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, 3 de abril 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 sin tipos

El cálculo lambda es un sistema formal creado con la finalidad de expresar, manipular y estudiar funciones. La manera en la que se trabaja con funciones en este sistema es diferente a como es usual en la matemática clásica. Por este motivo, se presenta una introducción informal que tiene como objetivo esclarecer estas diferencias.

La estructura de este capítulo se conforma de tres secciones: en la primera se aborda la introducción informal al cálculo lambda, en donde se presenta la terminología utilizada en las explicaciones y ejemplos, se describen algunas de las diferencias tanto conceptuales como de notación entre las funciones en este sistema y las funciones en la matemática clásica; en la segunda sección se presenta la formalización del cálculo lambda y en base a ésta se definen con precisión los conceptos abordados en la primer sección; en la tercer sección se exploran diferentes maneras de representar en el cálculo lambda algunos objetos y operaciones matemáticas utilizadas en el estudio de la computación.

1.1. Introducción

La definición de función en la matemática clásica es el de una relación entre un conjunto de entradas, llamado *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, formalmente:

Sean A y B dos conjuntos, una función f con dominio A y codominio B es un subconjunto del producto cartesiano $A \times B$, tal que para toda $a \in A$, existe $b \in B$ tal que $(a,\ b) \in f$ y si $(a,\ b') \in f$ con $b' \in B$, entonces $b = b'$.

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

Es posible utilizar esta definición para *describir* las funciones en el cálculo lambda, para esto se tiene que establecer cuál es el dominio y codominio de cada función; 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*, esto significa que la teoría formal del cálculo lambda solo se conforma de reglas de igualdad entre expresiones, estas reglas consideran únicamente la estructura de las expresiones y en el caso del cálculo lambda, las expresiones no se componen de conjuntos, conectivos o cuantificadores lógicos.

En el cálculo lambda existen expresiones para representar *variables*, *funciones* y *aplicaciones*. El concepto de aplicación hace alusión a la *aplicación de funciones*, es decir, el acto de obtener un elemento del codominio de una función, a partir de un elemento en su dominio, por ejemplo, considerando la función $f(x) = x^2$, aplicar f a 4 es $f(4) = 16$.

Las expresiones de funciones y aplicaciones en el cálculo lambda son en algúnos aspectos mas restrictivas que en la matemática clásica, ya que en las expresiones no se pueden escribir directamente números como el 2, ni operaciones como la exponenciación, por lo tanto, no es posible escribir directamente la función $f(x) = x^2$ como una expresión válida del cálculo lambda. Por otro lado, la aplicación en el cálculo lambda es menos restrictiva que la aplicación de funciones de la matemática clásica, ya que el cálculo lambda permite aplicar cualquier expresión válida a otra, no únicamente funciones a valores.

En general, las expresiones en el cálculo lambda asociadas a los conceptos de función y de aplicación de funciones se pueden escribir únicamente en términos de otras expresiones, las cuales a su vez pueden ser solo variables, funciones o aplicaciones. Esto no significa que al trabajar con este sistema no podamos trabajar también con teoría de conjuntos, aritmética, lógica o algúna otra rama de las matemáticas, esto solo significa que las expresiones en el *lenguaje* del sistema formal son restrictivas en la manera en la que se escriben. Sin embargo, al igual que las palabras en el español, en el lenguaje utilizado para examinar y describir el cálculo lambda es válido hacer uso de cualquier herramienta, ya sea matemática o computacional, a este otro lenguaje se le llama *metalenguaje*.

Al tratar con funciones en el cálculo lambda, se omite hablar de su dominio y codominio, esto es debido a que todas las funciones válidas tienen como dominio y codominio al conjunto de todas las expresiones válidas del cálculo lambda. Este detalle debe ser tratado con cuidado cuando se representan objetos y operaciones matemáticas en el cálculo lambda, ya que el dominio y codominio de estas operaciones segue siendo el del conjunto de todas las expresiones válidas del cálculo lambda, sin importar la operación que se represente. Por ejemplo, es posible representar cualquier número natural con expresiones válidas del cálculo lambda y también es posible tener una representación de la operación de exponenciación; se puede *emular* la función $f : \mathbb{N} \to \mathbb{N}$, $f(x, y) = x^y$, mas sin embargo la función del cálculo lambda que representa esta operación segue teniendo como dominio y codominio el conjunto de todas las expresiones válidas del cálculo lambda, esto significa que será valido aplicar esta representación de exponenciación a expresiones que no sean representaciones de números naturales y el resultado de dicha aplicación no necesariamente es una expresión que represente a un número natural.

El hecho de tener un lenguaje tan reducido y minimalista para las expresiones nos permite poder entender de manera clara y precisa todos los procesos de manipulación y transformación de la estructura de una expresión, a tal grado que todas las operaciones que se realizan sobre las expresiones pueden reproducirse paso a paso de manera mecánica, manipulando los símbolos que las conforman.

1.1.1. Notación

La notación utilizada en la matemática clásica para escribir la definición y aplicación de funciones suele ser la de expresar una regla de correspondencia como una expresión simbólica. En el cálculo lambda, también se utiliza ésta representación, pero los símbolos empleados para escribir las expresiones son definidos con precisión de antemano, en contraste con las expresiones 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, compactar repeticiones 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 la notación clásica, I se compone de la especificación de su dominio y codominio, en este caso $\mathbb{N}$, después se establece la regla de correspondencia la cual indica que, al aplicar I a un argumento $x \in \mathbb{N}$, el resultado es equivalente a la expresión del lado derecho de la ecuación, en donde toda aparición de la variable x hace referencia al argumento particular al que le fué aplicado I.

En el cálculo lambda, no se considera el dominio ni el codominio de las funciones, e incluso, no se considera el nombre con el que nos referimos a ellas. La manera en como I es escrita en este sistema es

$$(\lambda x.x)$$

el símbolo "λ" nos indica que la expresión es una función, y el símbolo "." separa la variable que hace referencia al argumento al que la función es aplicada y la expresión del lado derecho de la igualdad.

La aplicación de expresiones se denota de manera diferente también, mientras que en la notación clásica 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 las apariciones de x por y en la función, dando como resultado y. Sin embargo, no podemos afirmar que $((\lambda x.x)\ y) = y$ sin antes mencionar de manera explícita el significado que se le dá a la igualdad entre dos expresiones.

Como se mencionó anteriormente, lo único que se puede escribir en el cálculo lambda son variables, funciones definidas en términos de otra expresión y aplicaciones entre dos expresiones. Todas las partes de la aplicación del ejemplo anterior también son expresiones válidas: "$((\lambda x.x)\ y)$", "$(\lambda x.x)$", "y" y "x". Estas expresiones muestran la manera en como se escriben las aplicaciones, las funciones y en el caso de x y y, las variables.

A pesar de ser aparentemente una notación mas inconveniente debido a que se limita a tratar solo con tres clases de expresiones, esta notación nos permite ser mas explícitos en la descripción de expresiones y provee uniformidad en el lenguaje formal. Esta notación también permite tener mas control sobre la manera en que las expresiones son transformadas.

La estructura de las expresiones hace que sea mas directa la relación entre una expresión o una parte de la expreción y un significado. El significado de una expresión puede referirse a lo que la expresión represente conceptualmente hablando o a la manera en la que la expresión puede ser operada.

Un ejemplo de la importancia de la asignación explícita del significado operacional de las expresiones es el de los posibles problemas que se pueden encontrar cuando se realiza la sustitución al momento de aplicar una función a una expresión, desde la perspectiva de la matemática clásica, consideremos la función factorial $f : \mathbb{N} \to \mathbb{N}$ definida como

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

Para obtener el resultado correcto la aplicación de 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 expresión de la regla de correspon-

dencia 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 del análisis casos presente en el ejemplo, se pudieran contemplar otras maneras de sustituír al 5 en la expresión, 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, utilizar este modelo de sustitución en el ejemplo resulta en realizar una infinidad de sustituciones debido a la naturaleza recursiva de la definición.

En el uso cotidiano de las matemáticas, no se suele analizar el proceso de sustitución, sin embargo, en el cálculo lambda es de suma importancia. Esta diferencia se debe a que en la matemática clásica las comparaciones entre dos funciones o dos expresiones tienden a ser *declarativas*, es decir, se declaran las relaciones, aseverando que la expresión es cierta; mientras que en el cálculo lambda son *imperativas*, es decir, toda relación o equivalencia entre dos expresiones expresa un mecanismo para construir una expresión a partir de otra.

Un ejemplo de esta distinción es el manejo del concepto de función inversa desde ambas perspectivas, una definición declarativa es: Sea f una función cuyo dominio es A y cuya imágen es B, la función inversa de f es la función f^{-1} con dominio B e imagen A tal que, $f(a) = b$ si y sólo si $f^{-1}(b) = a$ con $a \in A$ y $b \in B$. En el cálculo lambda no es usual trabajar con este tipo de definiciones debido a que no describen un procedimiento mediante el cual se puede obtener f^{-1} a partir de f.

Al referirse a una expresión del cálculo lambda, usualmente, se conoce parcialmente su estructura, es decir, algúna descripción de sus partes. En el resto de esta sección nos referiremos a una variable entre la λ y el punto de una función como *argumento de la función* y a la expresión después del punto y antes del paréntesis cerrado como el *cuerpo de la función*. A continuación se muestran algunos ejemplos de expresiones:

a) x

b) $(\lambda x.x)$

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

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

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

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

Las variables en el cálculo lambda son expresiones válidas, 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 otra expresión, puede aumentar su importancia: en el caso del inciso b la misma variable x es el cuerpo de la función y como es también el argumento, esta variable tiene el potencial de convertirse en cualquier otra expresión a partir de la aplicación de la función $(\lambda x.x)$.

En el inciso c se tiene una aplicación inusual, es la variable y siendo aplicada a una función. Comunmente se trabaja con expresiones 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 expresión 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 toma el valor de $(\lambda w.w)$ y es ahora aplicada a la función $(\lambda x.x)$:

1. $((\lambda y.\,(y\ \ (\lambda x.x)))\ \ (\lambda w.w))$	expresión 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 describe una secuencia de transformaciones mecánicas sobre los símbolos de la expresión, este procedimiento tiene algunos detalles que son importan-

tes recalcar pero se abordan cuando se describa la formalización del cálculo lambda en la sección 1.2. 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 expresión es que encapsula la idea de replicar cualquier expresión a la que se aplique. Por ejemplo, si aplicamos la expresión a la variable y y realizamos el proceso de aplicación similar al mostrado con el anterior ejemplo, obtendremos $(y\ y)$ como resultado; si aplicamos la expresión a sí misma obtendremos un "*quine*" [7]:

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

Como podemos observar, el resultado de la aplicación es la expresión inicial, a pesar de que el término quine se asoció originalmente a una paradoja sobre valores de verdad [12], 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 otra 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 expresión es el de funciones de varias variables: Aplicar esta expresión a una expresión cualquiera M y posteriormente aplicar este resultado a otra expresión cualquiera N produce el mismo resultado a que si tuvieramos una función de dos argumentos f y x cuyo cuerpo es $(f\ x)$ y aplicaramos esta expresión hipotética a M y N.

Otra manera de trabajar con funciones de varias variables es la de representar a tuplas en el el cálculo lambda y tener expresiones para obtener cada elemento de una tupla. Sin embargo, representar tuplas es un mecanismo mas complejo que se aborda en la sección 1.3.

1.1.2. El concepto de igualdad

El concepto de igualdad es de grán interés en el cálculo lambda. En el desarrollo histórico del este sistema, el estudio de los criterios que permiten establecer que dos expresiones son iguales dió pié a una grán diversidad de variantes de la teoría original.

De la mano al concepto de igualdad, están los mecanismos de transformación de expresiones, estos mecanismos de transformación no son operadores dentro del lenguaje del cálculo lambda, si no mas bien, son transformaciones que permiten explorar las diferentes estructuras de las expresiones del cálculo lambda y son usados como metalenguaje para referirse a equivalencias entre dos expresiones. Hay una grán variedad de operaciones que pueden transformar expresiones, sin embargo, en esta subsección abordaremos las mas elementales.

Las definiciones precisas de cada transformación se plantearán en la sección 1.2, por el momento se describe el concepto de igualdad que engloba cada transformación elemental, describiendo el proceso mecánico de transformación asociado.

Sustitución

Un concepto que es de grán importancia para describir y definir las transformaciones que realizemos sobre expresiones es el de *sustitución*. Cuando se describió informalmente el proceso mecánico de la aplicación de funciones en los ejemplos anteriores se mencionó este concepto. A continuación se presenta una descripción mas detallada de este concepto abordado como operación de transformación.

La sustitución involucra dos expresiones M y N cualesquiera y una variable x, el proceso de transformación consiste en intercambiar todas las apariciones de la variable x en la expresión M por la expresión N, denotado $M[x := N]$. Es usual que en los sistemas formales, se tenga cuidado al definir la transformación de sustitución, los detalles de la transformación son pospuestos por el resto de la sección y se presentan ejemplos en donde

esta definición provisional es suficiente.

A manera de ejemplo, consideremos la sustitución de la variable y por x en la expresión $(y\ w)$, se escribe

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

y la expresión resultante es $(x\ w)$.

Retomando el proceso descrito en el ejemplo de la expresión del inciso *d*, el acto de aplicar la función $(\lambda y.\,(y\ \ (\lambda x.x)))$ en $(\lambda w.w)$ se escribe con esta notación como:

$$\begin{array}{ll} ((\lambda y.\,(y\ \ (\lambda x.x)))\ \ (\lambda w.w)) & (y\ \ (\lambda x.x))\,[y := (\lambda w.w)] \\ ((\lambda w.w)\ \ (\lambda x.x)) & w[w := (\lambda x.x)] \\ (\lambda x.x) & \end{array}$$

A diferencia de las transformaciones que se abordan a continuación, la sustitución no está relacionada directamente con algún concepto de igualdad mas que el que describe la operación por si misma. Sin embargo es la transformación fundamental sobre la cual se describe el resto.

Equivalencia de expresiones

Con la sustitución se pueden transformar expresiones del cálculo lambda independientemente del contexto en el que se utilizan y sin prestar atención a lo que la expresión representa conceptualmente. A excepción de la función identidad y una representación del número 1, no se han abordado expresiones que representen algún concepto mas allá de su estructura simbólica, y a pesar de ser tentador asignarle un significado preciso a cada expresión presentada, es importante seguir analizando únicamente su estructura para tratar el tema de equivalencia de expresiones.

Una pregunta que posiblemente surge cuando se estudian las expresiones en el cálculo lambda y mas aún cuando se comparan estas expresiones con las utilizadas en la matemática clásica es, si M y N son dos expresiones del cálculo lambda, ¿Son iguales?.

Usualmente podemos afirmar que dos expresiones son iguales cuando entendemos el contexto y el nivel de abstracción en el que se refiere a ellas. Por ejemplo, dos números se suelen considerar iguales si representan el mismo concepto con el que se desea trabajar, si observamos una ecuación como $3 = \frac{6}{2}$ sabemos de inmediato que la ecuación es cierta, a pesar de que $frac62$ explícitamente hace referencia a una división y los dos lados de la igualdad se escriban diferente simbolicamente. Cuando se involucran variables y operaciones el concepto de igualdad requiere contexto, considerando la ecuación $x \cdot y = y \cdot x$, es imposible poder aseverar si la ecuación es cierta sin establecer los valores que "x", "y" y "$\cdot$" representan, en caso que sean números naturales y la operación aritmética de multiplicación la ecuación es cierta, pero en caso que sean matrices y la operación de producto matricial, la ecuación es falsa.

Al abordar expresiones matemáticas, el contexto en el que se expresan casi siempre se puede inferir por la manera en como las expresiones son utilizadas y lo mas común es tener a la mano la definición de las estructuras matemáticas y operaciones utilizadas en las expresiones, sin embargo, en el estudio del cálculo lambda, se tiene que ser mas explícito en especificar el concepto de igualdad con el que se trabaja, es tan importante que modificar su significado, modifica los axiomas de la teoría.

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 son 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.

Consideredo 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

$$(\lambda z.\,((\lambda y.\,(y\ \ (\lambda x.x)))\ \ (\lambda x.\,(z\ x)))) \tag{1.1}$$

se escribe usando el índice de De Bruijn como

$$\lambda(\lambda 1(\lambda 1))(\lambda 21) \tag{1.2}$$

En la figura 1.1 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 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 utiliza 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

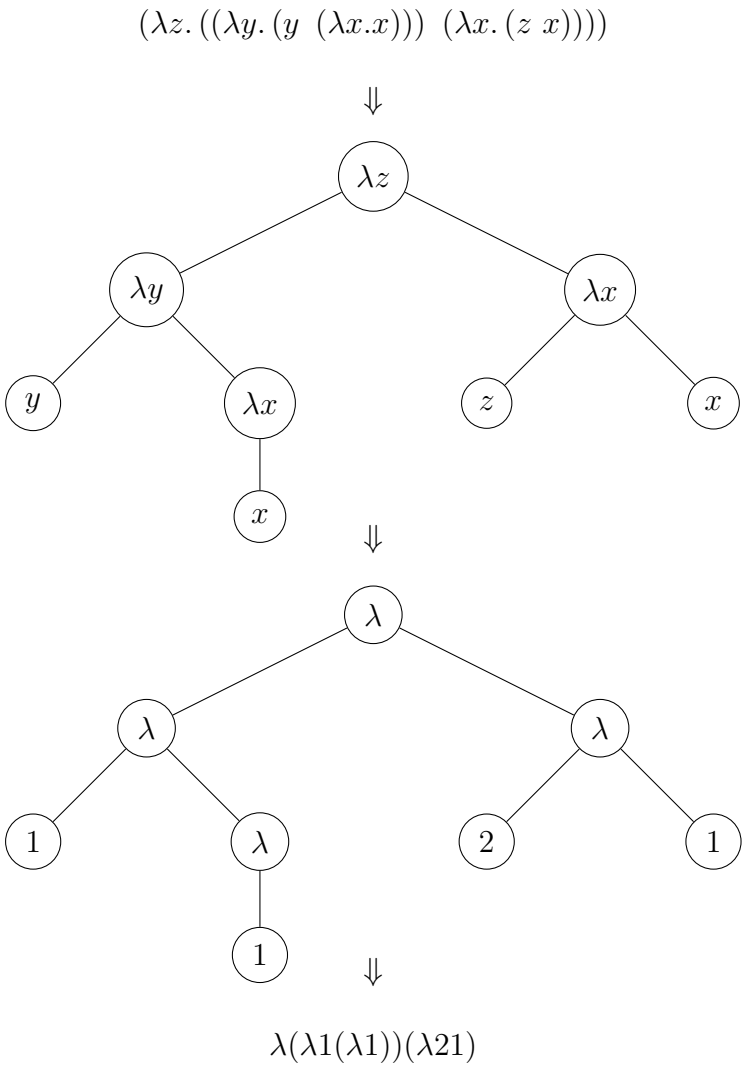

Figura 1.1: Transformación de (1.1) a (1.2).

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 son 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 resulta 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 es abordada en la sección 1.2.

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 definen 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.

1.2. 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 definen estos conceptos, los cuales nos permitirán comenzar el estudio formal del cálculo lambda.

1.2.1. 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 expresiones 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 denota como V, de tal manera que $\Sigma = \{\ (\ ,\ )\ ,\ .\ ,\ \lambda\ \} \cup V$.

Λ es el conjunto mas pequeño tal que:

$$x \in V \Rightarrow x \in \Lambda \tag{1.3}$$

$$M,\ N \in \Lambda \Rightarrow (M\ N) \in \Lambda \tag{1.4}$$

$$M \in \Lambda,\ x \in V \Rightarrow (\lambda x.M) \in \Lambda \tag{1.5}$$

Cada una de estas tres reglas corresponde a los tres tipos de términos lambda. La regla (1.3) implica que los símbolos en V son términos lambda, estos símbolos son llamados *átomos*; la regla (1.4) implica que dos términos lambda entre paréntesis también son términos lambda, a este tipo de términos se les llama *aplicaciones*; la regla (1.5) 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. $T \to E \ \mid \ F \ \mid \ V$
2. $E \to (\ T \ T \ )$
3. $F \to (\ \lambda \ A \ . \ T \ )$
4. $A \to v \ \mid \ E \ '$

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 mayúsculas 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 utilizan 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 hace referencia al mismo término.

III. Los paréntesis son 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 åtomos 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 utiliza 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 transformn a funciones de una sola variable [14].

1.2.2. 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 LOS CONCEPTOS NECESARIOS PARA HABLAR DE LO QUE SIGUE

DEFINIR SUSTITUCIÓN

$$x[x := M] \equiv M \tag{1.6}$$

$$y[x := M] \equiv y \qquad y \not\equiv x \tag{1.7}$$

$$(M\ N)\,[x := P] \equiv (M[x := P]\ N[x := P]) \tag{1.8}$$

$$(\lambda x.M)\,[x := N] \equiv (\lambda x.M) \tag{1.9}$$

$$(\lambda y.M)\,[x := N] \equiv (\lambda y.M) \qquad x \notin FV(P) \tag{1.10}$$

$$(\lambda y.M)\,[x := N] \equiv (\lambda\ y\ .\ M[x := N]) \qquad x \in FV(M),\ y \notin FV(N) \tag{1.11}$$

$$(\lambda y.M)\,[x := N] \equiv (\lambda\ z\ .\ M[y := z][x := N]) \qquad x \in FV(M),\ y \in FV(N) \tag{1.12}$$

En las ecuaciones e, f y g, la variable y debe de ser diferente a x y en el inciso g, la variable $z \in (FV(N) \cup FV(M))^c$.

DEFINIR ALPHA CONVERSIÓN Y SUS DETALLES FINOS

DEFINIR BETA CONVERSIÓN Y SUS DETALLES FINOS

DEFINIR REGLA XI Y SUS DETALLES FINOS

1.2.3. Conjunto cociente

EXPLICAR THIS SHIT Y CLASES DE EQUIVALENCIA

1.2.4. 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:

$$M \sim M \qquad \text{(reflexividad)} \tag{1.13}$$

$$M \sim N \Rightarrow N \sim M \qquad \text{(simetría)} \tag{1.14}$$

$$M \sim N,\ N \sim L \Rightarrow M \sim L \qquad \text{(transitividad)} \tag{1.15}$$

$$M \sim N \Rightarrow (M\ Z) \sim (N\ Z) \tag{1.16}$$

$$M \sim N \Rightarrow (Z\ M) \sim (Z\ N) \tag{1.17}$$

$$(\lambda x.M) \sim (\lambda y.M)\,[x := y] \qquad (\alpha\text{-conversión}) \tag{1.18}$$

$$((\lambda x.M)\ \ N) \sim M[x := N] \qquad (\beta\text{-conversión}) \tag{1.19}$$

$$M \sim N \Rightarrow (\lambda x.M) \sim (\lambda x.N) \qquad \text{(regla } \xi) \tag{1.20}$$

CLARIFICAR DIFERENCIAS SOBRE LAS TEORÍAS Y SUS NOMBRES

DESCRIBIR LÓGICA COMBINATORIA Y LA TEORIA LAMBDA I

1.3. Representaciones

1.3.1. Álgebra booleana

1.3.2. Aritmética

1.3.3. Estructuras complejas

1.3.4. Técnicas de representación

Capítulo 2

Tipos en el cálculo lambda

Formulación e introducción a aplicaciones.

Capítulo 3

El cálculo lambda en el desarrollo de la computación

3.1. 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 [8], Scheme [16] y Haskell [10]; así como en demostradores automáticos de teoremas como LCF [9].

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.

3.2. 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 [15] 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.

3.3. 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.

3.3.1. 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 [13].

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 [13], 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.

3.3.2. 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.

3.3.3. 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 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

Apéndice B

Compilación del cálculo lambda

Apéndice C

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] D. R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979.

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

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

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

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

[12] W. V. Quine. *The Ways of Paradox, and Other Essays*. Harvard University Press, 1976.

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

[14] M. Schönfinkel. «Über die bausteine der mathematischen logik». *Mathematische Annalen*, 1924. 92(3):305–316.

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

[16] 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.