mecanismo mas complejo que se abordará en otra sección. Las fórmulas presentes en los últimos dos ejemplos pueden resultar difíciles de leer por el uso de los paréntesis tanto en las funciones y en las aplicaciones, sin embargo, un abuso de notación será introducido en la siguiente sección y teniendo en cuenta las tres clases de fórmulas será mas fácil leerlas e identificar sus partes.

En este momento podría ser difícil imaginar cómo se representan conceptos complejos con el cálculo lambda, solo hemos abordado de manera muy informal y poco precisa la habilidad de replicar fórmulas como el quine y simular funciones de varias variables con funciones cuyos cuerpos son funciones. Sin embargo, como se aborda en futuras secciones, muchas representaciones son mas sencillas de lo que pudieran aparentar, para el lector curioso: ¡ *la fórmula del inciso* f *es una posible representación del número 1* !.

¡¡TERMINAR DE REVISAR ESTA SECCIÓN!!

2.1.2. 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.

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

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

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

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.

Considerndo 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 α-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 λ 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.

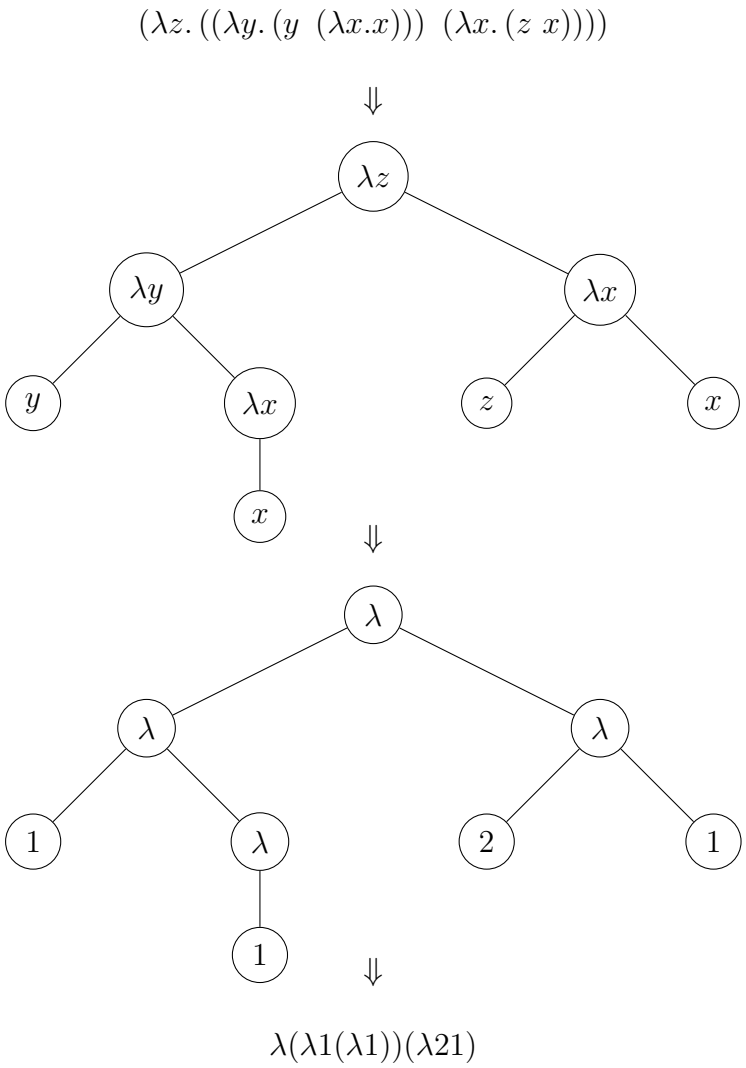

Figura 2.1: Transformación de (2.1) a (2.2).

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 λ 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 α-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 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 *β-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, 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 *η-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 juzgan dos objetos a partir de su evaluación. Sin embargo, las equivalencias de aplicación y de redundancia no comprenden 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 se dice que son ext equivalentes.

Una simple regla con fuertes implicaciones Existe una regla, llamada regla ξ, la cual establece una equivalencia muy sencilla: si dos expresiones M y N son equivalentes, entonces las expresiones $(\lambda x.M)$ y $(\lambda x.N)$ también lo son.

Aunque esta regla aparente aportar poco y pueda ser considerada innecesaria si combinamos todas las equivalencias previamente descritas, es suficiente para eliminar la equivalencia de redundancia y la equivalencia computacional de la formalización del cálculo lambda, la cuál será abordada en la siguiente sección.

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 estas dos expresiones sean equivalentes bajo algún criterio específico.

En el caso de la α-congruente la operación correspondiente consiste en cambiar nombres de variables, en la β-*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 η-*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.

2.2. Formalización de la teoría $\boldsymbol{\lambda}$

La teoría $\boldsymbol{\lambda}$ es el conjunto de axiomas que definen formalmente al cálculo lambda como sistema formal, el objeto de estudio principal de esta teoría es el del conjunto cociente formado a partir de un conjunto de fórmulas bien formadas y una relación de equivalencia. En las siguientes subsecciones se definirán estos conceptos, los cuales nos permitirán comenzar el estudio formal del cálculo lambda.

2.2.1. Fórmulas bien formadas

Una fórmula bien formada es un objeto formal sintáctico al que se le puede asociar un significado semántico. Para definir las fórmulas bien formadas de un lenguaje no es necesario darle un significado a cada fórmula, pero sí es importante expresar de manera rigurosa cómo se constituye simbolicamente.

El conjunto de fórmulas bien formadas del cálculo lambda es llamado *términos lambda*, denotado como Λ. Este conjunto tiene elementos que son fórmulas construidas a partir

del alfabeto Σ. Éste alfabeto es un conjunto que se conforma por los símbolos $(\ ,\ )\ ,\ .\ ,\ \lambda$ y una infinidad de símbolos $v,\ v',\ v'',\ \ldots$, etc. A esta secuencia infinita de símbolos v^i se denotará como V, de tal manera que $\Sigma = \{\ (\ ,\ )\ ,\ .\ ,\ \lambda\ \} \cup V$.

Λ es el conjunto mas pequeño tal que:

$$x \in V \Rightarrow x \in \Lambda \tag{2.3}$$

$$M,\ N \in \Lambda \Rightarrow (M\ N) \in \Lambda \tag{2.4}$$

$$M \in \Lambda,\ x \in V \Rightarrow (\lambda x.M) \in \Lambda \tag{2.5}$$

Cada una de estas tres reglas corresponde a los tres tipos de términos lambda. La regla (2.3) implica que los símbolos en V son términos lambda, estos símbolos son llamados *átomos*; la regla (2.4) implica que dos términos lambda entre paréntesis también son términos lambda, a este tipo de términos se les llama *aplicaciones*; la regla (2.5) implica que si se tiene entre paréntesis el símbolo λ seguido de un átomo, un punto y un término lambda cualquiera, entonces ésta expresión también es un término lambda, a este tipo de términos lambda se les llama *abstracciones*.

Desde la perspectiva de lenguajes formales, $\Lambda = L(G)$, donde G es una gramática libre de contexto con categorías sintácticas T (términos lambda), E (aplicaciones), F (abstracciones) y A (átomos); símbolos terminales $\{\ (\ ,\ )\ ,\ .\ ,\ \lambda\ ,\ v,\ '\ \}$; símbolo inicial T y con las siguientes reglas de producción:

1. $T \rightarrow E\ \mid\ F\ \mid\ V$
2. $E \rightarrow (\ T\ T\ )$
3. $F \rightarrow (\ \lambda\ A\ .\ T\ )$
4. $A \rightarrow v\ \mid\ E\ '$

Para facilitar la escritura de términos lambda, en este trabajo se realizan las siguientes consideraciones sobre la notación:

I. Cuando se hace referencia a cualquier término lambda se utilizan las letras mayúscu-

las M, N, O, etc. Es importante establecer que si en un ejemplo, explicación, teorema o demostración hacemos referencia a un término lambda con una letra mayúscula, cualquier otra aparición de esta letra hará referencia a este mismo término.

II. Cuando se hace referencia a cualquier átomo se utilizarán las letras minúsculas x, y, z, w, etc. Al igual que en el punto anterior, la aparición de una letra minúscula en un ejemplo, explicación, teorema o demostración hará referencia al mismo término.

III. Los paréntesis serán omitidos de acuerdo a las siguientes equivalencias sintácticas:

   a) $((M\ N)\ O) \equiv M\ N\ O$, en general, se considera la aplicación de términos lambda como una operación con asociación a la izquierda. Se tiene que tener cuidado con respetar la asociación, por ejemplo $(M\ (N\ (O\ P))) \equiv M\ (N\ (O\ P)) \not\equiv M\ N\ O\ P$.

   b) $(\lambda x.\,(M\ N)) \equiv \lambda x.M\ N$, en general, se puede escribir una abstracción omitiendo los paréntesis externos siempre y cuando no se escriba un término sintácticamente diferente. Por ejemplo $((\lambda x.\,(M\ N))\ O) \equiv (\lambda x.M\ N)\ O \not\equiv \lambda x.M\ N\ O$ ya que el lado derecho de la equivalencia es sintácticamente equivalente a $\lambda x.M\ N\ O$.

   c) $(\lambda x.\,(\lambda y.\,(\lambda z.M))) \equiv \lambda x\ y\ z.M$, en general, si el subtérmino a la derecha del punto en una abstracción es también una abstracción, se pueden agrupar los átomos antes del punto de ambas abstracciones después de una λ y antes que el punto, dejando el subtérmino después del punto de la segunda abstracción, como el del nuevo término.

IV. Se utilizará el símbolo $=$ para representar la igualdad entre dos objetos matemáticos que no sean términos lambda, como por ejemplo números.

La notación explicada en *III.a)* proviene de la reducción usada por Schönfinkel, en donde funciones de varias variables se transformaban a funciones de una sola variable. **¡¡REFERENCIA DE SCHÖNFINKEL AQUI!!**

2.2.2. Relación de equivalencia

Una relación de equivalencia es una relación binaria $\sim$ sobre elementos de un conjunto X, donde $\sim$ es reflexiva, simétrica y transitiva, es decir:

§ $a \in X \Rightarrow a \sim a$

§ $a,\ b \in X,\ a \sim b \Rightarrow b \sim a$

§ $a,\ b,\ c \in X, a \sim b,\ b \sim c \Rightarrow a \sim c$

En el estudio formal del cálculo lambda, la relación de equivalencia asociada a los términos lambda es llamada *convertibilidad*. Ésta relación es generada a partir de axiomas y para formular estos axiomas es necesario formalizar el concepto de *sustitución*.

DEFINIR SUSTITUCIÓN

DEFINIR ALPHA CONVERSIÓN Y SUS DETALLES FINOS

DEFINIR BETA CONVERSIÓN Y SUS DETALLES FINOS

DEFINIR REGLA XI Y SUS DETALLES FINOS

2.2.3. Conjunto cociente

EXPLICAR THIS SHIT Y CLASES DE EQUIVALENCIA

2.2.4. Axiomas de $\boldsymbol{\lambda}$

Sean $M,\ N,\ Z \in \Lambda$ y $x,\ y \in V$, la convertibilidad en la teoría $\boldsymbol{\lambda}$ se genera a partir de los siguientes axiomas:

$$M \sim M \qquad \text{(reflexividad)} \tag{2.6}$$

$$M \sim N \Rightarrow N \sim M \qquad \text{(simetría)} \tag{2.7}$$

$$M \sim N,\ N \sim L \Rightarrow M \sim L \qquad \text{(transitividad)} \tag{2.8}$$

$$M \sim N \Rightarrow (M\ Z) \sim (N\ Z) \tag{2.9}$$

$$M \sim N \Rightarrow (Z\ M) \sim (Z\ N) \tag{2.10}$$

$$(\lambda x.M) \sim (\lambda y.M)\,[x := y] \qquad (\alpha\text{-conversión}) \tag{2.11}$$

$$((\lambda x.M)\ \ N) \sim M[x := N] \qquad (\beta\text{-conversión}) \tag{2.12}$$

$$M \sim N \Rightarrow (\lambda x.M) \sim (\lambda x.N) \qquad \text{(regla } \xi) \tag{2.13}$$

CLARIFICAR DIFERENCIAS SOBRE LAS TEORÍAS Y SUS NOMBRES

DESCRIBIR LÓGICA COMBINATORIA Y LA TEORIA LAMBDA I

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.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{texlambda}[2016/03/09 Typesetting for the untyped lambda calculus]

\RequirePackage{suffix}
\RequirePackage{amsmath}
\RequirePackage{amsfonts}
\RequirePackage{xifthen}
\RequirePackage{etoolbox}

\newcommand{\LcFlags}{}
```

```
\newcommand{\lcflags}[1]{
  \ifthenelse{\equal{#1}{}}{
    \renewcommand{\LcFlags}{}
  }{
    \renewcommand{\LcFlags}{-#1}
  }
}

\newcommand\lc[2][]{%
  \ifstrempty{#1}{
    \immediate\write18{
      \unexpanded{texlambda -e} \LcFlags \unexpanded{ " #2 " > lc-buf.tex}
    }
    \input{lc-buf.tex}
  }{
    \immediate\write18{
      \unexpanded{texlambda -e -#1 " #2 " > lc-buf.tex}
    }
    \input{lc-buf.tex}
  }
}

\WithSuffix\newcommand\lc*[2][]{
  \ifstrempty{#1}{
    \immediate\write18{
      \unexpanded{texlambda } \LcFlags \unexpanded{ " #2 " > lc-buf.tex}
    }
    \input{lc-buf.tex}
  }{
    \immediate\write18{
      \unexpanded{texlambda -#1 " #2 " > lc-buf.tex}
    }
    \input{lc-buf.tex}
  }
}

\newcommand{\betaredu}{$\beta$-reducción}

\newcommand{\alphaconv}{$\alpha$-conversión}
```

```
\newcommand{\alphacong}{$\alpha$-congruente}

\newcommand{\synteq}{\equiv}

\newcommand{\termlen}[1]{\left\lVert#1\right\rVert}


\DeclareRobustCommand{\TeXLaMbDa}{
  \TeX-$\mathcal{L\!\!{\scriptstyle A}\!M\!^{B}\!D\!}\boldsymbol{{\scriptstyle \lambda}}$
}
```

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.

```
(define (parse str)
  (read
   (open-input-string
    (string-append
     "("
     (regexp-replace* #rx"\\." (regexp-replace* #rx"\\\\" str " lambda ( ") " ) ")
     ")"))))

(define (normalize x)
  (match x
    [(list a) (normalize a)]
    [(list 'lambda (list args ...) body ...)
     (list 'lambda
           (list (car args))
           (normalize (if (null? (cdr args)) body (list 'lambda (cdr args) body))))]
    [(list a b ...)
     (foldl (lambda (x y) (list y x))
            (normalize a)
            (map normalize b))]
    [(? symbol? a) a]))

(define (denormalize x)
  (match x
    [(? symbol? a) a]
    [(list 'lambda (list arg) body)
```

```
     (cons 'lambda (cons (uncurry-args x) (denormalize (uncurry-body x))))]
    [(list a b)
     (match a
       [(? symbol? x)
        (list (denormalize a) (denormalize b))]
       [(list 'lambda (list arg) body)
        (list (denormalize a) (denormalize b))]
       [_
        (append (denormalize a) (list (denormalize b)))])]
    [(list a) x]))

(define (uncurry-args x)
  (match x
    [(list 'lambda (list arg) body) (cons arg (uncurry-args body))]
    [_ null]))

(define (uncurry-body x)
  (match x
    [(list 'lambda (list arg) body) (uncurry-body body)]
    [(? symbol? a) (list a)]
    [_ x]))

(define (read-lambda x)
  (normalize (parse x)))

(define (latexify x)
  (match x
    [(? symbol? a)
     (format "~a~a~a" (lc:variable-pre) a (lc:variable-post))]
    [(list 'lambda (list arg) body)
     (format "~a~a~a~a~a~a"
             (lc:left-paren)
             (lc:lambda)
             (latexify arg)
             (lc:abstraction-separator)
             (latexify body)
             (lc:right-paren))]
    [(list a b)
     (format "~a~a~a~a~a"
             (lc:left-paren)
```

```
             (latexify a)
             (lc:application-separator)
             (latexify b)
             (lc:right-paren))]))

(define (latexify* x)
  (match x
    [(? symbol? a)
     (format "~a~a~a"
             (lc:variable-pre)
             a
             (lc:variable-post))]
    [(list 'lambda (list args ...) body ...)
     (format "~a~a~a~a"
             (lc:lambda)
             (string-join (map latexify* args) (lc:application-separator))
             (lc:abstraction-separator)
             (latexify* body))]
    [(list a (list b ...) c ...)
     (match a
       [(list 'lambda (list args ...) body ...)
        (format "~a~a~a~a~a~a~a~a~a"
                (lc:left-paren)
                (latexify* a)
                (lc:right-paren)
                (lc:application-separator)
                (lc:left-paren)
                (latexify* b)
                (lc:right-paren)
                (if (null? c) "" (lc:application-separator))
                (latexify* c))]
       [_
        (format "~a~a~a~a~a~a~a"
                (latexify* a)
                (lc:application-separator)
                (lc:left-paren)
                (latexify* b)
                (lc:right-paren)
                (if (null? c) "" (lc:application-separator))
                (latexify* c))])]
```

```
    [(list a b ...)
     (match a
       [(list 'lambda (list args ...) body ...)
        (format "~a~a~a~a~a"
                (lc:left-paren)
                (latexify* a)
                (lc:right-paren)
                (if (null? b) "" (lc:application-separator))
                (latexify* b))]
       [_
        (format "~a~a~a"
                (latexify* a)
                (if (null? b) "" (lc:application-separator))
                (latexify* b))])]
    ['() ""]))

(define (assert-predicate pred)
  (lambda (x)
    (if (pred x)
        x
        (error "Expected argument doesn't satisfy predicate" x pred))))

(define lc:left-paren            (make-parameter "\\left( "   (assert-predicate string?)))
(define lc:right-paren           (make-parameter " \\right)"  (assert-predicate string?)))
(define lc:abstraction-separator (make-parameter " . "        (assert-predicate string?)))
(define lc:lambda                (make-parameter " \\lambda " (assert-predicate string?)))
(define lc:variable-pre          (make-parameter ""           (assert-predicate string?)))
(define lc:variable-post         (make-parameter ""           (assert-predicate string?)))
(define lc:application-separator (make-parameter "\\ "        (assert-predicate string?)))

(define spaced?           (make-parameter #f (assert-predicate boolean?)))
(define bold-variables?   (make-parameter #f (assert-predicate boolean?)))
(define bold-lambdas?     (make-parameter #f (assert-predicate boolean?)))
(define bold-dots?        (make-parameter #f (assert-predicate boolean?)))
(define bold-parentheses? (make-parameter #f (assert-predicate boolean?)))
(define explicit?         (make-parameter #f (assert-predicate boolean?)))
(define term              (make-parameter "" (assert-predicate string?)))

(define console-args
  (command-line
```

```
   #:program "TeX-LaMbDa"
   #:once-each
   [("-s" "--spaced")           "Spaced terms mode - Introduces spacing"
                                (spaced? #t)]
   [("-v" "--bold-variables")   "Bold variables mode - Make variable names bold"
                                (bold-variables? #t)]
   [("-l" "--bold-lambdas")     "Bold lambdas mode - Makes lambdas bold"
                                (bold-lambdas? #t)]
   [("-d" "--bold-dots")        "Bold dots mode - Makes dots bold"
                                (bold-dots? #t)]
   [("-p" "--bold-parentheses") "Bold parentheses mode - Makes parentheses bold"
                                (bold-parentheses? #t)]
   [("-e" "--explicit")         "Explicit mode - Removes abuse of notation"
                                (explicit? #t)]
   #:args (str)
   str))

(term console-args)

(when (bold-variables?)
  (lc:variable-pre "\\boldsymbol{")
  (lc:variable-post "}"))

(when (bold-lambdas?)
  (lc:lambda " \\boldsymbol{\\lambda} "))

(when (bold-dots?)
  (lc:abstraction-separator " \\boldsymbol{.} "))

(when (bold-parentheses?)
  (lc:left-paren "\\boldsymbol{\\left( \\right.} ")
  (lc:right-paren " \\boldsymbol{\\left. \\right)}"))

(when (spaced?)
  (lc:left-paren (string-append (lc:left-paren) "\\ "))
  (lc:right-paren (string-append " \\" (lc:right-paren)))
  (lc:lambda (string-append (lc:lambda) "\\ "))
  (lc:abstraction-separator (string-append "\\ " (lc:abstraction-separator) "\\ ")))

(if (explicit?)
```

```
(display (latexify (normalize (parse (term)))))
(display (latexify* (denormalize (normalize (parse (term))))))
```

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 $(\lambda x\ y\ z.x\ z\ \ (y\ z))\ u\,v\,w$.

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 β-reducción y α-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` → β-reducción.

§ `\alphaconv` → α-conversión.

§ `$\synteq$` → $\equiv$.

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

§ `\alphacong` → α-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

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] D. Bridges y E. Palmgren. «Constructive mathematics». En E. N. Zalta, editor, «The Stanford Encyclopedia of Philosophy», Winter 2013 edición, 2013.

[3] 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.

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

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

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

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

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

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

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

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

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

ACM-TURING '12. ACM, 2012 .

[13] 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.