El cálculo lambda y los fundamentos de la computación

Eduardo Acuña Yeomans

2016

Índice general

Capítulo 1

Noción informal del cálculo lambda

El cálculo lambda es un sistema formal originalmente creado por Alonzo Church en 1932 con la finalidad de expresar, manipular y estudiar funciones para el desarrollo de los fundamentos de la lógica y las matemáticas [8, p. 248]. A lo largo de la historia, este sistema se ha adaptado para el estudio de los fundamentos de la computación y como sustento teórico para el desarrollo de lenguajes de programación.

Tres características fundamentales del cálculo lambda son

- El lenguaje e interpretación de las expresiones
- La manipulación y transformación de expresiones
- Las nociones de equivalencia entre expresiones

En este capítulo se describen estas tres características del cálculo lambda de manera informal, enfatizando las diferencias sintácticas, conceptuales y operacionales entre este sistema, la matemática clásica y la lógica de primer orden.

El contenido de este capítulo está basado en los primeros dos capítulos del libro *The Lambda Calculus, Its Syntax and Semantics* por H.P. Barendregt [5]; los capítulos 1 y 3 del libro *Lambda Calculus and Combinators, an Introduction* por J.R. Hindley y J.P. Seldin [13]; y la monografía titulada *The Calculi of Lambda-Conversion* por Alonzo Church [9].

En el estudio del cálculo lambda existen dos *lenguajes*

- El lenguaje de las expresiones del sistema
- El *metalenguaje* que describe al cálculo lambda

El lenguaje de las expresiones es un *lenguaje formal* que especifica las secuencias de símbolos que representan expresiones válidas del cálculo lambda, se relaciona con las clases de objetos del sistema que son válidos manipular, comparar y representar. Por otro lado, el metalenguaje es un lenguaje informal que permite describir *cómo* es que estas expresiones son manipuladas y analizadas, así como los mecanismos para representar conceptos y objetos matemáticos como expresiones.

1.1. Expresiones

Existen tres clases de expresiones en el cálculo lambda: los *átomos*, las *abstracciones* y las *aplicaciones*.

1.1.1. Átomos

Las expresiones más simples son los *átomos*, estos son objetos sintácticos usualmente representados con un símbolo como x, y o z. Los átomos son la representación simbólica de las variables [5, p. 577].

Cuando se tiene una expresión que se conforma de varias x se refiere al símbolo como "el átomo x", sin embargo es posible tener diferentes variables representadas con el mismo símbolo en la expresión y para referirse a una de ellas en particular se debe especificar en que parte de la expresión se encuentra la variable x a la que se hace referencia.

Al igual que en la matemática clásica y en la lógica de primer orden, las variables más relevantes son las que se *cuantifican*, por ejemplo en el estudio de funciones, la x en $f(x) = M$ y en el estudio de enunciados lógicos, la y en $\forall y\ M$ o $\exists y\ M$.

En matemáticas y en computación es usual *nombrar* valores, por ejemplo, declarar que $x = 8$ y expresar a x en diferentes contextos que no necesariamente son expresiones lógicas y funciones. En el cálculo lambda, los nombres son parte del metalenguaje y no del lenguaje de las expresiones.

1.1.2. Abstracciones

Las abstracciones son expresiones con *estructura*, es decir, se conforman de *partes* identificables. Las expresiones que son abstracciones representan la generalización de una expresión y son usualmente asociadas al concepto de *función*.

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 asocia 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 uso que se le dá a las funciones, puede ser conveniente representarlas simbólicamente con expresiones, gráficamente con dibujos, numéricamente con tablas o incluso verbalmente con palabras.

Las abstracciones en el cálculo lambda son representadas simbólicamente con un átomo y con otra expresión, se escriben de la forma $(\lambda x.M)$ donde x es algún átomo llamado variable enlazada o argumento y M es alguna expresión ya sea otra abstracción, una aplicación o un átomo a la cual llamamos cuerpo de la abstracción. Un ejemplo de abstracción es $(\lambda y.y)$ en donde el argumento y cuerpo de la abstracción son el átomo y.

Además de la semejanza sintáctica con las funciones y las expresiones lógicas cuantificadas, las abstracciones proveen el mecanismo para asociarle un valor a una variable. Sin embargo, tanto las expresiones de funciones como las fórmulas lógicas están basadas en conjuntos y en operaciones sobre conjuntos: $f(x) = M$ asocia a x un valor en el dominio de f en M y $\exists x\ M$ asocia a x valores en el dominio del discurso en el enunciado M. En contraste con el cálculo lambda, por definición no existen conjuntos, números, valores de verdad, ni cuantificadores lógicos en el lenguaje.

Es posible utilizar la definición de función para *describir* operaciones y transformaciones de expresiones en el cálculo lambda, o utilizar lógica de primer orden para aseverar verdades en el sistema, o utilizar

aritmética para cuantificar propiedades de las expresiones del cálculo lambda, sin embargo estos objetos matemáticos no están incrustados en el lenguaje de las expresiones y conforman lo que es el *metalenguaje*.

1.1.3. Aplicaciones

Las aplicaciones, al igual que las abstracciones, son expresiones con estructura. Se conforman por otras dos expresiones y se escriben de la forma $(M\,N)$ donde M y N son cualesquiera átomos, abstracciones o aplicaciones. Este tipo de expresiones son de cierta manera complementarias a las abstracciones ya que representan un potencial acto de concretar una expresión.

El concepto relacionado con las aplicaciones en la matemática clásica es el de *aplicación de funciones*, sin embargo, en el cálculo lambda se hace una distinción entre la expresión sintáctica de una aplicación y el acto de "evaluar" una aplicación.

En el estudio de funciones, la aplicación de una función f en x se denota $f(x)$ y es considerada igual al resultado de obtener el elemento del codominio de f a partir del valor de x. Por ejemplo, considerando la función $f(x) = x^2$, la expresión sintáctica de la aplicación de esta función a 8 es $f(8)$, mientras que el valor de la evaluación de esta aplicación es 64. La notación de una abstracción del cálculo lambda es similar a la notación de aplicación de funciones, pero con el paréntesis abierto antes del nombre de la función, es decir, $(f\ x)$ en lugar de $f(x)$.

En el cálculo lambda, el concepto relacionado con la evaluación de funciones es el de *reducción* de expresiones. En particular cuando se tiene la aplicación de una abstracción $(\lambda x.M)$ en algúna expresión N, se dice que $((\lambda x.M)N)$ se *reduce* a una expresión Z la cual es similar a M solo que con x cambiada por N. En la evaluación de funciones, el procedimiento es parecido, por ejemplo $f(3)$ donde $f = 2x^2 + x - 8$, se evalúa sustituyendo las x de $2x^2 + x - 8$ por 3, obteniendo así la expresión aritmética $2(3)^2 + 3 - 8$, la cual puede ser operada para obtener el resultado 13.

Las abstracciones y aplicaciones del cálculo lambda son en algunos aspectos más restrictivos que las funciones y la aplicación de funciones. La función considera dos conjuntos cualquiera y no importa que propiedades tengan sus elementos o que operaciones se pueden realizar sobre ellos. Por otro lado, las abstracciones y aplicaciones sólo pueden ser descritas a partir de otras expresiones del cálculo lambda.

1.1.4. Extensiones

Cuando se desea representar en el cálculo lambda alguna función, se deben *codificar* como expresiones del lenguaje los elementos del dominio y el codominio de la función, así como las operaciones entre elementos de ambos conjuntos. Por ejemplo, para representar la función $f : \mathbb{N} \to \mathbb{N}$ dada por $f(x) = x^2$ primero se deben codificar los números naturales con expresiones del cálculo lambda, esta codificación debe ser acompañada de la codificación de las operaciones aritméticas elementales como la suma y resta así como de los predicados sobre números naturales como discriminar entre el mayor de dos números o si un número es cero; posteriormente se debe expresar la operación de exponenciación de cualquier número natural como base y el número 2 como exponente. La codificación es similar a la implementación de algoritmos y estructuras de datos en lenguajes de programación.

El hecho de tener un lenguaje tan reducido y minimalista para las expresiones del cálculo lambda nos obliga a entender con detalle y precisión todos los procesos de manipulación y transformación de expresiones y siendo que todo lo que se representa con el cálculo lambda debe ser codificado como expresiones, los objetos representados pueden ser entendidos de la misma manera.

1.1.5. Ejemplos

Con solo átomos, aplicaciones y abstracciones se pueden formular expresiones complejas. A continuación se presentan seis ejemplos de expresiones del cálculo lambda y se describen diferentes maneras en las cuales estas se pueden componer para formar otras expresiones más complejas.

Ejemplo 1.1.1. Algunas expresiones del cálculo lambda

$$x \tag{a}$$
$$(\lambda x.x) \tag{b}$$
$$(y(\lambda x.x)) \tag{c}$$
$$((\lambda y.(y(\lambda x.x)))(\lambda w.w)) \tag{d}$$
$$(\lambda x.(xx)) \tag{e}$$
$$(\lambda f.(\lambda x.(f(fx)))) \tag{f}$$

Los átomos por si solos son expresiones válidas, en el inciso (a) aparece el átomo x, como tal no tiene mucha utilidad, no podemos decir que toma valores en algún conjunto o que representa algún valor en particular como falso o verdadero, es tan sólo un símbolo. Al ser parte de otra expresión, un átomo puede tener más relevancia, en el inciso (b) el átomo x es el cuerpo de la abstracción $(\lambda x.x)$ y ahora tiene el potencial de ser cambiado por cualquier otra expresión debido a que también es el argumento.

En el inciso (c) se tiene la aplicación del átomo y en la abstracción del inciso (b). A pesar de ser contraintuitivo, las expresiones de aplicación se componen de dos expresiones cualesquiera, por lo tanto, a pesar de estar asociada conceptualmente con la aplicación de funciones, la expresión $(y(\lambda x.x))$ es válida. La expresión del inciso (d) contiene la expresión anterior en una abstracción en la primer parte de la aplicación y nos permite observar dos ideas importantes: primero, las abstracciones pueden ser aplicadas a abstracciones; segundo, al reducir la aplicación de $(\lambda y.(y(\lambda x.x)))$ a $(\lambda w.w)$, el átomo y es cambiado por la expresión $(\lambda w.w)$, el resultado puede ser reducido nuevamente para obtener la expresión $(\lambda x.x)$.

Ejemplo 1.1.2. Procedimiento de reducir la aplicación de $(\lambda y.(y(\lambda x.x)))$ en $(\lambda w.w)$

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 el inciso (e) se presenta una abstracción cuyo cuerpo es la aplicación de su argumento sobre sí mismo. Lo interesante de esta expresión es que es que encapsula la idea de replicar cualquier expresión a la que sea aplicada. Por ejemplo, si aplicamos $(\lambda x.(x\ x))$ al átomo y y se realiza el procedimiento de reducción como en el ejemplo 1.1.2, se obtiene $(y\ y)$ y en general al realizar la reducción de $((\lambda x.(x\ x))M)$ donde M es cualquier expresión, se obtiene $(M\ M)$. Con esta expresión se puede formular una expresión auto-replicante en el cálculo lambda:

Ejemplo 1.1.3. Procedimiento de reducir la aplicación de $(\lambda x.(x\ x))$ en $(\lambda x.(x\ x))$

1. $((\lambda x.(x\ x))(\lambda x.(x\ x)))$	expresión del inciso (e) aplicada a sí misma

En $(x\,x)$ de la expresión izquierda se cambia cada x por $(\lambda x.(x\,x))$

2. $((\lambda x.(x\,x))(\lambda x.(x\,x)))$ resultado de la reducción.

A este tipo de expresiones se les llaman "quines" [14, pp. 431–437] término originalmente asociado a una paradoja sobre sistemas lógicos [22]. En la actualidad, el término "quine" hace referencia a un programa cuya *salida* es el programa mismo.

En el inciso (f) se tiene una abstracción cuyo cuerpo es otra abstracción. El concepto interesante que ilustra esta expresión es el de representación de abstracciones de varias variables. Al reducir la aplicación de $(\lambda f.(\lambda x.(f(f\,x))))$ en una expresión cualquiera M, se obtiene $(\lambda x.(M(M\,x)))$. Si posteriormente se reduce la aplicación de este resultado a una expresión cualquiera N se obtiene $(M(M\,N))$. El resultado final es similar al resultado que se obtendría al evaluar en M y N una función $g(f,x) = f(f(x))$, es decir, $g(M,N) = M(M(N))$.

Una manera de codificar funciones de varias variables como abstracciones del cálculo lambda es codificar a *tuplas* o *secuencias* y poder hacer referencia a sus elementos de manera individual, sin embargo, representar secuencias es un procedimiento más complejo que se aborda en la sección 3.4.1.

1.2. Operaciones

En el cálculo lambda se pueden realizar algunas operaciones para transformar expresiones, estas operaciones son parte del metalenguaje y consisten de una serie de cambios mecánicos a la estructura de las expresiones de acuerdo a un criterio particular.

En la sección 1.1 se describió de manera tangencial el acto de *reducir* una aplicación basándonos en la noción de *cambiar* una expresión por otra, en esta sección se describen operaciones basadas en esta misma noción.

1.2.1. Sustitución

La *sustitución* es la operación que nos permite transformar una expresión cualquiera M cambiando las apariciones de un átomo x por alguna otra expresión M, este procedimiento se denota

$$M\,[x := N]$$

En muchos casos la operación de sustitución se puede realizar de fácilmente:

Ejemplo 1.2.1. Sustituciones sencillas

$$x\,[x := y] = y \tag{a}$$
$$(x(x(\lambda y.y)))\,[x := z] = (z(z(\lambda y.y))) \tag{b}$$
$$(((w\,x)y)z)\,[x := a]\,[y := b] = (((w\,a)b)z) \tag{c}$$
$$(x\,x)\,[x :=(\lambda w.w)] = ((\lambda w.w)(\lambda w.w)) \tag{d}$$

Existen algunos detalles sutiles de la sustitución que se deben tomar en cuenta para evitar obtener expresiones erróneas, en particular cuando se sustituye en expresiones que contienen abstracciones. Para ilustrar estos casos especiales, consideremos la abstracción lambda análoga a una función constante

$(\lambda x.y)$, en donde reducir la aplicación de ésta en cualquier otra expresión, resulta siempre en el átomo y. Si se realiza la sustitución $(\lambda x.y)[y := z]$ se obtiene la expresión $(\lambda x.z)$ la cual también es análoga a una función constante pero con el átomo z. Si no se tiene cuidado, sustituir un átomo por otro en esta abstracción puede resultar en una expresión con diferente *interpretación*.

Ejemplo 1.2.2. Caso patológico de la sustitución ingenua

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

Se puede pensar que el resultado es $(\lambda x.x)$ la cuál es análoga a la función identidad, sin embargo, la sustitución no permite cambiar las expresiones de esta manera.

Para entender la operación de sustitución se tiene que pensar que lo que le da sentido a una variable x es una λx. Consideremos la expresión

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

el átomo x que aparece en el cuerpo de la expresión se dice ser una variable *ligada* a la λx, la cual se puede pensar como una especie de "referencia" a la expresión a la que la abstracción es aplicada, esto limita a la operación de sustitución a no *romper* la referencia de una variable ligada. De igual manera, el átomo y es una variable ligada a la λy y debe mantener su referencia bajo la operación de sustitución. Sin embargo, el átomo z es lo que se llama variable *libre*: No está en el *alcance* de alguna λz y puede ser libremente sustituida por alguna otra expresión.

En el ejemplo 1.2.2 se pretende sustituir la variable libre y por una expresión x, lo cual no debería presentar problemas, sin embargo, una sustitución tal cual de y por x *introduciría* una referencia a la λx de la expresión, la cuál no existía previamente. Con esto se identifica que la operación de sustitución $M[x := N]$ no debe introducir o eliminar referencias a alguna λ en M.

1.2.2. Cambio de variable ligada

Para resolver el problema presentado en el ejemplo 1.2.2 se debe considerar otra operación llamada *cambio de variable ligada*. Se parte de la observación que en una expresión del cálculo lambda, las referencias entre λx y las variables x son más importantes que el símbolo con el que se representa el átomo. En las expresiones simbólicas de funciones sucede lo mismo, al expresar $f(x) = x^2$ y $f(y) = y^2$ hacemos referencia a la misma regla de correspondencia y por lo tanto a la misma función (sin considerar el dominio y el codominio de f). En el cálculo lambda, cambiar el símbolo que representa el átomo x en la expresión $(\lambda x.y)$ por otro símbolo no utilizado como z nos permite realizar la sustitución sin problemas.

Ejemplo 1.2.3. Procedimiento de sustitución para el ejemplo 1.2.2

1. $(\lambda x.y)[y := x]$
2. $(\lambda z.y)[y := x]$ después de realizar un cambio de variable ligada
3. $(\lambda z.x)$ resultado del procedimiento de sustitución

Cuando se realiza un cambio de variable ligada sobre una abstracción $(\lambda x.M)$ se cambia tanto el átomo x acompañado por la λ, llamada variable *enlazada* como todas las apariciones del átomo en el cuerpo de la abstracción, también llamado *alcance de* λx a menos que en M se encuentre una expresión

de la forma $(\lambda x.N)$, ya que las x en N hacen referencia a la λx de $(\lambda x.N)$ no de $(\lambda x.M)$.

En el ejemplo 1.2.3 el cambio de variable ligada únicamente cambió la variable enlazada, en otras expresiones el cambio de variable ligada puede realizarse múltiples veces para transformar varias abstracciones.

Ejemplo 1.2.4. Múltiples cambios de variable ligada

1. $(\lambda f.(\lambda x.(f(f(f\,x)))))$	
2. $(\lambda g.(\lambda x.(g(g(g\,x)))))$	Cambiando f por g
3. $(\lambda g.(\lambda y.(g(g(g\,y)))))$	Cambiando x por y

El cambio de variable ligada en una abstracción $(\lambda x.M)$ de x a y resulta en la abstracción

$$(\lambda y.M\left[x := y\right])$$

La definición de la operación de sustitución es recursiva y hace uso de la operación de cambio de variable ligada, considerando a x, y, z como átomos diferentes y M, N y P como expresiones cualquiera:

- $x\,[x := M]$ resulta en M;
- $y\,[x := M]$ resulta en y;
- $(M\,N)\,[x := P]$ resulta en $(M\,[x := P]\,N\,[x := P])$;
- $(\lambda x.M)\,[x := N]$ resulta en $(\lambda x.M)$ debido a que las referencias a x no deben eliminarse;
- $(\lambda y.M)\,[x := N]$ resulta en:
  - $(\lambda y.M)$ cuando x no es una variable libre en M,
  - $(\lambda y.M\,[x := N])$ cuando x es una variable libre en M pero y no es una variable libre en N debido a que esto introduciría una referencia a λy,
  - $(\lambda z.M\left[y := z\right][x := N])$ cuando x es una variable libre en M y y es una variable libre en N.

1.2.3. Reducción de aplicaciones

La operación de *reducción de aplicaciones* es el mecanismo mediante el cual se puede "concretar" una abstracción haciendo uso de otra expresión como valor de la variable enlazada. De la misma manera en como se efectúa la evaluación de funciones en la matemática clásica, el concretar una función consiste en sustituir todas las apariciones del argumento por el valor en el que la función es aplicada.

La definición de la reducción de la aplicación de $(\lambda x.M)$ en una expresión cualquiera N es

$$M\,[x := N]$$

A continuación se presentan ejemplos de reducciones con los pasos de la transformación

Ejemplo 1.2.5. Procedimiento de reducir la aplicación de $(\lambda x.x)$ en y

1. $((\lambda x.x)y)$
2. $x\,[x := y]$
3. y

Ejemplo 1.2.6. Procedimiento de reducir la aplicación de $(\lambda w.w)$ en x dentro de otra expresión

1. $(\lambda x.((\lambda w.w)x))$
2. $(\lambda x.w\,[w := x])$
3. $(\lambda x.x)$

Ejemplo 1.2.7. Procedimiento de múltiples reducciones

1. $(((\lambda f.(\lambda x.(f(f(f\,x)))))g)y)$
2. $((\lambda x.(f(f(f\,x)))\,[f := g])y)$
3. $((\lambda x.(g(g(g\,x))))y)$
4. $(g(g(g\,x)))\,[x := y]$
5. $(g(g(g\,y)))$

El cálculo lambda es un sistema maleable y se permite definir operaciones arbitrarias sobre expresiones para estudiar como el sistema se comporta en diferentes contextos, por ejemplo, se puede considerar una operación similar a la sustitución que permite introducir referencias a una o más λ en una expresión, sin embargo, el presente trabajo está constituido para entender plenamente las ideas centrales del cálculo lambda haciendo solamente uso de las operaciones de *sustitución*, *cambio de variable ligada* y *aplicación de abstracciones*.

1.3. Equivalencias

El cálculo lambda se considera formalmente como una *teoría ecuacional*, esto significa que los axiomas del sistema formal son ecuaciones que relacionan expresiones del lenguaje. Esto hace que el concepto de *equivalencia* de expresiones sea de suma importancia.

Es tan relevante la formalización de las nociones de equivalencia que considerar alguna equivalencia entre dos expresiones que se escriben diferente puede cambiar por completo el sistema formal que se estudia. En el desarrollo histórico del cálculo lambda, el estudio de los criterios que permiten establecer que dos expresiones son equivalentes ha dado pie a una gran diversidad de variantes de la teoría original; es por ello que en la literatura se suele hablar de *los cálculos lambda* y no únicamente de un cálculo lambda.

Como se aborda en la sección 1.2, con la operación de sustitución se puede transformar expresiones del cálculo lambda y definir otras operaciones como el cambio de variable ligada y la reducción de aplicaciones. Usualmente, las transformaciones de expresiones se pueden asociar a nociones de equivalencia. En terminología del cálculo lambda, las nociones de equivalencia entre expresiones son asociadas a la propiedad de *convertibilidad*, la cual significa que si dos expresiones M y N son equivalentes en el

sistema, es posible transformar M a N y viceversa por medio de un número finito de operaciones.

En esta sección se describen algunos criterios de equivalencia entre expresiones del cálculo lambda y las maneras en las que las equivalencias se relacionan entre sí.

1.3.1. Equivalencia sintáctica

La *equivalencia sintáctica* es una relación binaria entre expresiones que no está asociada a una transformación. Se considera como una equivalencia trivial, ya que asevera la igualdad entre dos expresiones que son escritas exactamente igual, símbolo por símbolo a excepción de abusos de notación. Por ejemplo, la expresión $\sin^2(x)$ es un abuso de notación de $(\sin(x))^2$ y ambas se consideran exactamente iguales. En el cálculo lambda, la equivalencia sintáctica es denotada como

$$M \equiv N$$

cuando M es sintácticamente la misma expresión que N.

Todos los cálculos lambda, al igual que la mayoría de los sistemas formales, comprenden la noción de equivalencia sintáctica. Sin embargo las equivalencias más interesantes son las que involucran transformaciones entre expresiones.

1.3.2. α-convertibilidad

La operación de cambio de variable ligada se relaciona con una equivalencia estructural entre dos expresiones. Cuando se realiza esta operación no se modifica la estructura de la expresión, únicamente se modifica el símbolo usado para representar un átomo.

Considerando la expresión análoga a la función identidad $(\lambda x.x)$ se observa que tiene la misma estructura que la abstracción $(\lambda y.y)$ y que $(\lambda z.z)$, estas tres representan el mismo concepto. De igual manera otras expresiones como $((x\ y)z)$ o $(\lambda w.x)$ son estructuralmente equivalentes a $((a\ b)c)$ y $(\lambda f.h)$ respectivamente. A pesar de que no se escriben sintácticamente igual, la correspondencia que hay entre las posiciones de los átomos en una y otra expresión nos permite considerarlas como equivalentes. Sin embargo, la operación de cambio de variable ligada no considera cambios de nombres a átomos que sean variables libres.

Esta relación de equivalencia es llamada α-convertibilidad y se denota como

$$M =_\alpha N$$

para dos expresiones del cálculo lambda M y N en donde a partir de un número finito de cambios de variables ligadas en M o parte de M y en N o parte de N se puedan obtener expresiones sintácticamente equivalentes.

Índices de De Bruijn

Una técnica utilizada por algoritmos que verifican si dos expresiones M y N son α-convertibles es la de *índices de De Bruijn*, esta transformación cambia la aparición de átomos por números naturales que representan la "distancia" de los átomos a las λ que hacen referencia.

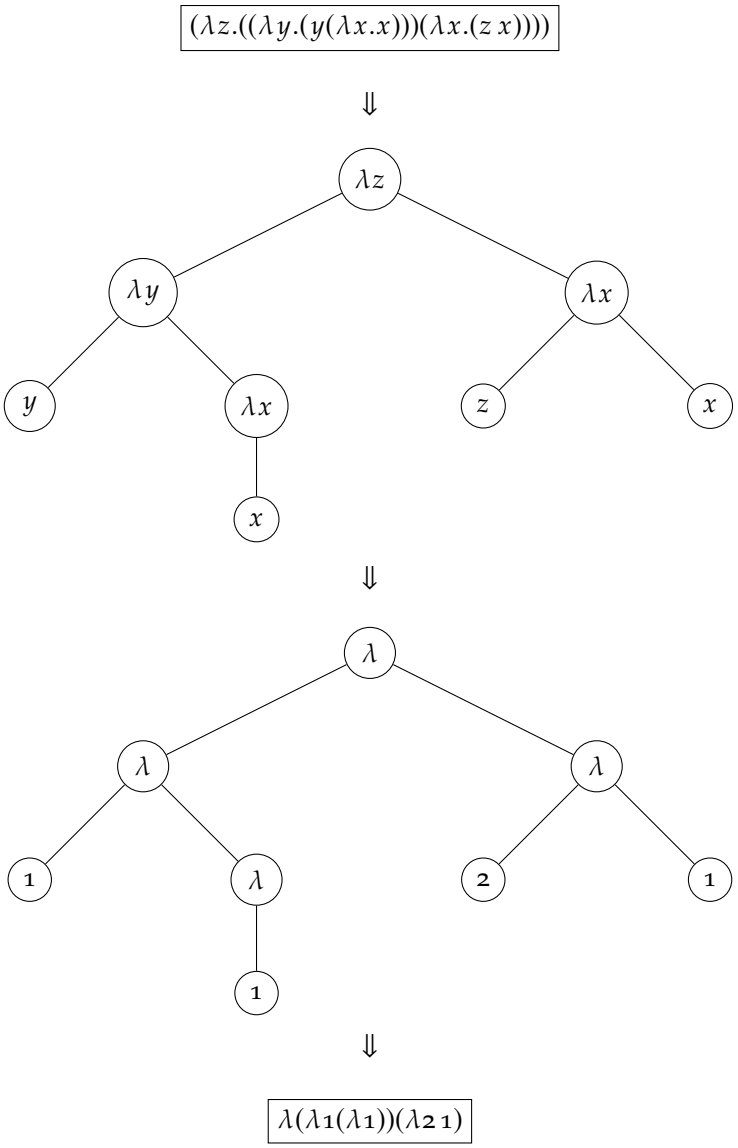


Figura 1.1: Transformación gráfica del ejemplo 1.3.1

Ejemplo 1.3.1 (índices de De Bruijn)**.**

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

Se escribe usando índices de De Bruijn como

$$\lambda(\lambda 1(\lambda 1))(\lambda 2\,1)$$

En la figura 1.1 se puede observar de manera gráfica la transformación de una notación a otra para este ejemplo, visualizando las expresiones del cálculo lambda como árboles.

Una desventaja de utilizar la notación de De Bruijn es que ciertas expresiones del cálculo lambda no pueden ser escritas, en particular, los átomos no pueden ser variables libres para que esta notación

pueda ser utilizada.

1.3.3. β-convertibilidad

Al igual que el cambio de variable ligada, la operación de reducción de aplicaciones es utilizada para describir una equivalencia entre expresiones. La idea básica de esta equivalencia consiste en observar que al aplicar una abstracción $(\lambda x.M)$ a una expresión N, el resultado de su reducción siempre es el mismo. De manera similar a la aplicación de funciones, cuando se define una función $f(x) = x^2$, la aplicación $f(3)$ se suele igualar al resultado de la aplicación: $f(3) = 8$.

Esta relación de equivalencia es llama β-convertibilidad y se denota como

$$M =_\beta N$$

para dos expresiones M y N en donde a partir de un número finito de reducciones a aplicaciones, reducciones inversas y cambios de variable ligada en M o parte de M y N o parte de N se puedan obtener expresiones sintácticamente equivalentes.

El proceso inverso a la reducción de aplicaciones se puede ilustrar considerando la reducción de $(((\lambda g.(\lambda y.(g(g(g\ y)))))f)x)$ a $(f(f(f\ x)))$. No es posible reducir la segunda expresión a la primera, pero ya la segunda expresión es una reducción de la primera, ambas son β-convertibles:

$$(f(f(f\ x))) =_\beta (((\lambda g.(\lambda y.(g(g(g\ y)))))f)x)$$

1.3.4. Relaciones de equivalencia

Todas las nociones de convertibilidad son relaciones de equivalencia, las cuales por definición cumplen con tres propiedades.

Sea $\sim$ una relación de equivalencia

a. Toda expresión M es equivalente a sí misma, es decir, $M \sim M$.

b. Si una expresión M es relacionada con una equivalencia a otra expresión N, entonces N también es relacionada a M, es decir $M \sim N \implies N \sim M$.

c. Si una expresión M se relaciona con una equivalencia a otra expresión N y N se relaciona con la misma equivalencia a P, entonces, M y P se relacionan con esta equivalencia, es decir, $M \sim N,\ N \sim P \implies M \sim P$.

La equivalencia sintáctica corresponde al inciso a de las propiedades de equivalencias mencionadas y es llamada propiedad de *reflexividad*; al igual que la α-conversión y la β-conversión, la equivalencia sintáctica no está asociada a una regla de inferencia. En los incisos b y c se tienen inferencias que parten de expresiones equivalentes y basado en si estas expresiones son equivalentes o no, ciertas propiedades se deben cumplir. En el inciso b la propiedad es llamada *simetría*, mientras que en el inciso c la propiedad es llamada *transitividad*.

La α-conversión y la β-conversión fueron definidas como equivalencias independientes y su definición cumple con las tres propiedades mencionadas a pesar de ser definidas en base a un procedimiento y no en una regla declarativa, sin embargo, es deseable referirse a una sola equivalencia de expresiones que

tenga las propiedades de *reflexividad*, *simetría* y *transitividad* y posteriormente considerar otras reglas que la equivalencia deba de cumplir.

Al igual que Haskell Curry en [12, p. 59] se utilizan las letras griegas α y β para referirse a las ecuaciones relacionadas con la α-conversión y β-conversión respectivamente y las letras ρ, σ y τ para referirse a las propiedades de reflexividad, simetría y transitividad respectivamente, se retoma esta convención para elaborar la siguiente definición de una relación de equivalencia $\sim$:

Definición 1.3.1 (Ecuaciones de $\sim$)**.** Las ecuaciones con $\sim$ que se satisfacen para expresiones del cálculo lambda son

$$(\lambda x.M) \sim (\lambda y.M\,[x := y]) \tag{α}$$
$$((\lambda x.M)N) \sim M\,[x := N] \tag{β}$$
$$M \sim M \tag{ρ}$$
$$M \sim N \implies N \sim M \tag{σ}$$
$$M \sim N,\ N \sim P \implies M \sim P \tag{τ}$$

Las ecuaciones en la definición 1.3.1 son muy parecidas a las propiedades de la β-conversión, con la excepción de que la β-conversión relaciona expresiones en donde sus partes fueron transformadas y $\sim$ no, por ejemplo

$$(\lambda f.((\lambda x.(f\ x))y)) =_\beta (\lambda f.(f\ y))$$

pero

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

Para capturar la definición de β-convertibilidad con ecuaciones, es necesario definir a $\sim$ en partes de una expresión. Las siguientes reglas, nombradas por Curry [12, p. 59] como ν, μ y ξ, junto con las reglas de $\sim$ completan la definición declarativa de β-convertibilidad:

Definición 1.3.2. Reglas que debe cumplir $\sim$ para ser $=_\beta$

$$M \sim N \implies (M\,Z) \sim (N\,Z) \tag{ν}$$
$$M \sim N \implies (Z\,M) \sim (Z\,N) \tag{μ}$$
$$M \sim N \implies (\lambda x.M) \sim (\lambda x.N) \tag{ξ}$$

Con estas reglas y a partir de un razonamiento lógico, podemos demostrar la β-equivalencia entre dos expresiones.

Ejemplo 1.3.2. Razonamiento para concluir que $(\lambda f.((\lambda x.(f\ x))y)) =_\beta (\lambda f.(f\ y))$

1. $((\lambda x.(f\ x))y) =_\beta (f\ y)$ por β
2. $(\lambda f.((\lambda x.(f\ x))y)) =_\beta (\lambda f.(f\ y))$ por ξ

Es posible incluir aún más reglas de equivalencia cuando se estudia el cálculo lambda, a pesar de poder trabajar con expresiones en este sistema a partir de equivalencias arbitrarias, usualmente cada

regla de equivalencia se asocia con alguna argumentación basada en la noción de función.

Por ejemplo, se pueden considerar dos abstracciones diferentes $(\lambda x.M)$ y $(\lambda y.N)$ que al ser aplicadas a cualquier expresión Z sean β-convertibles a una misma expresión W. Si se relacionan las abstracciones del cálculo lambda con funciones, es natural pensar que M y N sean equivalentes, ya que por definición, dos funciones f y g son equivalentes si para toda x en su dominio $f(x) = g(x)$. Por ejemplo, las funciones $f(n) = \sum_{i=0}^{n} i$ y $g(n) = \frac{n(n+1)}{2}$ a pesar de describir dos procedimientos diferentes para el cálculo de la suma de los primeros n números naturales son "funcionalmente" equivalentes ya que para todo natural $f(n) = g(n)$. Por otro lado, si se relacionan las abstracciones del cálculo lambda con algoritmos, M y N no pudieran ser consideradas equivalentes ya que en el estudio de la complejidad algorítmica, el énfasis en la comparación entre dos procedimientos no es las entradas y salidas, si no el proceso que describen. Por ejemplo, el algoritmo de ordenamiento *merge sort* logra ordenar una secuencia de n números de menor a mayor en $\mathcal{O}(n \log n)$ mientras que el algoritmo *bubble sort* computa el mismo resultado pero en $\mathcal{O}(n^2)$. La equivalencia "funcional" se pudiera incluír en la definición de $\sim$ añadiendo la siguiente regla:

$$(M\,P) \sim (N\,P) \implies M \sim N$$

Con esto se termina la introducción informal al cálculo lambda, las ideas que se han manejado en esta sección son formalizadas y definidas de manera rigurosa en el capítulo 2.

Capítulo 2

Formalización del cálculo lambda

La noción de *generalización* es de suma importancia en el estudio general de funciones, operaciones o transformaciones. Los predicados en la lógica de primer orden, las funciones en la matemática clásica, los algoritmos en la computación y las abstracciones en el cálculo lambda pueden ser considerados como implementaciones del concepto de generalización para los sistemas de los que forman parte y en algunos casos son la motivación original para el desarrollo de las teorías que los fundamentan.

El estudio de las propiedades generales de las funciones es una de las motivaciones originales del cálculo lambda, sin embargo, este cálculo se formuló de tal manera que es posible abstraer de su propósito original y ser tratado meramente como un sistema formal [9].

El presente capítulo tiene el objetivo de formalizar las ideas presentadas en el capítulo 1. La formalización del cálculo lambda se realiza desde dos perspectivas:

1. Construyendo una *teoría formal*, en donde un conjunto de axiomas y reglas de inferencia permiten plantear razonamientos lógicos para demostrar propiedades del cálculo lambda.

2. Formulando nociones de *reducción*, de tal manera que mediante procedimientos de transformación de expresiones del cálculo lambda, se puedan estudiar sus propiedades.

Independientemente de la perspectiva de la formalización, los conceptos son similares, se describe la misma idea general del cálculo lambda y ambos trabajan con el lenguaje formal de sus expresiones.

De acuerdo a Barendregt [5, p. 22], el objeto de estudio del cálculo lambda es el conjunto de términos lambda módulo convertibilidad. Estos conceptos serán presentados a lo largo de este capítulo.

El contenido de este capítulo está basado en los primeros cuatro capítulos del libro *The Lambda Calculus, Its Syntax and Semantics* de H.P. Barendregt [5] y los capítulos 1, 3, 6, 7 y 8 del libro *Lambda Calculus and Combinators, an Introduction* de J.R. Hindley y J.P. Seldin [13] así como el artículo *A Set of Postulates for the Foundation of Logic* y la monografía *The Calculi of Lambda-Conversion* de Alonzo Church [8, 9].

2.1. Términos lambda

Esta subsección está basada principalmente en el capítulo 2 de [5].

Los *términos lambda* son la formalización de las expresiones descritas en la sección 1.1. El conjunto de todos los términos lambda es un lenguaje formal Λ en donde sus elementos son cadenas compuestas de símbolos de un alfabeto [15].

El lenguaje Λ se puede definir de diferentes maneras, a continuación se presenta una definición inductiva y posteriormente una definición basada en una gramática libre de contexto.

Observación (Notación)**.**

- El símbolo $\implies$ denota una implicación lógica, $P \implies Q$ se lee "Si P, entonces Q".
- El símbolo $\longrightarrow$ denota una producción en una gramática, $P \longrightarrow Q$ se lee "P produce Q".
- El símbolo $\Rightarrow$ denota un paso en la derivación de una cadena.

Definición 2.1.1 (Términos lambda)**.** El conjunto Λ tiene elementos que son cadenas conformadas por símbolos en el alfabeto $\Sigma = \{(,\ ),\ .,\ \lambda\} \cup V$, donde V es un conjunto infinito $\{v_0,\ v_{00},\ \ldots\}$ de variables. Λ es el conjunto más pequeño que satisface:

$$x \in V \implies x \in \Lambda \tag{a}$$
$$M \in \Lambda,\ x \in V \implies (\lambda x.M) \in \Lambda \tag{b}$$
$$M,\ N \in \Lambda \implies (M\,N) \in \Lambda \tag{c}$$

Cada uno de estos tres incisos corresponde a las tres clases de términos lambda:

(a) establece que todo elemento de V es un término lambda a los cuales se les llama *átomos*;

(b) establece que las cadenas de la forma $(\lambda x.M)$ son términos lambda, donde x es un átomo y M es cualquier término lambda, a estos términos se les llama *abstracciones*;

(c) establece que las cadenas de la forma $(M\,N)$ son términos lambda, donde M y N son términos lambda cualesquiera, a estos términos se les llama *aplicaciones*.

En el estudio usual de lenguajes formales [15], Λ pertenece a la clase de lenguajes libres de contexto y puede ser definido de la siguiente manera:

Definición 2.1.2 (Términos lambda)**.** El conjunto de términos lambda es el lenguaje generado por la gramática libre de contexto G conformado por

categorías sintácticas T, E, F y A, las cuales denotan las reglas para derivar términos lambda, aplicaciones, abstracciones y átomos respectivamente;

símbolos terminales $\{(,\ ),\ .,\ \lambda,\ v,\ {}_0\}$, los cuales son los símbolos que conforman a las cadenas en Λ;

símbolo inicial T, el cual es el símbolo del que se derivan todos los términos lambda;

reglas de producción

$$T \to E \mid F \mid A \tag{a}$$
$$A \to \mathrm{v}_0 \mid A_0 \tag{b}$$
$$F \to (\lambda\ A\ .\ T\ ) \tag{c}$$
$$E \to (\ T\ T\ ) \tag{d}$$

Dada una secuencia de símbolos M, se pueden utilizar estas dos definiciones para verificar si M es o no un término lambda. En el caso de la definición inductiva, se debe presentar un razonamiento que pruebe que las partes de la cadena satisface la definición 2.1.1. En el caso de la gramática libre de contexto 2.1.2 se debe presentar una derivación de la cadena a partir de la categoría sintáctica T.

Ejemplo 2.1.1. Sea $M = (\lambda v_o.(v_{oo}(\lambda v_{oo}.v_{ooo})))$, la cadena M es un término lambda ya que

Por definición inductiva

$$\begin{aligned} v_{ooo} \in V &\implies v_{ooo} \in \Lambda; \\ v_{oo} \in V,\ v_{ooo} \in \Lambda &\implies (\lambda v_{oo}.v_{ooo}) \in \Lambda,\ v_{oo} \in \Lambda; \\ v_{oo},\ (\lambda v_{oo}.v_{ooo}) \in \Lambda &\implies (v_{oo}(\lambda v_{oo}.v_{ooo})) \in \Lambda; \\ v_o \in V,\ (v_{oo}(\lambda v_{oo}.v_{ooo})) \in \Lambda &\implies (\lambda v_o.(v_{oo}(\lambda v_{oo}.v_{ooo}))). \end{aligned}$$

Por gramática Se mantienen los espacios en los lados derechos de las producciones de la gramática para ser consistentes, sin embargo, el espacio en blanco no es un símbolo terminal, por lo tanto pueden ser omitidos.

$$\begin{aligned} T &\Rightarrow F \Rightarrow (\ \lambda\ A\ .\ T\ ) \Rightarrow (\ \lambda\ v_o\ .\ T\ ) \Rightarrow (\ \lambda\ v_o\ .\ E\ ) \Rightarrow (\ \lambda\ v_o\ .\ (\ T\ T\ )\ ) \\ &\Rightarrow (\ \lambda\ v_o\ .\ (\ A\ T\ )\ ) \Rightarrow (\ \lambda\ v_o\ .\ (\ A_o\ T\ )\ ) \Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ T\ )\ ) \\ &\Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ F\ )\ ) \Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ (\ \lambda\ A\ .\ T\ )\ )\ ) \\ &\Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ (\ \lambda\ A_o\ .\ T\ )\ )\ ) \Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ (\ \lambda\ v_{oo}\ .\ T\ )\ )\ ) \\ &\Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ (\ \lambda\ v_{oo}\ .\ A\ )\ )\ ) \Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ (\ \lambda\ v_{oo}\ .\ A_o\ )\ )\ ) \\ &\Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ (\ \lambda\ v_{oo}\ .\ A_{oo}\ )\ )\ ) \Rightarrow (\ \lambda\ v_o\ .\ (\ v_{oo}\ (\ \lambda\ v_{oo}\ .\ v_{ooo}\ )\ )\ ). \end{aligned}$$

Ejemplo 2.1.2. Sea $N = ((\lambda v_{oo}.v_o\ v_{oo})v_o)$, la cadena N no es un término lambda ya que

Por definición inductiva Ya que Λ se definió como el *conjunto más pequeño*, se demuestra que $N \notin \Lambda$ de la siguiente manera

$$\begin{aligned} (\lambda v_{oo}.v_o\ v_{oo}),\ v_o \in \Lambda &\implies ((\lambda v_{oo}.v_o\ v_{oo})v_o) \in \Lambda; \\ v_{oo} \in V,\ v_o\ v_{oo} \in \Lambda &\implies (\lambda v_{oo}.v_o\ v_{oo}) \in \Lambda; \\ v_o\ v_{oo} \notin \Lambda &\therefore ((\lambda v_{oo}.v_o\ v_{oo})v_o) \notin \Lambda. \end{aligned}$$

Por gramática La gramática no es ambigua, realizando una derivación por la izquierda

$$\begin{aligned} T &\Rightarrow E \Rightarrow (\ T\ T\ ) \Rightarrow (\ F\ T\ ) \Rightarrow (\ (\ \lambda\ A\ .\ T\ )\ T\ ) \\ &\Rightarrow (\ (\ \lambda\ A_o\ .\ T\ )\ T\ ) \Rightarrow (\ (\ \lambda\ v_{oo}\ .\ T\ )\ T\ ) \\ &\nRightarrow (\ (\ \lambda\ A\ .\ v_o\ v_{oo}\ )\ T\ ). \end{aligned}$$

La sintaxis del cálculo lambda es uniforme, lo cual permite identificar su estructura con facilidad y evitar ambigüedades, sin embargo, suele ser tedioso escribir términos largos debido al extenso uso de paréntesis. Es por esto que en este trabajo se hacen las siguientes consideraciones sobre la notación:

1. El símbolo $\equiv$ denota la equivalencia sintáctica entre dos términos lambda, esta equivalencia contempla las consideraciones de este listado.

2. Cuando se hace referencia a *cualquier* término lambda se utilizan las letras mayúsculas M, N, P, etc. Es importante establecer que si en un ejemplo, explicación, teorema o demostración se hace referencia a un término lambda con una letra mayúscula, cualquier otra aparición de esta letra hace referencia a este mismo término dentro de ese contexto.

3. Cuando se hace referencia a *cualquier* átomo se utilizan las letras minúsculas x, y, z, 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 átomo.

4. Los paréntesis son omitidos de acuerdo a las siguientes equivalencias sintácticas:

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

   b) $(\lambda x.(MN)) \equiv \lambda x.(MN)$, en general, se puede escribir una abstracción omitiendo los paréntesis externos. Es necesario escribir de manera explícita los paréntesis en algunos casos, 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 cuerpo de una abstracción es también una abstracción, se pueden agrupar las variables ligadas y enlazadas. Éste abuso de notación es consistente con la representación de funciones de varias variables usada por Schönfinkel [24].

5. Para hacer referencia a una secuencia con una cantidad arbitraria de términos lambda se usa la notación $\vec{x} = x_1, \ldots, x_n$ cuando es secuencia de átomos y $\vec{M} = M_1, \ldots, M_n$ cuando es secuencia de términos lambda en general. Con esta notación se puede abreviar la consideración de 4*a* como

$$((\ldots((M_1\,M_2)M_3)\ldots)M_n) \equiv \vec{M}$$

   y la consideración de 4*c* como

$$(\lambda x_1.(\lambda x_2.(\lambda x_3.\ldots(\lambda x_n.M)\ldots))) \equiv (\lambda \vec{x}.M)$$

   Ya que la notación no indica la cantidad de términos en la secuencia, se suele decir que $\vec{M}$ cabe en $\vec{N}$ cuando son secuencias con la misma cantidad de elementos.

6. Al escribir términos lambda con repetición de aplicaciones suele ser conveniente utilizar una notación más compacta. Cuando se aplica n veces un término F por la izquierda a otro término M se denota $F^n\,M$. Cuando se aplica n veces un término M por la derecha a otro término F se denota $F\,M^{\sim n}$. Por ejemplo, el término $(f(f(f(f\,x))))$ se puede denotar como $(f^4\,x)$ y el término $(f\,x\,x\,x\,x)$ se puede denotar como $(f\,x^{\sim 4})$. La definición inductiva de esta notación es:

$$\begin{aligned} F^{n+1}\,M &\equiv F(F^n\,M) \\ F^0\,M &\equiv M \end{aligned} \tag{2.3}$$

$$\begin{aligned} F\,M^{\sim n+1} &\equiv (F\,M^{\sim n})M \\ F\,M^{\sim 0} &\equiv F \end{aligned} \tag{2.4}$$

Inicialmente, estos abusos de notación pueden resultar confusos, sin embargo, al escribir términos lambda complejos resulta conveniente acortarlos. A continuación se muestran ejemplos de términos lambda asociados a términos sintácticamente equivalentes pero escritos con abuso de notación:

Ejemplo 2.1.3.

$$\begin{aligned} (((x\ y)z)(y\ x)) &\equiv x\ y\ z(y\ x) \\ (\lambda x.((u\ x)y)) &\equiv \lambda x.u\ x\ y \\ (\lambda y.(u(\lambda x.y))) &\equiv \lambda y.u(\lambda x.y) \\ (((\lambda y.((v\ u)u))z)y) &\equiv (\lambda y.v\ u\ u)z\ y \\ (((u\ x)(y\ z))(\lambda v.(v\ y))) &\equiv u\ x(y\ z)(\lambda v.v\ y) \\ ((((\lambda x.(\lambda y.(\lambda z.((x\ z)(y\ z)))))u)v)w) &\equiv (\lambda x\ y\ z.x\ z(y\ z))u\ v\ w \end{aligned}$$

2.1.1. Estructura

Dado un término lambda M es deseable poder cuantificar algunas propiedades de acuerdo a su estructura, la medida más común es la de *longitud*. Esta propiedad resulta importante en los razonamientos inductivos, por ejemplo, al plantear una demostración se suele usar la expresión "por inducción sobre M" la cual técnicamente se refiere a una inducción sobre la longitud de M.

Definición 2.1.3 (Longitud)**.** La longitud de un término lambda M, denotada como $\|M\|$, es la cantidad de *apariciones* de átomos en M, se determina a partir de la estructura del término lambda como:

$$\begin{aligned} \|x\| &= 1 \\ \|M\,N\| &= \|M\| + \|N\| \\ \|\lambda x.M\| &= 1 + \|M\| \end{aligned}$$

Debido a que la definición considera la cantidad de átomos en M y la longitud de un átomo es 1, se infiere que para cualquier término lambda M, su longitud será estrictamente mayor a cero. Una implicación de esta observación es que al "desbaratar" la longitud de un término lambda de acuerdo a su estructura, en el caso de que M sea una aplicación o una abstracción, la longitud de sus partes es estríctamente menor a su longitud.

Ejemplo 2.1.4. A continuación se presenta el procedimiento para calcular la longitud del término

$M \equiv (x(\lambda y.y\ u\ x))$ siguiendo la definición 2.1.3

$$\begin{aligned}
\|M\| &= \|(x(\lambda y.y\ u\ x))\| = \|(x(\lambda y.((y\ u)x)))\\
&= \|x\| + \|(\lambda y.((y\ u)x))\| = 1 + \|(\lambda y.((y\ u)x))\|\\
&= 1 + (1 + \|((y\ u)x)\|) = 2 + \|((y\ u)x)\|\\
&= 2 + \|(y\ u)\| + \|x\| = 2 + \|(y\ u)\| + 1 = 3 + \|(y\ u)\|\\
&= 3 + \|y\| + \|u\| = 3 + 1 + 1\\
&= 5
\end{aligned}$$

Una cuestión importante al momento de demostrar un teorema o definir un concepto por inducción sobre un término lambda es que usualmente la inducción matemática relaciona proposiciones con números naturales. Sin embargo es posible tener dos términos diferentes M y N tal que $\|M\| = \|N\|$, por ejemplo $(\lambda x.x)$ y $(z\ z)$ tienen longitud 2.

La inducción sobre la longitud de un término lambda considera también la estructura del término, de tal manera que para una proposición P sobre un término lambda M, los casos base de la inducción son aquellos en donde la estructura no es compuesta (en átomos cuya longitud siempre es 1) y la hipótesis de inducción considera que P se cumple para los subtérminos de M cuya longitud siempre es estrictamente menor que $\|M\|$.

En la definición de longitud se menciona de la cantidad de *apariciones* de átomos en M, el concepto de aparición de M en N para cualesquiera M y N se formaliza a partir del concepto de *subtérmino*.

Definición 2.1.4 (Subtérmino)**.** M es un subtérmino de N, denotado $M \subset N$ si $M \in \text{Sub}(N)$, donde $\text{Sub}(N)$ es la colección de subtérminos de N definida de manera inductiva como

$$\begin{aligned}
\text{Sub}(x) &= \{x\}\\
\text{Sub}(\lambda x.M) &= \text{Sub}(M) \cup \{\lambda x.M\}\\
\text{Sub}(M\ N) &= \text{Sub}(M) \cup \text{Sub}(N) \cup \{M\ N\}
\end{aligned}$$

Definición 2.1.5 (Aparición)**.** La aparición de M en N implica que $M \subset N$ o que M es *el* argumento de una abstracción en N.

Usualmente se habla de la aparición de M en N para referirse a una subtérmino en particular en N, sin embargo, un subtérmino pude *aparecer* varias veces en un término. Algunas clasificaciones de subtérminos son:

- Si M_1 y M_2 son subtérminos de N y no tienen átomos en común, se dice que son términos *disjuntos* de N, ya que si esta condición se cumple $\text{Sub}(M_1) \cap \text{Sub}(M_2) = \emptyset$;
- Si $M \subset N$ y $(M\ Z) \subset N$ se dice que M es un término *activo* en N, de lo contrario, se le llama *pasivo*;
- Si $M \subset N$ y $(\lambda x.M) \subset N$, se dice que la aparición M es el *alcance* de la aparición del átomo x que acompaña a la λ.

Ejemplo 2.1.5. Sea $M \equiv \lambda x.x\ y(\lambda z.y)$:

- el término $(x\ y) \subset M$;

- el átomo $z \not\subset M$ pero si aparece en M, debido a que z acompaña a una λ;
- el término $y(\lambda z.y)$ a pesar de parecer ser un subtérmino de M no lo es, esto se puede corroborar escribiendo los términos sin el abuso de notación: $y(\lambda z.y) \equiv (y(\lambda z.y))$ y $M \equiv \lambda x.xy(\lambda z.y) \equiv (\lambda x.((x\ y)(\lambda z.y)))$, en este caso, la clave está en observar la estructura de la aplicación $(x\ y(\lambda z.y))$.
- Las apariciones de x y $(\lambda z.y)$ en M son disjuntas.
- Los términos x y $(x\ y)$ son subtérminos activos de M, mientras que y y $(\lambda z.y)$ son subtérminos pasivos.

Los conceptos de longitud y de subtérmino nos permiten razonar de manera clara sobre la estructura de los términos lambda y con la clasificación de los subtérminos se puede caracterizar el rol que juegan las partes de un término en la estructura general.

2.1.2. Clasificación

A continuación se presentan algunos criterios para clasificar partes de los términos lambda y las propiedades que tienen los términos de acuerdo a su clasificación.

Al considerar las apariciones de átomos en un término lambda, es conveniente diferenciar a los átomos sintácticamente iguales dependiendo de el papel que juegan en el término.

Definición 2.1.6 (Clasificación de variables)**.** La aparición de un átomo x en un término P es llamada:

- *variable ligada* si es un subtérmino de M en una abstracción $(\lambda x.M)$ en P;
- *variable enlazada* si y sólo si es la x que acompaña la λ de $(\lambda x.M)$ en P;
- *variable libre* en otro caso.

La diferencia entre un átomo $x \subset M$ y una aparición de x en M es que la aparición se refiere a una variable en particular nombrada x en una parte específica de la estructura de M. Por ejemplo, en el término lambda $((\lambda x.x)x)$ la primera aparición del átomo x es una variable enlazada, la segunda aparición es una variable ligada y la tercera aparición es una variable libre.

Cuando se abordó el concepto de reducción en la sección 1.2.3 la distinción entre una variable libre y una ligada era importante ya que las variables libres nunca son sustituídas en una reducción ya que el procedimiento relacionaba únicamente a las variables ligadas en el alcance de una abstracción activa.

Ejemplo 2.1.6. Sea $M \equiv x(\lambda y.x\ y)$:

- El átomo x aparece como variable libre dos veces en M;
- El átomo y aparece como variable ligada en M;
- El átomo y aparece como la variable enlazada de la abstracción.

En la definición formal de algunos conceptos es conveniente hacer referencia a todas las variables libres de un término lambda.

Definición 2.1.7 (Variables libres)**.** El conjunto de variables libres de un término lambda M se denota $\mathrm{FV}(M)$ y se define de manera inductiva como:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) \setminus \{x\} \\ \mathrm{FV}(M\,N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{aligned}$$

Cuando $\mathrm{FV}(M) = \emptyset$ se dice que M es un *combinador* o *término cerrado*.

Ejemplo 2.1.7. Consideremos los términos $(x(\lambda x.x\ y\ z))$, $(\lambda x\ y\ z.y)$ y $((\lambda y.x)\lambda x.y)$.

- $\mathrm{FV}(x(\lambda x.x\ y\ z)) = \{x,\ y,\ z\}$;
- $\mathrm{FV}(\lambda x\ y\ z.y) = \emptyset$, por lo tanto es un combinador;
- $\mathrm{FV}((\lambda y.x)\lambda x.y) = \{x,\ y\}$.

En ocaciones es importante distinguir los términos lambda cerrados de aquellos que contienen variables libres, para ello se identifica el subconjunto de Λ que contiene a todos los términos cerrados:

Definición 2.1.8 (Términos cerrados)**.** Se denota como Λ^o al conjunto

$$\{M \in \Lambda \mid M \text{ es un término cerrado}\}$$

La notación Λ^o se puede generalizar para identificar diferentes subconjuntos de Λ a partir de las variables libres de los términos lambda:

$$\Lambda^o(\vec{x}) = \{M \in \Lambda \mid \mathrm{FV}(M) \subseteq \{\vec{x}\}\}$$

De tal manera que:

$$\Lambda^o = \Lambda^o(\emptyset)$$

Si consideramos un término M con variables libres, se puede encontrar otro término $N \in \Lambda^o$ similar a M, al cual se le llama clausura de M.

Definición 2.1.9 (Clausura)**.** La clausura de un término lambda M con $\mathrm{FV}(M) \neq \emptyset$ es un término lambda

$$(\lambda \vec{x}.M)$$

con $\vec{x} = \mathrm{FV}(M)$

Ejemplo 2.1.8. Sea $M \equiv \lambda z.x\ y\ z$

- $(\lambda x\ y.\lambda z.x\ y\ z)$ es una clausura de M;
- $(\lambda y\ x\ z.x\ y\ z)$ es una clausura de M;
- $(\lambda z\ x\ y.\lambda z.x\ y\ z)$ no es una clausura de M.

2.1.3. Sustitución de términos

Pendiente explicar el resto de la sección

Definición 2.1.10 (Sustitución)**.** Para cualesquiera términos lambda M, N y x, se define $M[x := N]$ como el resultado de sustituir cada aparición libre de x por N en M de acuerdo a las siguientes reglas:

$$
\begin{aligned}
x[x := N] &\equiv N; \\
a[x := N] &\equiv a && a \not\equiv x; \\
(P\,Q)[x := N] &\equiv P[x := N]\,Q[x := N]; \\
(\lambda x.P)[x := N] &\equiv \lambda x.P; \\
(\lambda y.P)[x := N] &\equiv \lambda y.P && x \not\equiv y,\ x \notin \mathrm{FV}(P); \\
(\lambda y.P)[x := N] &\equiv \lambda y.P[x := N] && x \not\equiv y,\ x \in \mathrm{FV}(P),\ y \notin \mathrm{FV}(N); \\
(\lambda y.P)[x := N] &\equiv \lambda z.P[y := z][x := N] && x \not\equiv y,\ x \in \mathrm{FV}(P),\ y \in \mathrm{FV}(N),\ z \notin \mathrm{FV}(NP).
\end{aligned}
$$

Ejemplo 2.1.9. Procedimientos de sustituciones para cada uno de los casos de la definición 2.1.10:

- Caso $x[x := N]$

$$y[y := \lambda x.x] \equiv \lambda x.x$$

- Caso $a[x := N]$, donde $a \not\equiv x$

$$z[w := x\,x] \equiv z$$

- Caso $(P\,Q)[x := N]$

$$
\begin{aligned}
(y\,x\,x)[x := y] &\equiv ((y\,x)x)[x := y] \\
&\equiv (y\,x)[x := y]\,x[x := y] \\
&\equiv (y[x := y]\,x[x := y])y \\
&\equiv y\,y\,y
\end{aligned}
$$

- Caso $(\lambda x.P)[x := N]$

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

- Caso $(\lambda y.P)[x := N]$, donde $x \not\equiv y$, $x \notin \mathrm{FV}(P)$

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

- Caso $(\lambda y.P)[x := N]$, donde $x \not\equiv y$, $x \in \mathrm{FV}(P)$, $y \notin \mathrm{FV}(N)$

$$\begin{aligned}(\lambda f.x\ \lambda x.f\ f\ x)[x := y] &\equiv \lambda f.(x\ \lambda x.f\ f\ x)[x := y] \\ &\equiv \lambda f.x[x := y]\,(\lambda x.f\ f\ x)[x := y] \\ &\equiv \lambda f.y\ \lambda x.f\ f\ x\end{aligned}$$

- Caso $(\lambda y.P)[x := N]$, donde $x \not\equiv y$, $x \in \mathrm{FV}(P)$, $y \in \mathrm{FV}(N)$ y $z \notin \mathrm{FV}(NP)$

$$\begin{aligned}(\lambda f.x\ \lambda x.f\ f\ x)[x := f] &\equiv \lambda g.(x\ \lambda x.f\ f\ x)[f := g][x := f] \\ &\equiv \lambda g.(x[f := g](\lambda x.f\ f\ x)[f := g])[x := f] \\ &\equiv \lambda g.(x\ \lambda x.(f\ f\ x)[f := g])[x := f] \\ &\equiv \lambda g.(x\ \lambda x.((f\ f)[f := g]x[f := g]))[x := f] \\ &\equiv \lambda g.(x\ \lambda x.((f[f := g]\ f[f := g])x))[x := f] \\ &\equiv \lambda g.(x\ \lambda x.g\ g\ x)[x := f] \\ &\equiv \lambda g.x[x := f](\lambda x.g\ g\ x)[x := f] \\ &\equiv \lambda g.f\ \lambda x.g\ g\ x\end{aligned}$$

En el último caso es importante observar que las apariciones ligadas de x no se sustituyen.

Lema 2.1.1. Si $(y\ x) \notin \mathrm{FV}(L)$ y $x \not\equiv y$, entonces

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

En contraste a la operación de sustitución en donde no se permite introducir o quitar referencias a variables enlazadas, el *contexto* es un término con "hoyos":

Definición 2.1.11 (Contexto)**.** Un contexto es un término lambda denotado $C[\ ]$ definido de manera inductiva:

- x es un contexto;
- $[\ ]$ es un contexto;
- Si $C_1[\ ]$ y $C_2[\ ]$ son contextos, entonces $C_1[\ ]\,C_2[\ ]$ y $\lambda x.C_1[\ ]$ también lo son.

Si $C[\ ]$ es un contexto y $M \in \Lambda$, entonces $C[M]$ denota el resultado de reemplazar por M los hoyos de $C[\ ]$. Al realizar esto, las variables libres de M pueden convertirse en variables ligadas de $C[M]$.

Ejemplo 2.1.10. Consideremos el contexto $C[\ ] \equiv \lambda x.x\ \lambda y.[\ ]$ y el término lambda $M \equiv (x\,y)$.

$$\begin{aligned}C[M] &\equiv (\lambda x.x\ \lambda y.[\ ])[(x\ y)] \\ &\equiv (\lambda x.x\ \lambda y.(x\ y))\end{aligned}$$

El caso análogo con la sustitución es

$$\begin{aligned}(\lambda x.x\ \lambda y.w)[w := (x\ y)] &\equiv \lambda z.(x\ \lambda y.w)[x := z][w := (x\ y)]\\ &\equiv \lambda z.(x[x := z](\lambda y.w)[x := z])[w := (x\ y)]\\ &\equiv \lambda z.(z\ \lambda y.w)[w := (xy)]\\ &\equiv \lambda z.z[w := (x\ y)](\lambda y.w)[w := (x\ y)]\\ &\equiv \lambda z.z\ \lambda v.w[w := (x\ y)]\\ &\equiv \lambda z.z\ \lambda v.(x\ y)\end{aligned}$$

2.2. Los cálculos de la conversión lambda

El objetivo principal de esta subsección es presentar una formalización del cálculo lambda descrito en el capítulo 1 desde el punto de vista de *teorías formales*. El nombre técnico de la teoría formal principal de este trabajo es $\boldsymbol{\lambda K\beta}$, se pueden realizar modificaciones y extensiones a esta teoría y los siguientes conceptos permiten estudiar las implicaciones de estos cambios.

2.2.1. Teorías formales

Una *teoría formal* $\mathcal{T}$ es una tripleta $(\mathcal{F}, \mathcal{A}, \mathcal{R})$ donde

- $\mathcal{F}$ es el conjunto de todas las *fórmulas* $X = Y$ con X y Y elementos de un lenguaje formal;
- $\mathcal{A}$ es un conjunto de *axiomas* y $\mathcal{A} \subseteq \mathcal{F}$;
- $\mathcal{R}$ es un conjunto de *reglas*.

Una regla es una función $\phi\colon \mathcal{F}^n \to \mathcal{F}$ con $n \geq 1$. Si se consideran n fórmulas $A_1, \ldots, A_n$ tal que

$$\phi(A_1, \ldots, A_n) = B$$

Se dice que la secuencia $\langle A_1, \ldots, A_n, B\rangle$ es una *instancia* de la regla ϕ. Las primeras n fórmulas de una instancia son llamadas *premisas* y la última fórmula es llamada *conclusión*. Para escribir una instancia de una regla se utiliza la notación

$$\frac{A_1 \quad \ldots \quad A_n}{B}$$

Observación. En la literatura se pueden encuentrar diferentes maneras de trabajar con teorías formales, dependiendo de su "estilo" y definición, por ejemplo en [1] las reglas se definen como conjuntos de secuencias $\langle A_1, \ldots, A_{n+1}\rangle$ con n premisas y una conclusión, en donde los axiomas se definen como elementos de $\mathcal{R}$ con cero premisas. La definición de teoría formal presentada en este trabajo es del estilo Hilbert y está basada en [13, pp. 69–70].

Si consideramos un conjunto de *suposiciones* $\Gamma \subseteq \mathcal{F}$, una *deducción* de una fórmula B desde Γ es un árbol dirigido de fórmulas en donde los vértices de un extremo son elementos de $\mathcal{A}$ o Γ, los vértices intermedios son deducidos a partir de los vértices que inciden en ellos a partir de una regla y el vértice

de el otro extremo siendo B. Si y solo si existe una deducción para una fórmula B, se dice que B es *demostrable* en $\mathcal{T}$ suponiendo Γ, denotado

$$\mathcal{T}, \Gamma \vdash B$$

En caso que la deducción no tenga suposiciones, se dice que es una *demostración* y que B es un *teorema*. Cuando una deducción no tiene suposiciones, es decir, $\Gamma = \emptyset$ se denota

$$\mathcal{T} \vdash B$$

La relación binaria $=$ en las fórmulas de una teoría es una relación de equivalencia, la cual por definición es *reflexiva*, *simétrica* y *transitiva*. La *clase de equivalencia* de un objeto x con respecto a $=$ de una teoría formal $\mathcal{T}$, denotado $[x]_{\mathcal{T}}$, es el conjunto de todos los objetos y tal que $x = y$ es una fórmula de $\mathcal{T}$.

En el contexto de las teorias que formalizan los cálculos lambda, los objetos que se relacionan son términos lambda. La frase "módulo convertibilidad" se refiere al conjunto de todas las clases de equivalencia de Λ considerando la relación de equivalencia de la teoría formal con la que se esté trabajando.

Que este conjunto sea el objeto de estudio de una teoría $\boldsymbol{\lambda}$ del cálculo lambda significa que cada elemento de Λ módulo convertibilidad, denotado $\Lambda/=_{\lambda}$, es distinto y representa una clase de términos lambda considerados en la teoría $\boldsymbol{\lambda}$ como equivalentes. Cuando $\boldsymbol{\lambda} \vdash M = N$ se dice que M y N son términos *convertibles*, también denotado $M =_{\lambda} N$.

Habiendo definido una teoría $\boldsymbol{\lambda}$, el interés de estudiarla es

- determinar los términos que son convertibles en $\boldsymbol{\lambda}$;
- estudiar las propiedades que comparten dos términos convertibles;
- modificar a $\boldsymbol{\lambda}$ y comparar la teoría modificada con la original.

La comparación entre teorías usualmente consiste en partir de una teoría $\boldsymbol{\lambda}$, modificar sus fórmulas, axiomas o reglas para obtener otra teoría $\boldsymbol{\lambda}'$ y determinar si $\boldsymbol{\lambda}$ y $\boldsymbol{\lambda}'$ son equivalentes. Para poder realizar esto, se debe definir formalmente cómo se modifica una teoría y cómo se demuestra que dos teorías son equivalentes.

Modificar una teoría $\boldsymbol{\lambda}$ puede implicar cambiar la definición de sus términos, es decir, utilizar un lenguaje formal diferente al de $\boldsymbol{\lambda}$ para expresar las fórmulas $M = N$. Hacer cambios al lenguaje formal suele requerir modificar al conjunto $\mathcal{F}$, $\mathcal{A}$ y $\mathcal{R}$ de la teoría.

La modificación al lenguaje formal puede ser únicamente de relevancia sintáctica, por ejemplo modificar una teoría $\boldsymbol{\lambda}$ cuyo lenguaje de términos es el conjunto Λ^{o} para que los términos sean escritos con el índice de DeBruijn mostrado en 1.3.1 no tendría implicaciones fuertes en la convertibilidad de la teoría, ya que hay una correspondencia uno a uno entre estas dos notaciones. Por otro lado, modificar una teoría $\boldsymbol{\lambda}$ con términos Λ de tal manera que se consideren únicamente los términos cerrados Λ^{o} si puede tener fuertes implicaciones en la convertibilidad de la teoría ya que habrá términos lambda no admitidos en fórmulas.

Otra manera de modificar una teoría $\boldsymbol{\lambda}$ es añadir o quitar axiomas y reglas de inferencia. Para abordar la modificación de estas dos componentes de una teoría consideramos que los axiomas son reglas sin premisas.

Cuando se considera extender una teoría λ con una nueva regla ϕ lo primero que se debe estudiar es si ϕ es *derivable* en λ, es decir, si para cada instancia de ϕ, su conclusión es deducible en λ considerando sus premisas como suposiciones. Formalmente, para cada instancia $\langle A_1, \ldots, A_n, B\rangle$ de ϕ, ϕ es derivable en λ si y sólo si

$$\lambda, \{A_1, \ldots, A_n\} \vdash B \tag{2.5}$$

Cuando añadir una regla ϕ a una teoría λ no cambia el conjunto de teoremas se dice que la regla es *admisible*, por ejemplo si ϕ es utilizada en la demostración de un teorema, pero este teorema se puede demostrar sin suponer las premisas de ϕ, entonces añadir a ϕ no afecta el hecho de que el teorema exista en la teoría.

Otra manera de verificar si una regla ϕ es admisible en λ es demostrando que la regla es *correcta*. Una regla se dice ser correcta en una teoría λ si y sólo si, para cada instancia $\langle A_1, \ldots, A_n, B\rangle$ de ϕ:

$$(\lambda \vdash A_1), \ldots, (\lambda \vdash A_n) \implies (\lambda \vdash B) \tag{2.6}$$

Si una regla es derivable, entonces es admisible, sin embargo, una regla admisible no siempre es derivable. Consideremos una instancia r de una regla admisible en λ tal que ni las premisas, ni la conclusión de r son demostrables en la teoría, entonces la implicación (2.6) es verdadera para r, sin embargo esto no implica que se pueda demostrar la conclusión suponiendo las premisas. Por otro lado, si consideramos una instancia r de una regla derivable en λ, entonces ya que la conclusión es demostrable suponiendo las premisas, demostrar las premisas asegura que se puede demostrar la conclusión.

Con estos conceptos se pueden definir dos criterios de equivalencia entre teorías: *equivalentes en teoremas* y *equivalentes en reglas*.

Definición 2.2.1 (Equivalencia de teorías)**.** Sean λ y λ' dos teorías formales con el mismo conjunto de fórmulas.

Se dice que las teorías son *equivalentes en teoremas* cuando cada regla y axioma de λ es admisible en λ' y viceversa.

Se dice que las teorías son *equivalentes en reglas* cuando cada regla y axioma de λ es derivable en λ' y viceversa.

La equivalencia en teoremas es una equivalencia más débil que la equivalencia en reglas.

2.2.2. Teoría $\lambda K\beta$

La teoría $\lambda K\beta$ es la formalización del cálculo lambda que se ha tratado desde el inicio de este trabajo. Ya que es la teoría principal, a partir de este punto cuando se hable de *la* teoría λ se estará refiriendo a la teoría $\lambda K\beta$ y cuando se hable de *las* teorías λ se estará refiriendo a la familia de teorías que formalicen los cálculos lambda.

Definición 2.2.2 (Teoría $\lambda K\beta$)**.** El conjunto de fórmulas $\mathcal{F}$ en $\lambda K\beta$ tiene como elementos ecuaciones de la forma:

$$M = N \qquad\qquad \forall M, N \in \Lambda \text{ (de la definición 2.1.1)}$$

Los axiomas $\mathscr{A}$ de $\boldsymbol{\lambda K\beta}$ son:

$$\lambda x.M = \lambda y.M[x := y] \qquad \forall y \notin \mathrm{FV}(M) \tag{α}$$

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

$$M = M \tag{ρ}$$

Las reglas $\mathscr{R}$ de $\boldsymbol{\lambda K\beta}$ son:

$$\frac{M = N}{Z\,M = Z\,N} \tag{μ}$$

$$\frac{M = N}{M\,Z = N\,Z} \tag{ν}$$

$$\frac{M = N}{\lambda x.M = \lambda x.N} \tag{ξ}$$

$$\frac{M = N \quad N = P}{M = P} \tag{τ}$$

$$\frac{M = N}{N = M} \tag{σ}$$

Consideremos la convertibilidad en $\boldsymbol{\lambda K\beta}$ de los términos lambda

$$\begin{aligned} M &\equiv (\lambda f.x((\lambda y.y\,f)\lambda z.z))w \\ N &\equiv x\,w \end{aligned}$$

Se demuestra que $M =_{\lambda K\beta} N$ construyendo un árbol de deducción como el de la figura 2.1.

2.2.3. Combinadores *SKI*

Definición 2.2.3 (Combinadores SKI)**.** Tres términos lambda de suma importancia son

$$\begin{aligned} \boldsymbol{I} &\equiv \lambda x.x \\ \boldsymbol{K} &\equiv \lambda x\,y.x \\ \boldsymbol{S} &\equiv \lambda x\,y\,z.x\,z(y\,z) \end{aligned}$$

Corolario. Para todo término $M, N, L \in \Lambda$

$$\begin{aligned} \boldsymbol{I}\,M &=_\lambda M \\ \boldsymbol{K}\,M\,N &=_\lambda M \\ \boldsymbol{S}\,M\,N\,L &=_\lambda M\,L(N\,L) \end{aligned}$$

Estos tres combinadores generan en la teoría $\boldsymbol{\lambda}$ al conjunto Λ^o con combinaciones de aplicaciones. Debido a que $\boldsymbol{SKK} =_\lambda \boldsymbol{I}$, sólo es necesario combinar con aplicaciones a $\boldsymbol{K}$ y a $\boldsymbol{S}$ para generar cualquier término cerrado.

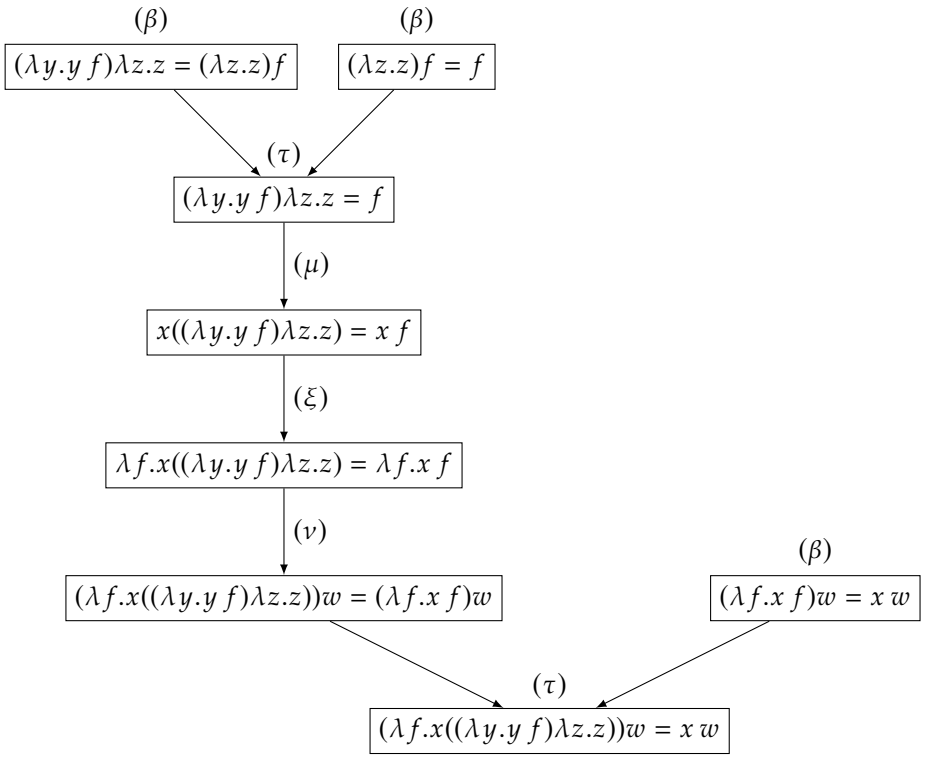


Figura 2.1: Árbol de deducción para demostrar la convertibilidad entre dos términos

Definición 2.2.4 (Bases). 1. Sea $\mathfrak{X} \subset \Lambda$. El conjunto de términos *generado* por $\mathfrak{X}$, denotado $\mathfrak{X}^+$, es el conjunto mas pequeño tal que

   a) $\mathfrak{X} \subseteq \mathfrak{X}^+$,

   b) $M, N \in \mathfrak{X}^+ \implies (MN) \in \mathfrak{X}^+$.

2. Sea $\mathcal{P} \subset \Lambda$. $\mathfrak{X} \subset \Lambda$ es una *base* para $\mathcal{P}$ si para toda $M \in \mathcal{P}$ existe $N \in \mathfrak{X}^+$ tal que $N = M$.

3. $\mathfrak{X}$ es llamada una *base* si $\mathfrak{X}$ es una base para Λ^o.

Lema 2.2.1. Sea $\lambda x.M$ una abstracción tal que Sub(M) no contiene abstracciones

1. Si $M = x$, entonces $\lambda x.M = \boldsymbol{I}$;

2. Si $x \notin \mathrm{FV}(M)$, entonces $\lambda x.M = (\boldsymbol{K}M)$;

3. Si $M = P\,Q$, entonces $\lambda x.M = \boldsymbol{S}(\lambda x.P)(\lambda x.Q)$.

Demostración. Ver 2.2.3 y 2.2.3

1.

$$\begin{aligned}(\boldsymbol{I}\,N) &= N \\ &= ((\lambda x.x)N) \\ &= ((\lambda x.M)N)\end{aligned}$$

2.

$$\begin{aligned}((\boldsymbol{K}\,M)N) &= (\lambda x\, y.x)M\,N \\ &= (\lambda y.M)N \\ &= ((\lambda x.M)N)\end{aligned}$$

3.

$$\begin{aligned}\boldsymbol{S}(\lambda x.P)(\lambda x.Q) &= (\lambda abc.(a\,c)(b\,c))(\lambda x.P)(\lambda x.Q) \\ &= \lambda c.((\lambda x.P)c)((\lambda x.Q)c) \\ &= \lambda c.P[x := c]Q[x := c] \\ &= \lambda c.(P\,Q)[x := c] \\ &= \lambda x.P\,Q \\ &= \lambda x.M\end{aligned}$$

□

Proposición 2.2.2. $\{\boldsymbol{S}, \boldsymbol{K}, \boldsymbol{I}\}$ es una base, es decir, para todo término $M \in \Lambda^o$, existe un término M' compuesto de aplicaciones de $\boldsymbol{S}$, $\boldsymbol{K}$ e $\boldsymbol{I}$ tal que $M = M'$.

La demostración de la proposición 2.2.2 consiste en la construcción de un algoritmo para transformar M a M'.

Demostración. Sea $M \in \Lambda^o$, se construye un término $M' \in \{\boldsymbol{S}, \boldsymbol{K}, \boldsymbol{I}\}^+$ tal que $M' = M$ enumerando los subtérminos en M que sean abstracciones de menor a mayor longitud.

Sea $\lambda x.N$ la abstracción con menor longitud en M, según la estructura de N se aplican las siguientes transformaciones:

1. Si $N = a$
   - *a*) Si $a = x$ se transforma $\lambda x.N$ a $\boldsymbol{I}$ en M.
   - *b*) Si $a \neq x$ se transforma $\lambda x.N$ a $(\boldsymbol{K}\,a)$ en M.
2. $N = (P\,Q)$
   - *a*) Si $x \notin \text{FV}(P)$ y $x \notin \text{FV}(Q)$ se transforma $\lambda x.P\,Q$ a $\boldsymbol{S}(\boldsymbol{K}\,P)(\boldsymbol{K}\,Q)$ en M.
   - *b*) Si $x \notin \text{FV}(P)$ y $x \in \text{FV}(Q)$ se transforma $\lambda x.P\,Q$ a $\boldsymbol{S}(\boldsymbol{K}\,P)(\lambda x.Q)$ en M.
   - *c*) Si $x \in \text{FV}(P)$ y $x \notin \text{FV}(Q)$ se transforma $\lambda x.P\,Q$ a $\boldsymbol{S}(\lambda x.P)(\boldsymbol{K}\,Q)$ en M.
   - *d*) Si $x \in \text{FV}(P)$ y $x \in \text{FV}(Q)$ se transforma $\lambda x.P\,Q$ a $\boldsymbol{S}(\lambda x.P)(\lambda x.Q)$ en M.

En los casos 2*a*, 2*b*, 2*c*, 2*d* se forman abstracciones con longitud menor a $\lambda x.N$, por lo tanto serán las que se transformarán después. Ya que la longitud de estas abstracciones es estrictamente menor a $\lambda x.N$ y los casos base 1*a* y 1*b* de la transformación no introducen abstracciones, en una cantidad finita de pasos el término M transformado no tendrá abstracciones de la forma $\lambda x.N$.

Para un término M con sólo una abstracción, a aplicaciones y v variables ligadas (no enlazadas) una cota superior para la máxima cantidad de pasos se calcula considerando que para los términos de la forma $\lambda x.a$ se cumple el caso 1*b* (el cual aumenta la cantidad de aplicaciones en 1) y que para los términos de la forma $\lambda x.P\,Q$ se cumple el caso 2*d* (el cual aumenta la cantidad de aplicaciones en 2) ya que en estos casos se produce el término con mas aplicaciones, las cuales determinan la cantidad de veces que se repite el algoritmo por cada abstracción en M. Para calcular la cota superior de la cantidad de aplicaciones a' que produce el algoritmo para un término con n abstracciones se plantea la siguiente relación de recurrencia:

$$\begin{aligned} a'_0 &= a \\ a'_n &= 2 \times a'_{n-1} + v \end{aligned}$$

Esta recurrencia describe la función $a' : \mathbb{N} \to \mathbb{N}$:

$$a'(n) = 2^n \times a + (2^n - 1) \times v$$

Para la cota superior de la cantidad de pasos realizados por el algoritmo para un término M con n abstracciones, se plantea la siguiente relación de recurrencia basada en a' y en el hecho de que la cantidad de variables ligadas y no enlazadas no aumenta en los pasos del algoritmo:

$$\begin{aligned} p_0 &= 0 \\ p_n &= p_{n-1} + a'(n-1) + v \end{aligned}$$

Esta recurrencia describe la función $p : \mathbb{N} \to \mathbb{N}$:

$$\begin{aligned} p(n) &= (a+v) \times \sum_{i=0}^{n-1} 2^i \\ &= (a+v) \times (2^n - 1) \end{aligned}$$

□

Algoritmo para compilar $\Lambda^0 \mapsto \{S,\ K,\ I\}^+$

2.2.4. Teoría $\lambda I\beta$

En el artículo [9], Alonzo Church presenta una definición del cálculo lambda con un conjunto restringido de términos lambda. A la teoría que considera a este conjunto restringido de términos lambda (denotado Λ_I) y los axiomas y reglas de inferencia de la teoría $\boldsymbol{\lambda}$ cambiando Λ por Λ_I se le conoce como teoría $\boldsymbol{\lambda I\beta}$ (o el cálculo λI).

Algoritmo 2.1 SKI

Entrada: $M \in \Lambda^o$
Salida: $M' \in \{\boldsymbol{S},\ \boldsymbol{K},\ \boldsymbol{I}\}$
  $M' \leftarrow M$
  $\mathscr{L} \leftarrow \{A \in \text{Sub}(M') \mid A \ \equiv\ \lambda x.N\}$
  mientras $\mathscr{L} \neq \emptyset$ **hacer**
    $A \leftarrow \lambda x.N \in \mathscr{L} \mid \|\lambda x.N\| \leq A', \forall A' \in \mathscr{L}$
    si $A \ \equiv\ \lambda x.a$ **entonces**
      si $a \ \equiv\ x$ **entonces**
        $M'[A] \leftarrow \boldsymbol{I}$
      de lo contrario y **si** $a \not\equiv\ x$ **entonces**
        $M'[A] \leftarrow \boldsymbol{K}\, a$
      fin si
    de lo contrario y **si** $A \ \equiv\ \lambda x.P\, Q$ **entonces**
      si $x \notin \text{FV}(P) \wedge x \notin \text{FV}(Q)$ **entonces**
        $M'[A] \leftarrow \boldsymbol{S}(\boldsymbol{K}\, P)(\boldsymbol{K}\, Q)$
      de lo contrario y **si** $x \notin \text{FV}(P) \wedge x \in \text{FV}(Q)$ **entonces**
        $M'[A] \leftarrow \boldsymbol{S}(\boldsymbol{K}\, P)(\lambda x.Q)$
      de lo contrario y **si** $x \in \text{FV}(P) \wedge x \notin \text{FV}(Q)$ **entonces**
        $M'[A] \leftarrow \boldsymbol{S}(\lambda x.P)(\boldsymbol{K}\, Q)$
      de lo contrario y **si** $x \in FV(P) \wedge x \in \text{FV}(Q)$ **entonces**
        $M'[A] \leftarrow \boldsymbol{S}(\lambda x.P)(\lambda x.Q)$
      fin si
      $M'[A] \leftarrow \text{SKI}(\ M'[A]\ )$
    fin si
    $\mathscr{L} \leftarrow \mathscr{L} \setminus \{A\}$
  fin mientras
  regresar M'

Definición 2.2.5 (Términos en Λ_I)**.**

$$\begin{aligned} x \in V &\implies x \in \Lambda_I \\ M \in \Lambda_I,\ x \in \mathrm{FV}\, M &\implies \lambda x.M \in \Lambda_I \\ M, N \in \Lambda_I &\implies M\, N \in \Lambda_I \end{aligned}$$

La diferencia fundamental entre las teorías $\boldsymbol{\lambda K\beta}$ y $\boldsymbol{\lambda I\beta}$ es el término lambda $\boldsymbol{K}$, ya que $\boldsymbol{K} \in \Lambda \setminus \Lambda_I$ pero $\boldsymbol{K} \notin \Lambda_I$. Esto es debido a que el subtérmino $\lambda y.x$ en $\boldsymbol{K}$ de la definición 2.2.3 no puede existir en Λ_I debido a que $y \notin \mathrm{FV}(x)$.

2.2.5. Extensionalidad

El concepto de igualdad de funciones usado en la mayoría de las ramas de la matemática es lo que se conoce como "extensional", esta propiedad de las relaciones de equivalencia hace referencia a las características externas de los objetos que compara, en el caso de las funciones, se incluye la suposición de que para funciones f y g con el mismo dominio

$$\forall x[f(x) = g(x)] \implies f = g$$

Contraria a esta suposición, en la computación, el tema central son los procedimientos y procesos que describen los programas o algoritmos, cuyas igualdades "intensional", es decir, si dos programas computan la misma función matemática, no necesariamente se dice que son el mismo programa ya que uno pudiera ser mas eficiente que otro (la característica de eficiencia es interna a cada algorítmo).

La teoría $\boldsymbol{\lambda}$ también es intensional: existen dos términos lambda F y G tales que para tódo término X

$$\boldsymbol{\lambda} \vdash F\, X = G\, X$$

Pero no $\boldsymbol{\lambda} \vdash F = G$. Por ejemplo, $F \equiv y$ y $G \equiv \lambda x.y\, x$

Cuando se plantea formalizar un cálculo lambda que sea extensional, surge la pregunta, ¿Qué es demostrable en el sistema extensional que no es demostrable en $\boldsymbol{\lambda}$. A continuación se presentan tres diferentes agregados a la teoría $\boldsymbol{\lambda}$ las cuales incluyen la propiedad de extensionalidad y que han sido propuestas en la literatura [13, 5]. Las teorías extendidas son llamadas $\boldsymbol{\lambda\zeta}$, $\boldsymbol{\lambda + ext}$ y $\boldsymbol{\lambda\eta}$ de acuerdo a la regla que se añade a la definición 2.2.2.

Definición 2.2.6 (Reglas de extensionalidad)**.** Cada una de las siguientes reglas nos permite añadir a $\boldsymbol{\lambda}$ la propiedad de extensionalidad.

Reglas de inferencia

$$\frac{M\, x = N\, x}{M = N} \qquad \text{si } x \notin \mathrm{FV}(M\, N) \qquad (\zeta)$$

$$\frac{M\, P = N\, P}{M = N} \qquad \forall P \in \Lambda \qquad (\mathrm{ext})$$

Axiomas

$$\lambda x.M\, x = M \qquad \text{si } x \notin \mathrm{FV}(M) \qquad (\eta)$$

La regla (ζ) dice, de manera informal, que si M y N tienen el mismo efecto sobre un objeto no especificado x, entonces $M = N$. La regla (ext) tiene una infinidad de premisas, una por cada término lambda P, por lo tanto, las deducciones en donde se involucre esta regla serán árboles infinitos.

Pendiente Comparación entre teorías formales, énfasis en η contra ζ contra ext.

2.3. Teoría de reducción

1.14 y 2.9

2.3.1. Contracciones

Transformaciones de términos con un paso.

Definición 2.3.1 (Contracciones Hindley y Seldin)**.** Dado un término lambda X, una *contracción* en X es una tripleta $\langle X, R, Y\rangle$, denotada $X \to_R Y$, donde R es una aparición de un *redex* en X y Y es el resultado de contraer R en X.

Ejemplo 2.3.1.

$$(\lambda x.(\lambda y.y\ x)z)v \to_{(\lambda x.(\lambda y.y\,x)z)v} (\lambda y.y\ v)z,$$
$$(\lambda x.(\lambda y.y\ x)z)v \to_{(\lambda y.y\,x)z} (\lambda x.z\ x)v.$$

2.3.2. Reducciones

Reducciones basadas en contracciones, de Barendregt

Definición 2.3.2 (Relación compatible)**.** Si dice que una relación binaria $\boldsymbol{R}$ sobre Λ es:

1. Una *relación compatible* cuando

$$(M, M') \in \boldsymbol{R} \implies (Z\,M, Z\,M') \in \boldsymbol{R},\ (M\,Z, M'\,Z) \in \boldsymbol{R},\ (\lambda x.M, \lambda x.M') \in \boldsymbol{R}$$

   para toda $M, M', Z \in \Lambda$.

2. Una *relación de congruencia* cuando $\boldsymbol{R}$ es compatible, reflexiva, transitiva y simétrica.

3. Una *relación de reducción* cuando $\boldsymbol{R}$ es compatible, reflexiva y transitiva.

compatible cuando

Nota. Una relación $\boldsymbol{R} \subseteq \Lambda^2$ es compatible cuando

$$(M, M') \in \boldsymbol{R} \implies (C[M], C[M']) \in \boldsymbol{R}$$

para toda $M, M' \in \Lambda$ y todo contexto $C[\ \ ]$, con un hoyo.

Definición 2.3.3. Una *noción de reducción* en Λ es una relación binaria $\boldsymbol{R}$ en Λ.

Sean $\boldsymbol{R}_1$ y $\boldsymbol{R}_2$ nociones de reducción, la relación $\boldsymbol{R}_1 \cup \boldsymbol{R}_2$ se denota $\boldsymbol{R}_1\boldsymbol{R}_2$.

Definición 2.3.4 (Reducción $\boldsymbol{\beta}$)**.** La regla (β) en la teoría $\boldsymbol{\lambda}$ se puede definir como la reducción:

$$\boldsymbol{\beta} = \{((\lambda x.M)N, M[x := N]) : M, N \in \Lambda\}$$

Definición 2.3.5. Sea $\boldsymbol{R}$ una noción de reducción en Λ, $\boldsymbol{R}$ introduce las relaciones binarias:

- R-reducción en un paso, denotada $\to_{\mathrm{R}}$ y definida de manera inductiva como:

$$\begin{aligned}
&(1) && (M, N) \in \boldsymbol{R} \implies M \to_{\mathrm{R}} N \\
&(2) && M \to_{\mathrm{R}} N \implies Z\,M \to_{\mathrm{R}} Z\,N \\
&(3) && M \to_{\mathrm{R}} N \implies M\,Z \to_{\mathrm{R}} N\,Z \\
&(4) && M \to_{\mathrm{R}} N \implies \lambda x.M \to_{\mathrm{R}} \lambda x.N
\end{aligned}$$

- R-reducción, denotada $\twoheadrightarrow_{\mathrm{R}}$ y definida de manera inductiva como:

$$\begin{aligned}
&(1) && M \to_{\mathrm{R}} N \implies M \twoheadrightarrow_{\mathrm{R}} N \\
&(2) && M \twoheadrightarrow_{\mathrm{R}} M \\
&(3) && M \twoheadrightarrow_{\mathrm{R}} N,\ N \twoheadrightarrow_{\mathrm{R}} L \implies M \twoheadrightarrow_{\mathrm{R}} L
\end{aligned}$$

- R-convertibilidad, denotada $=_{\mathrm{R}}$ y definida de manera inductiva como:

$$\begin{aligned}
&(1) && M \twoheadrightarrow_{\mathrm{R}} N \implies M =_{\mathrm{R}} N \\
&(2) && M =_{\mathrm{R}} N \implies N =_{\mathrm{R}} M \\
&(3) && M =_{\mathrm{R}} N,\ N =_{\mathrm{R}} L \implies M =_{\mathrm{R}} L
\end{aligned}$$

Lema 2.3.1. Las relaciones $\to_{\mathrm{R}}$, $\twoheadrightarrow_{\mathrm{R}}$ y $=_{\mathrm{R}}$ son compatibles. Por lo tanto $\twoheadrightarrow_{\mathrm{R}}$ es una relación de reducción y $=_{\mathrm{R}}$ es una relación de congruencia.

Demostración. □

Usualmente una noción de reducción se introduce de la siguiente manera: "Sea $\boldsymbol{R}$ definida por las siguientes *reglas de contracción* $\boldsymbol{R}$: $M \to N$ dado que ...".

Esto significa que $\boldsymbol{R} = \{(M, N) : ...\}$, por ejemplo, $\boldsymbol{\beta}$ pudo haber sido introducida por la siguiente regla de contracción

$$\boldsymbol{\beta} : (\lambda x.M)N \to M[x := N]$$

Definición 2.3.6.

1. Un R-redex es un término M tal que $(M, N) \in R$ para algún término N. En este caso N es llamado un R-contractum de M.

2. Un término M es llamado una forma normal de R, denotado $R - fn$, si M no contiene algún R-redex.

3. Un término N es una $R - fn$ de M (o M tiene la $R - fn$ N) si N es una $R - fn$ y $M =_{\mathrm{R}} N$.

El proceso de pasar de un redex a su contractum es llamado *contracción*. En lugar de escribir "M es una $R - fn$" usualmente se escribe "M está en $R - fn$", pensando en una máquina que ha llegado a su estado final.

Ejemplo 2.3.2. $(\lambda x.x\,x)\lambda y.y$ es un β-redex. Por lo tanto $(\lambda x.x\,x)(\lambda y.y)z$ no está en $\beta - fn$; sin embargo este término tiene la $\beta - fn\ z$.

Plantear las ideas para pasar del concepto de contracción al de reducción y posteriormente al de convertibilidad.

Gráficas de reducción.

Teorema de Church-Rosser y toda la magia necesaria para abordarlo y la magia que valga la pena mencionar en la que CR es importante va aquí.

Probablemente es buena idea separar la sección "Noción informal del cálculo lambda" y "Formalización del cálculo lambda" en dos capítulos diferentes. ¿Valdrá la pena abordar árboles de Böhm?

Capítulo 3

Codificación de objetos

3.1. Álgebra Booleana

El álgebra booleana es una rama del álgebra en donde las expresiones tienen asociado un valor de *falso* o *verdadero*. Estas expresiones son fundamentales en el estudio de circuitos y programas escritos en lenguajes de programación.

Los términos lambda no tienen asignados un valor de verdad y las operaciones que se plantearon en los primeros dos capítulos involucraron el concepto de falso y verdadero únicamente en el metalenguaje y asociando estos valores no a los términos lambda en sí, si no a propiedades de estos, por ejemplo, es falso que $\|\lambda x.x\| = 5$ y es verdadero que $(\boldsymbol{K}\, x) \twoheadrightarrow_\beta (\lambda x.y)$. Sin embargo es posible codificar los valores de verdad como elementos de Λ y construir abstracciones que emulen las propiedades de las operaciones booleanas bajo la β-reducción. De esta manera se pueden escribir términos que, de acuerdo con la codificación establecida, representen expresiones booleanas y términos lambda al mismo tiempo.

En los lenguajes de programación usualmente se mezclan las expresiones booleanas con otras expresiones y objetos a partir de *predicados*, éstos son funciones con algún dominio X y codominio $\{\text{falso},\ \text{verdadero}\}$. Por ejemplo, al escribir un programa en donde se necesite tomar una desición a partir de si un número n es positivo o negativo se escribiría (en pseudocódigo):

si esPositivo(n) **entonces**
  ...
de lo contrario
  ...
fin si

En este ejemplo «esPositivo» es un predicado que es evaluado a falso si `n` no es positivo y a verdadero si lo es.

La codificación de valores de verdad y operaciones booleanas es común incluso en lenguajes de programación populares, por ejemplo en C, el tipo `bool` es codificado como un entero, en donde falso es `0` y verdadero cualquier otro entero, a su vez, los enteros son codificados usualmente como secuencias de 32 bits en complemento a dos. Por lo tanto, si «esPositivo» fuera una función de C: «esPositivo(8)» sería evaluado a `1` y «esPositivo(-8)» sería evaluado a `0`.

Al igual que el cálculo lambda, otras teorías que fundamentan las ciencias de la computación también carecen de expresiones y operaciones booleanas. En el caso de la máquina de Turing los cambios de estado

en la ejecución de un programa se determinan a partir de su función de transición y predicados simples de igualdad entre símbolos del alfabeto de cinta se realizan en un paso, sin embargo, predicados mas complejos requieren ser codificados con estados, transiciones y anotaciones en su cinta.

3.1.1. Valores de verdad

En el álgebra booleana, los valores de las expresiones son falso y verdadero. El nombre de estos valores no es de relevancia y usualmente falso se representa como 0 y verdadero como 1. El aspecto importante de estos valores es que son distintos y si un valor x no es uno, entonces es el otro.

Revisar Me parece que suena confusa la analogía, el objetivo es plantear la idea de no codificar falso y verdadero como un valor en particular, si no como una desición.

Podemos ignorar la representación concreta de estos valores y pensar en una situación hipotética: Una persona omnisciente y muda llamada P puede decirme si una oración que le digo es falsa o verdadera dándole una manzana y una pera; si me regresa la manzana significa que la oración es verdadera y si me regresa la pera significa que la oración es falsa. En este planteamiento irreal e hipotético, no fué necesario conocer la estructura de la verdad y la falsedad, solo fué necesario tener a alguien que tomara una desición (en este caso P) y proveer dos objetos que podemos distinguir entre sí (en este caso la manzana y la pera). Las desiciones de esta persona pueden ser los conceptos de falso y verdadero si nunca podemos conocer los valores booleanos.

Detrás del concepto de falso y verdadero, está el concepto de *desición*, la codificación que se desarrolla está basada en este concepto y aparece en [5, p. 133].

Supongamos que P es un término lambda el cual puede ser aplicado a una oración O, al β-reducir $(P\ O)$ se obtiene una decisión D la cual al ser aplicada a dos términos lambda M y N se β-reduce a M si la oración O es verdadera y a M si es falsa:

$$P\ O \twoheadrightarrow_\beta D,$$

$$D\ M\ N \twoheadrightarrow_\beta \begin{cases} M & \text{si } O \text{ es verdadera} \\ N & \text{si } O \text{ es falsa} \end{cases}.$$

Para fines prácticos no es necesario saber cómo es P ni O, lo importante es que cuando O es cierta, D eligirá M y si O es falsa, eligirá N. Por lo tanto, $(P\ O) = D$ es un término lambda de la forma

$$\lambda x\ y.Q$$

Si D es una desición tomada por que O es verdadera, podemos asegurar que $(D\ M\ N) = M$, por lo tanto:

$$D \equiv \lambda x\ y.x$$

Si D es una desición tomada por que O es falsa, podemos asegurar que $(D\ M\ N) = N$, por lo tanto:

$$D \equiv \lambda x\ y.y$$

Teniendo los términos lambda que representan la desición de P ante una oración falsa y ante una oración verdadera, se puede considerar que estos términos representan el concepto de falso y verdadero.

Definición 3.1.1 (Valores de verdad)**.** El concepto de falso y verdadero es codificado en el cálculo lambda como los términos $\boldsymbol{T}$ y $\boldsymbol{F}$ respectivamente.

$$\boldsymbol{T} \equiv \lambda x\, y.x \qquad\qquad \boldsymbol{F} \equiv \lambda x\, y.y$$

Utilizar $\boldsymbol{T}$ y $\boldsymbol{F}$ en términos lambda es similar a imitar a P y determinar cuando O es verdadera o falsa. Esto es debido a que se pueden plantear predicados que sean conceptualmente ilógicos, por ejemplo, si «esPositivo» se define de tal manera que sin importar en que valor sea evaluado siempre resulte en falso, los programas que se escriban no van a funcionar suponiendo que «esPositivo» calcula lo que debe de calcular, sin embargo lo importante de codificar el álgebra booleana es poder manipular los valores de falso y verdadero, no representar un término P que determine verdades absolutas.

3.1.2. Expresiones booleanas

Las expresiones booleanas se conforman de operaciones y valores de verdad. Las operaciones más básicas son la conjunción, la disyunción y la negación, también llamadas AND, OR, NOT y denotadas $\wedge$, $\vee$ y $\neg$ respectivamente.

La conjunción y la disyunción son operaciones binarias definidas en

$$\{\text{falso},\ \text{verdadero}\}^2 \to \{\text{falso},\ \text{verdadero}\}$$

y la negación es una operación unaria definida en

$$\{\text{falso},\ \text{verdadero}\} \to \{\text{falso},\ \text{verdadero}\}.$$

Las tablas de verdad en el cuadro 3.1 establecen los resultados de estas tres operaciones para cada valor en su dominio.

x	y	$x \wedge y$	$x \vee y$
falso	falso	falso	falso
falso	verdadero	falso	verdadero
verdadero	falso	falso	verdadero
verdadero	verdadero	verdadero	verdadero

x	$\neg x$
falso	verdadero
verdadero	falso

Cuadro 3.1: Tablas de verdad para $\wedge$, $\vee$ y $\neg$

En el álgebra booleana, las expresiones se escriben en notación de infijo, utilizan paréntesis para agrupar expresiones y cuando los paréntesis son omitidos la negación tiene mayor presedencia que la conjunción y la conjunción tiene mayor presedencia que la disyunción, por ejemplo:

$$\text{verdadero} \wedge \text{falso} \vee \neg\text{falso}$$

$$\neg(\text{falso} \vee \text{falso})$$

$$\text{verdadero} \wedge (\text{falso} \vee \text{falso})$$

Esta notación es conveniente para escribir expresiones booleanas de manera concisa, pero es únicamente una conveniencia sintáctica del álgebra booleana. La codificación que se desarrolla de las operaciones seguirá las convenciones sintácticas del cálculo lambda, por ejemplo, suponiendo que $\wedge$, $\vee$, $\neg$ son términos lambda, las expresiones mencionadas escribirían con notación de prefijo:

$$\vee(\wedge\, \boldsymbol{T}\, \boldsymbol{F})\boldsymbol{F}$$

$$\neg(\vee\, \boldsymbol{F}\, \boldsymbol{F})$$

$$\wedge\, \boldsymbol{T}(\vee\, \boldsymbol{F}\, \boldsymbol{F})$$

Al igual que los valores de verdad, las operaciones básicas son codificadas como abstracciones del cálculo lambda. Hay varias metodologías para derivar términos lambda para las operaciones booleanas a partir de $\boldsymbol{T}$ y $\boldsymbol{F}$, en esta sección se abordarán dos:

- Combinando valores de verdad
- Programando las operaciones

La primer metodología parte de la observación de que la codificación de falso y verdadero son abstracciones, por lo tanto, es posible β-reducirlas al aplicarlas a otros términos; se explora la clase de términos lambda en $\{\boldsymbol{T},\ \boldsymbol{F}\}^{+}$.

La segunda metodología presenta la construcción del operador condicional, a partir del cual se derivan las operaciones booleanas como si fueran programas de computadora.

Combinaciones de valores de verdad

Una manera de obtener términos lambda a partir de $\boldsymbol{F}$ y $\boldsymbol{T}$ es β-reducir combinaciones de aplicaciones entre estos valores. En el cuadro 3.2 se muestran los términos obtenidos al reducir combinaciones de dos valores de verdad.

$\boldsymbol{F}\,\boldsymbol{F}$	$(\lambda x\, y.y)\boldsymbol{F} \twoheadrightarrow_\beta \lambda y.y \equiv \boldsymbol{I}$
$\boldsymbol{F}\,\boldsymbol{T}$	$(\lambda x\, y.y)\boldsymbol{T} \twoheadrightarrow_\beta \lambda y.y \equiv \boldsymbol{I}$
$\boldsymbol{T}\,\boldsymbol{F}$	$(\lambda x\, y.x)\boldsymbol{F} \twoheadrightarrow_\beta \lambda y.\boldsymbol{F} \equiv \boldsymbol{K}\,\boldsymbol{F}$
$\boldsymbol{T}\,\boldsymbol{T}$	$(\lambda x\, y.x)\boldsymbol{T} \twoheadrightarrow_\beta \lambda y.\boldsymbol{T} \equiv \boldsymbol{K}\,\boldsymbol{T}$

Cuadro 3.2: Posibles combinaciones de valores de verdad por pares.

En las reducciones de 3.2 se pueden observar cuatro términos, a partir de estos se puede descubrir la operación de negación:

- $(\boldsymbol{F}\,\boldsymbol{F})$ se reduce a la abstracción identidad, esto significa que para cualquier término $M \in \Lambda$

$$\boldsymbol{\lambda} \vdash (\boldsymbol{F}\,\boldsymbol{F}\,M) = M$$

- Al igual que la primer reducción $(\boldsymbol{F}\,\boldsymbol{T})$ se reduce a $\boldsymbol{I}$, por lo tanto se concluye que para cualesquiera términos $M \in \Lambda$, $N \in \{\boldsymbol{F},\ \boldsymbol{T}\}$

$$\boldsymbol{\lambda} \vdash (\boldsymbol{F}\,N\,M) = M$$

- $(\boldsymbol{T}\,\boldsymbol{F})$ se reduce a la abstracción constante de $\boldsymbol{F}$, esto significa que para cualquier término $M \in \Lambda$

$$\boldsymbol{\lambda} \vdash (\boldsymbol{T}\,\boldsymbol{F}\,M) = \boldsymbol{F}$$

- Al igual que la tercer reducción $(\boldsymbol{T}\,\boldsymbol{T})$ se reduce a $(\boldsymbol{K}\,\boldsymbol{T})$, por lo tanto se concluye que para cualesquiera términos $M \in \Lambda$, $N \in \{\boldsymbol{F},\ \boldsymbol{T}\}$

$$\boldsymbol{\lambda} \vdash (\boldsymbol{T}\,N\,M) = N$$

Debido a las reducciones mostradas en el cuadro 3.2 se puede analizar que a partir de un témino $\boldsymbol{F}$, se puede obtener $\boldsymbol{T}$ al reducir $(\boldsymbol{F}\,N\,\boldsymbol{T})$ y que a partir de un término $\boldsymbol{T}$, se puede obtener $\boldsymbol{F}$ al reducir $(\boldsymbol{T}\,\boldsymbol{F}\,M)$. Considerando que $N \equiv \boldsymbol{F}$ y $M \equiv \boldsymbol{T}$ las reducciones serían:

$$\boldsymbol{F}\,\boldsymbol{F}\,\boldsymbol{T} \twoheadrightarrow_\beta \boldsymbol{T}$$

$$\boldsymbol{T}\,\boldsymbol{F}\,\boldsymbol{T} \twoheadrightarrow_\beta \boldsymbol{F}$$

Si se considera que $P \in \{\boldsymbol{F},\ \boldsymbol{T}\}$

$$P\,\boldsymbol{F}\,\boldsymbol{T} \twoheadrightarrow_\beta \neg P$$

Observación (Sobre la β-reducción)**.** En el tratamiento de la codificación del álgebra booleana en el cálculo lambda, cuando se β-reducen términos lambda que tienen como subtérminos valores que suponemos son $\boldsymbol{F}$ o $\boldsymbol{T}$ se extiende la teoría $\boldsymbol{\lambda}$ con la siguiente ecuación:

$$P\,\boldsymbol{T}\,\boldsymbol{F} = P \qquad\qquad \text{si } P \in \{\boldsymbol{F},\ \boldsymbol{T}\}$$

Definición 3.1.2 (Operación de negación)**.** El término lambda $\neg \equiv (\lambda p.p\,\boldsymbol{F}\,\boldsymbol{T})$ se reduce a $\boldsymbol{T}$ cuando es aplicado a $\boldsymbol{F}$ y viceversa

$$\begin{aligned}\neg\boldsymbol{F} &\equiv (\lambda p.p\,\boldsymbol{F}\,\boldsymbol{T})\boldsymbol{F}\\ &\to_\beta \boldsymbol{F}\,\boldsymbol{F}\,\boldsymbol{T}\\ &\twoheadrightarrow_\beta \boldsymbol{T}\end{aligned}$$

$$\begin{aligned}\neg\boldsymbol{T} &\equiv (\lambda p.p\,\boldsymbol{F}\,\boldsymbol{T})\boldsymbol{T}\\ &\to_\beta \boldsymbol{T}\,\boldsymbol{F}\,\boldsymbol{T}\\ &\twoheadrightarrow_\beta \boldsymbol{F}\end{aligned}$$

Las reducciones del cuadro 3.2 se pueden aplicar a $\boldsymbol{F}$ y $\boldsymbol{T}$ para obtener todas las posibles combinaciones de aplicaciones de valores de verdad de la forma $((P\,M)N)$, en el cuadro 3.3 se muestran las reducciones de las nuevas aplicaciones.

Al observar el cuadro 3.3, se distinguen algúnos patrones en los resultados de las reducciones,

$\boldsymbol{F\,F\,F}$	$\boldsymbol{I\,F} \twoheadrightarrow_\beta \boldsymbol{F}$
$\boldsymbol{F\,F\,T}$	$\boldsymbol{I\,T} \twoheadrightarrow_\beta \boldsymbol{T}$
$\boldsymbol{F\,T\,F}$	$\boldsymbol{I\,F} \twoheadrightarrow_\beta \boldsymbol{F}$
$\boldsymbol{F\,T\,T}$	$\boldsymbol{I\,T} \twoheadrightarrow_\beta \boldsymbol{T}$
$\boldsymbol{T\,F\,F}$	$\boldsymbol{K\,F\,F} \twoheadrightarrow_\beta \boldsymbol{F}$
$\boldsymbol{T\,F\,T}$	$\boldsymbol{K\,F\,T} \twoheadrightarrow_\beta \boldsymbol{F}$
$\boldsymbol{T\,T\,F}$	$\boldsymbol{K\,T\,F} \twoheadrightarrow_\beta \boldsymbol{T}$
$\boldsymbol{T\,T\,T}$	$\boldsymbol{K\,T\,T} \twoheadrightarrow_\beta \boldsymbol{T}$

Cuadro 3.3: Posibles combinaciones de valores de verdad con asociación a la izquierda.

por ejemplo, si P es un valor de verdad cualquiera, $(P\,\boldsymbol{F}\,\boldsymbol{F})$ se reduce a $\boldsymbol{F}$ y $(P\,\boldsymbol{T}\,\boldsymbol{T})$ se reduce a $\boldsymbol{T}$, las combinaciones mas interesantes se presentan en los renglones 2, 3, 6 y 7.

En búsqueda de las operaciones binarias de conjunción y disyunción se desarrollan tablas de verdad con las posibles combinaciones de dos términos $P, Q \in \{\boldsymbol{F},\ \boldsymbol{T}\}$. La cantidad de combinaciones de estos valores es $2 \times \binom{3}{2} = 2 \times \frac{3!}{2!} = 6$ y son $(P\,Q\,\boldsymbol{F})$, $(P\,Q\,\boldsymbol{T})$, $(P\,\boldsymbol{F}\,Q)$, $(P\,\boldsymbol{T}\,Q)$, $(\boldsymbol{F}\,P\,Q)$, $(\boldsymbol{T}\,P\,Q)$. Las tablas de verdad de estas combinaciones intercambiando las posiciones de P y Q serían las mismas ya que ambos términos toman los valores de falso y verdadero en las tablas de verdad. En el cuadro 3.4 se muestran estas tablas.

P	Q	$P\,Q\,\boldsymbol{F}$	$P\,Q\,\boldsymbol{T}$	$P\,\boldsymbol{F}\,Q$	$P\,\boldsymbol{T}\,Q$	$\boldsymbol{F}\,P\,Q$	$\boldsymbol{T}\,P\,Q$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{T}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{F}$	$\boldsymbol{T}$	$\boldsymbol{F}$	$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{F}$
$\boldsymbol{T}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{T}$	$\boldsymbol{F}$	$\boldsymbol{T}$
$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{F}$	$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{T}$

Cuadro 3.4: Tablas de verdad considerando dos variables P y Q en aplicaciones de tres términos

Las columnas de la combinación $(P\,Q\,\boldsymbol{F})$ y $(P\,\boldsymbol{T}\,Q)$ del cuadro 3.4 corresponden a la operación de conjunción y disyunción respectivamente, como aparecen en el cuadro 3.1. Las otras combinaciones corresponden a operaciones no básicas del álgebra booleana: $(P\,Q\,\boldsymbol{T})$ es la implicación material; $(P\,\boldsymbol{F}\,Q)$ es la no implicación inversa; $(\boldsymbol{F}\,P\,Q)$ es la proyección de Q; y $(\boldsymbol{T}\,P\,Q)$ es la proyección de P.

Ya que $\boldsymbol{\lambda} \vdash (P\,Q\,\boldsymbol{F}) = (\boldsymbol{\wedge}\,P\,Q)$ y $\boldsymbol{\lambda} \vdash (P\,\boldsymbol{T}\,Q) = (\boldsymbol{\vee}\,P\,Q)$, se construyen los términos $\boldsymbol{\wedge}$ y $\boldsymbol{\vee}$ abstrayendo a P y Q de las igualdades.

Definición 3.1.3 (Operación de conjunción)**.** El término lambda que representa la conjunción es

$$\boldsymbol{\wedge} \equiv \lambda p\,q.p\,q\,\boldsymbol{F}$$

y cumple las siguientes propiedades de β-reducción al ser aplicada a valores de verdad:

$$\begin{aligned} \boldsymbol{\wedge\,F\,F} &\equiv (\lambda p\,q.p\,q\,\boldsymbol{F})\boldsymbol{F\,F} \\ &\twoheadrightarrow_\beta \boldsymbol{F\,F\,F} \\ &\twoheadrightarrow_\beta \boldsymbol{F} \end{aligned}$$

$$\begin{aligned} \wedge\, \boldsymbol{F}\, \boldsymbol{T} &\equiv (\lambda p\, q.p\, q\, \boldsymbol{F})\boldsymbol{F}\, \boldsymbol{T} \\ &\twoheadrightarrow_\beta \boldsymbol{F}\, \boldsymbol{T}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{F} \end{aligned}$$

$$\begin{aligned} \wedge\, \boldsymbol{T}\, \boldsymbol{F} &\equiv (\lambda p\, q.p\, q\, \boldsymbol{F})\boldsymbol{T}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{T}\, \boldsymbol{F}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{F} \end{aligned}$$

$$\begin{aligned} \wedge\, \boldsymbol{T}\, \boldsymbol{T} &\equiv (\lambda p\, q.p\, q\, \boldsymbol{F})\boldsymbol{T}\, \boldsymbol{T} \\ &\twoheadrightarrow_\beta \boldsymbol{T}\, \boldsymbol{T}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{T} \end{aligned}$$

Definición 3.1.4 (Operación de disyunción)**.** El término lambda que representa la disyunción es

$$\vee \equiv \lambda p\, q.p\, \boldsymbol{T}\, q$$

y cumple las siguientes propiedades de β-reducción al ser aplicada a valores de verdad:

$$\begin{aligned} \vee\, \boldsymbol{F}\, \boldsymbol{F} &\equiv (\lambda p\, q.p\, \boldsymbol{T}\, q)\boldsymbol{F}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{F}\, \boldsymbol{T}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{F} \end{aligned}$$

$$\begin{aligned} \vee\, \boldsymbol{F}\, \boldsymbol{T} &\equiv (\lambda p\, q.p\, \boldsymbol{T}\, q)\boldsymbol{F}\, \boldsymbol{T} \\ &\twoheadrightarrow_\beta \boldsymbol{F}\, \boldsymbol{T}\, \boldsymbol{T} \\ &\twoheadrightarrow_\beta \boldsymbol{T} \end{aligned}$$

$$\begin{aligned} \vee\, \boldsymbol{T}\, \boldsymbol{F} &\equiv (\lambda p\, q.p\, \boldsymbol{T}\, q)\boldsymbol{T}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{T}\, \boldsymbol{T}\, \boldsymbol{F} \\ &\twoheadrightarrow_\beta \boldsymbol{T} \end{aligned}$$

$$\begin{aligned} \vee \boldsymbol{T}\,\boldsymbol{T} &\equiv (\lambda p\, q.p\, \boldsymbol{T}\, q)\boldsymbol{T}\,\boldsymbol{T} \\ &\twoheadrightarrow_\beta \boldsymbol{T}\,\boldsymbol{T}\,\boldsymbol{T} \\ &\twoheadrightarrow_\beta \boldsymbol{T} \end{aligned}$$

Esta metodología para encontrar operaciones del álgebra booleana, aplicando los términos codificados de los valores de verdad, es tediosa pero hasta cierto grado efectiva. Como en los casos de las operaciones no básicas mostradas en el cuadro 3.4, operaciones del álgebra booleana pueden ser "descubiertas" y no construídas. Ya que la negación, la conjunción y la disyunción fueron descubiertas con este método, cualquier operación booleana eventualmente será encontrada como combinación de valores de verdad. Sin embargo, descubrir la codificación de una operación booleana complicada utilizando este método es un proceso muy tardado.

Programación de operaciones booleanas

Otra metodología que permite construír las operaciones booleanas como términos lambda es la de partir de un algorítmo que las describa. Usualmente las operaciones booleanas no son definidas como procedimientos, si no como operaciones primitivas del lenguaje utilizado para describirlos.

Consideremos dos términos M y N. Ya que $(\boldsymbol{T}\,M\,N) \twoheadrightarrow_\beta M$ y $(\boldsymbol{F}\,M\,N) \twoheadrightarrow_\beta N$, si $M \twoheadrightarrow_\beta M'$ y $N \twoheadrightarrow_\beta N'$, entonces

$$\boldsymbol{T}\,M\,N \twoheadrightarrow_\beta M'$$

$$\boldsymbol{F}\,M\,N \twoheadrightarrow_\beta N'$$

Es decir, si $P \in \{\boldsymbol{F}, \boldsymbol{T}\}$:

$$P\,M\,N \twoheadrightarrow_\beta \begin{cases} M' & P \equiv \boldsymbol{T} \\ N' & P \equiv \boldsymbol{F} \end{cases}$$

Esta aplicación de un valor de verdad a dos términos lambda cualquiera permite capturar el concepto de una expresión o sentencia condicional, usualmente llamada en los lenguajes de programación como sentencia `if-then-else`.

Definición 3.1.5 (Expresión condicional)**.** El término lambda que representa a la expresión condicional es

$$\prec \equiv \lambda p\, m\, n.p\, m\, n$$

Y si P es un valor de verdad, entonces

$$\begin{aligned} \prec P\,M\,N &\equiv (\lambda p\, m\, n.p\, m\, n)P\,M\,N \\ &\twoheadrightarrow_\beta P\,M\,N \end{aligned}$$

Un programa de la forma

si P **entonces**
  M
de lo contrario
  M
fin si

Puede ser traducido a $(\prec\ P\ M\ N)$

Consideremos la siguiente definición en pseudocódigo de la operación de negación:

Algoritmo 3.1 Negación de p

Entrada: $p \in \{\text{falso},\ \text{verdadero}\}$
Salida: $\neg p$
  si p **entonces**
    regresar falso
  de lo contrario
    regresar verdadero
  fin si

El pseudocódigo se traduce al cálculo lambda como

$$\neg\ \equiv\ \lambda p.\prec p\ \boldsymbol{F}\ \boldsymbol{T}$$

El cuerpo de la abstracción puede ser β-reducido para obtener el término de la definición 3.1.2

$$\begin{aligned}\lambda p.\prec p\ \boldsymbol{F}\ \boldsymbol{T}\ &\equiv\ \lambda p.(\lambda p\ m\ n.p\ m\ n)p\ \boldsymbol{F}\ \boldsymbol{T}\\ &\twoheadrightarrow_\beta\ \lambda p.p\ \boldsymbol{F}\ \boldsymbol{T}\end{aligned}$$

Para la operación de conjunción, se considera el siguiente pseudocódigo:

Algoritmo 3.2 Conjunción de p_1 y p_2

Entrada: $p_1, p_2 \in \{\text{falso},\ \text{verdadero}\}$
Salida: $p_1 \wedge p_2$
  si p_1 **entonces**
    si p_2 **entonces**
      regresar verdadero
    de lo contrario
      regresar false
    fin si
  de lo contrario
    regresar falso
  fin si

Traducido al cálculo lambda como

$$\wedge\ \equiv\ \lambda p_1\ p_2.\prec p_1(\prec\ p_2\ \boldsymbol{T}\ \boldsymbol{F})\boldsymbol{F}$$

Al β-reducir el cuerpo de la abstracción se obtiene el término de la definición 3.1.3

$$\begin{aligned}
\lambda p_1\, p_2.\prec p_1(\prec p_2\, \boldsymbol{T}\, \boldsymbol{F})\boldsymbol{F} &\equiv \lambda p_1\, p_2.(\lambda p\, m\, n.p\, m\, n)p_1((\lambda p\, m\, n.p\, m\, n)p_2\, \boldsymbol{T}\, \boldsymbol{F})\boldsymbol{F} \\
&\twoheadrightarrow_\beta \lambda p_1\, p_2.p_1((\lambda p\, m\, n.p\, m\, n)p_2\, \boldsymbol{T}\, \boldsymbol{F})\boldsymbol{F} \\
&\twoheadrightarrow_\beta \lambda p_1\, p_2.p_1(p_2\, \boldsymbol{T}\, \boldsymbol{F})\boldsymbol{F} \\
&=_\lambda \lambda p_1\, p_2.p_1\, p_2\, \boldsymbol{F}
\end{aligned}$$

De igual manera, considerando el siguiente pseudocódigo de la operación de disyunción:

Algoritmo 3.3 Disyunción de p_1 y p_2

Entrada: $p_1, p_2 \in \{\text{falso}, \text{verdadero}\}$
Salida: $p_1 \vee p_2$
  si p_1 **entonces**
    regresar verdadero
  de lo contrario
    si p_2 **entonces**
      regresar verdadero
    de lo contrario
      regresar falso
    fin si
  fin si

Se traduce al cálculo lambda como

$$\vee \equiv \lambda p_1\, p_2.\prec p_1\, \boldsymbol{T}(\prec p_2\, \boldsymbol{T}\, \boldsymbol{F})$$

Y al β-reducir el cuerpo de la abstracción se obtiene el término de la definición 3.1.4

$$\begin{aligned}
\lambda p_1\, p_2.\prec p_1\, \boldsymbol{T}(\prec p_2\, \boldsymbol{T}\, \boldsymbol{F}) &\equiv \lambda p_1\, p_2.(\lambda p\, m\, n.p\, m\, n)p_1\, \boldsymbol{T}((\lambda p\, m\, n.p\, m\, n)p_2\, \boldsymbol{T}\, \boldsymbol{F}) \\
&\twoheadrightarrow_\beta \lambda p_1\, p_2.p_1\, \boldsymbol{T}((\lambda p\, m\, n.p\, m\, n)p_2\, \boldsymbol{T}\, \boldsymbol{F}) \\
&\twoheadrightarrow_\beta \lambda p_1\, p_2.p_1\, \boldsymbol{T}(p_2\, \boldsymbol{T}\, \boldsymbol{F}) \\
&=_\lambda \lambda p_1\, p_2.p_1\, \boldsymbol{T}\, p_2
\end{aligned}$$

Utilizando esta técnica, se puede obtener el término lambda para una operación a partir del pseudocódigo basado en valores de verdad y la sentencia `if-then-else`. Teniendo estos resultados resulta natural, generalizar el pseudocódigo para construír un término lambda que a partir de la tabla de verdad de una operación booleana binaria, resulte en la abstracción que codifica la operación.

Definición 3.1.6 (Traducción de operaciones booleanas binarias)**.** Sea $\odot$ una operación booleana binaria con la siguiente tabla de verdad

P	Q	$P \odot Q$
$\boldsymbol{F}$	$\boldsymbol{F}$	x_1
$\boldsymbol{F}$	$\boldsymbol{T}$	x_2
$\boldsymbol{T}$	$\boldsymbol{F}$	x_3
$\boldsymbol{T}$	$\boldsymbol{T}$	x_4

El procedimiento generalizado es

Algoritmo 3.4 Operación booleana $\odot$ dado x_1, x_2, x_3, x_3

Entrada: $p_1, p_2 \in \{\text{falso, verdadero}\}$
Salida: Valor x_i de la tabla de verdad
si p_1 **entonces**
  si p_2 **entonces**
    regresar x_4
  de lo contrario
    regresar x_3
  fin si
de lo contrario
  si p_2 **entonces**
    regresar x_2
  de lo contrario
    regresar x_1
  fin si
fin si

Y la traducción al cálculo lambda es

$$\lambda x_1\, x_2\, x_3\, x_4.(\lambda p_1\, p_2.(\prec\ p_1(\prec\ p_2\, x_4\, x_3)(\prec\ p_2\, x_2\, x_1)))$$

Ejemplo 3.1.1 (Operaciones *NAND* y *NOR*)**.** Estas operaciones booleanas binarias conforman los conjuntos unitarios {NAND} y {NOR} los cuales son conjuntos funcionalmene completos, es decir, únicamente con la operación NAND se puede emular cualquier operación booleana y únicamente con la operación NOR se puede emular cualquier operación booleana.

La operación *NAND* se denota $P \uparrow Q$ y tiene la siguiente tabla de verdad

P	Q	$P \uparrow Q$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{T}$
$\boldsymbol{F}$	$\boldsymbol{T}$	$\boldsymbol{T}$
$\boldsymbol{T}$	$\boldsymbol{F}$	$\boldsymbol{T}$
$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{F}$

Con el proceso de traducción mostrado en la definición 3.1.6, el término lambda $\uparrow$ que codifica la operación NAND sería

$$\begin{aligned}\uparrow \;&\equiv\; \lambda p_1\, p_2.\prec\ p_1(\prec\ p_2\, \boldsymbol{F}\, \boldsymbol{T})(\prec\ p_2\, \boldsymbol{T}\, \boldsymbol{T})\\ &\twoheadrightarrow_\beta\; \lambda p_1\, p_2.p_1(p_2\, \boldsymbol{F}\, \boldsymbol{T})\boldsymbol{T}\end{aligned}$$

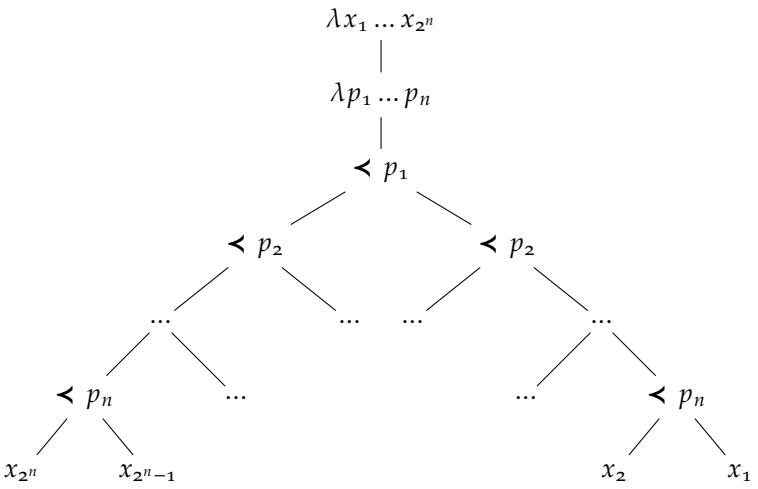
La operación *NOR* se denota $P \downarrow Q$ y tiene la siguiente tabla de verdad

P	Q	$P \downarrow Q$
$\boldsymbol{F}$	$\boldsymbol{F}$	$\boldsymbol{T}$
$\boldsymbol{F}$	$\boldsymbol{T}$	$\boldsymbol{F}$
$\boldsymbol{T}$	$\boldsymbol{F}$	$\boldsymbol{F}$
$\boldsymbol{T}$	$\boldsymbol{T}$	$\boldsymbol{F}$

Usando el mismo proceso de traducción que con la operación NAND, se obtiene

$$\begin{aligned} \downarrow &\equiv \lambda p_1\, p_2.\prec p_1(\prec p_2\, \boldsymbol{F}\, \boldsymbol{F})(\prec p_2\, \boldsymbol{F}\, \boldsymbol{T}) \\ &\twoheadrightarrow_\beta \lambda p_1\, p_2.p_1\, \boldsymbol{F}(p_2\, \boldsymbol{F}\, \boldsymbol{T}) \end{aligned}$$

Cuando se generaliza el método de traducción de 3.1.6 a operaciones booleanas n-árias, se obtiene un término bosquejado de la siguiente manera



3.1.3. Extensiones al álgebra booleana

Conociendo el proceso de codificación del álgebra booleana en el cálculo lambda, resulta simple adaptar la codificación.

Consideremos el caso en donde, además de tener los valores de falso y verdadero, se desea incorporar un valor "desconocido" utilizado para representar un valor que no es ni falso, ni verdadero. La interpretación de estos valores es similar a 3.1.1, pero en lugar de decidir sobre dos términos, se decide sobre tres.

Definición 3.1.7 (Valores de álgebra trivalente)**.** La codificación en términos lambda de los valores de ésta álgebra trivalente son

$$\begin{aligned} \boldsymbol{T} &\equiv \lambda x\, y\, z.x \\ \boldsymbol{F} &\equiv \lambda x\, y\, z.y \\ \boldsymbol{U} &\equiv \lambda x\, y\, z.z \end{aligned}$$

Al igual que en la codificación bivalente, se puede codificar un término $\prec_3$, similar a $\prec$ de la definición 3.1.5 pero con tres ramificaciones

Definición 3.1.8 (Condicional trivalente)**.**

$$\prec_3 \equiv \lambda p\, m\, n\, o.p\, m\, n\, o$$

De tal manera que, si $P \in \{\boldsymbol{T},\ \boldsymbol{F},\ \boldsymbol{U}\}$

$$(\prec_3\ P\,M\,N\,O) \twoheadrightarrow_\beta \begin{cases} M & P \equiv \boldsymbol{T}; \\ N & P \equiv \boldsymbol{F}; \\ O & P \equiv \boldsymbol{U}. \end{cases}$$

Sea $\odot$ una operación trivalente binaria con la siguiente tabla de valores

P	Q	$P \odot Q$
$\boldsymbol{T}$	$\boldsymbol{T}$	x_1
$\boldsymbol{T}$	$\boldsymbol{F}$	x_2
$\boldsymbol{T}$	$\boldsymbol{U}$	x_3
$\boldsymbol{F}$	$\boldsymbol{T}$	x_4
$\boldsymbol{F}$	$\boldsymbol{F}$	x_5
$\boldsymbol{F}$	$\boldsymbol{U}$	x_6
$\boldsymbol{U}$	$\boldsymbol{T}$	x_7
$\boldsymbol{U}$	$\boldsymbol{F}$	x_8
$\boldsymbol{U}$	$\boldsymbol{U}$	x_9

El procedimiento en pseudocódigo que la describe es:

Algoritmo 3.5 Operación booleana $\odot$ dado $x_1, x_2, \ldots, x_9$

Entrada: $p_1, p_2 \in$ {verdadero, falso, desconocido}
Salida: Valor x_i de la tabla de verdad

```
si p1 = verdadero entonces
  si p2 = verdadero entonces
    regresar x1
  de lo contrario y si p2 = falso entonces
    regresar x2
  de lo contrario y si p2 = desconocido entonces
    regresar x3
  fin si
de lo contrario y si p1 = falso entonces
  si p2 = verdadero entonces
    regresar x4
  de lo contrario y si p2 = falso entonces
    regresar x5
  de lo contrario y si p2 = desconocido entonces
    regresar x6
  fin si
de lo contrario y si p1 = desconocido entonces
  si p2 = verdadero entonces
    regresar x7
  de lo contrario y si p2 = falso entonces
    regresar x8
  de lo contrario y si p2 = desconocido entonces
    regresar x9
  fin si
fin si
```

Traducido al cálculo lambda como

$$\lambda x_1\, x_2\, x_3\, x_4\, x_5\, x_6\, x_7\, x_8\, x_9.(\lambda p_1\, p_2.(\prec_3\ p_1\, R_1\, R_2\, R_3))$$

Donde

$$\begin{aligned} R_1 &\equiv (\prec_3\ p_2\, x_1\, x_2\, x_3) \\ R_2 &\equiv (\prec_3\ p_2\, x_4\, x_5\, x_6) \\ R_3 &\equiv (\prec_3\ p_2\, x_7\, x_8\, x_9) \end{aligned}$$

3.1.4. Lógica en el cálculo lambda

En esta sección se abordó la codificación de valores de verdad y operaciones como la conjunción, disyunción y negación. Usualmente en las ciencias de la computación y las matemáticas, estos objetos se abordan desde la perspectiva algebraica y desde la perspectiva lógica.

En el cálculo lambda se distinguen las dos perspectivas por el lenguaje en el que se manejan los objetos, en el caso del álgebra booleana, los valores de verdad y las operaciones booleanas se codifican como términos lambda y pueden ser combinados con otros términos lambda que no forman parte del álgebra booleana. Por otra parte, la lógica proposicional estudia proposiciones, las cuales son enunciados matemáticos a los cuales se les puede atribuír el valor de falso o verdadero y que, a partir de conectivos lógicos como $\neg$, $\wedge$ y $\vee$, se infieren verdades sobre los enunciados.

El cálculo lambda se puede extender de tal manera que nos permita inferir verdades sobre sus términos, en esta subsección se abordan los cálculos lambda aumentados con nociones *ilativas*, denotados $i\boldsymbol{\lambda}$. El tratamiento de esta extensión está basado en el apéndice B "Lógica combinatoria ilativa" de [5, pp. 573–576].

Definición 3.1.9 (Términos $i\boldsymbol{\lambda}$)**.** Los términos de la teoría $i\boldsymbol{\lambda}$, denotados $i\Lambda$ se definen sobre el alfabeto de Λ extendido con un conjunto C de *constantes lógicas*.

$$\begin{aligned} M \in \Lambda &\implies M \in i\Lambda \\ c \in C &\implies c \in i\Lambda \\ M, N \in i\Lambda &\implies (M\, N) \in i\Lambda \end{aligned}$$

Definición 3.1.10 (Fórmulas en $i\boldsymbol{\lambda}$)**.** Las fórmulas de la teoría $i\boldsymbol{\lambda}$ se definen de la siguiente manera

- Si $M, N \in i\Lambda$, entonces $M = N$ es una fórmula.
- Si $M \in i\Lambda$, entonces M es una fórmula.

En el estudio de teorías ilativas, la interpretación de $i\boldsymbol{\lambda} \vdash M$ y $i\boldsymbol{\lambda} \vdash M = N$ es "La fórmula M es verdadera" y "La fórmula $M = N$ es verdadera" respectivamente.

Revisar La sección será muy extensa si desarrollo todos los pendientes planteados, sería conveniente no plantear la paradoja de Kleene-Rosser y considerar solo la de Curry, mostrar que cualquier término puede

ser inferido, omitir la definibilidad de las constantes en términos de $\boldsymbol{Q}$ y $\boldsymbol{\Xi}$ y mencionar la compensación entre expresibilidad e inconsistencia, enlazando las ideas de computabilidad y teoría de tipos.

Teoría $i\boldsymbol{\lambda}_{KR}$

Constantes $C = \{\boldsymbol{N}\}$ con la interpretación $\neg\, M$ es $(\boldsymbol{N}\, M)$
Paradoja de Kleene-Rosser

$$X \;\equiv\; \lambda x.\boldsymbol{N}(x\, x)$$

$$X\, X = (\lambda x.\boldsymbol{N}(x\, x))X = \boldsymbol{N}(X\, X)$$

Se puede derivar que una fórmula $X\, X = \neg(X\, X)$, es decir, en $i\boldsymbol{\lambda}_{KR}$ se pueden derivar contradicciones.

Teoría $i\boldsymbol{\lambda}_{\mathrm{o}}$

Constantes $C = \{\boldsymbol{F},\ \boldsymbol{Q},\ \boldsymbol{E},\ \boldsymbol{\Xi},\ \boldsymbol{\Pi},\ \boldsymbol{P}\}$ con las siguientes interpretaciones

- $M \in N$ es $N\, M$
- $M \implies N$ es $\boldsymbol{P}\, M\, N$
- $M \subseteq N$ es $\boldsymbol{\Xi}\, M\, N$
- N^M es $\boldsymbol{F}\, M\, N$
- $\forall x\ M$ es $\boldsymbol{\Xi}(\lambda x.M)$

Pendientes

- Definibilidad de $\boldsymbol{E}$, $\boldsymbol{F}$, $\boldsymbol{P}$ y $\boldsymbol{\Pi}$ a partir de $\boldsymbol{Q}$, $\boldsymbol{\Xi}$.
- Inconsistencia de $i\boldsymbol{\lambda}_{\mathrm{o}}$ por la paradoja de Curry "Una teoría lambda ilativa es inconsistente si cada M puede ser derivada" (la paradoja de Kleene-Rosser es una consecuencia de la paradoja de Curry).
- Mencionar la técnica utilizada por Church para "escapar" de la paradoja de Curry y presentar el "tradeoff" de tener un sistema suficientemente poderoso: Equivalencia $\boldsymbol{\lambda}$ y máquina de Turing *vs.* separar la noción ilativa de la abstracción.

3.2. Aritmética

La aritmética es una de las ramas mas antiguas de las matemáticas. Consiste en el estudio de los números y de las operaciones elementales como la suma y la multiplicación. El manejo de expresiones aritméticas es ubicuo en la vida cotidiana y es una parte fundamental de la formación básica en matemáticas.

En el cálculo lambda, los números naturales no son términos lambda, sin embargo, desde el metalenguaje se pueden manejan números naturales y expresiones aritméticas como por ejemplo en la definición

de longitud 2.1.3. Al igual que el álgebra boolena, las expresiones aritméticas pueden ser codificadas como términos lambda.

En los lenguajes de programación los números naturales y las operaciones aritméticas son de los objetos más utilizados para expresar la mayoría de los cómputos. Virtualmente todo programa no trivial ejecutable en una computadora hace uso de números y operaciones sobre ellos. Como se menciona al inicio de la sección anterior, el concepto de número es codificado usualmente como una secuencia de bits de longitud fija y las operaciones aritméticas terminan siendo traducidas a instrucciones ejecutadas por la unidad aritmética lógica de la computadora.

En esta sección se plantea la codificación de expresiones aritméticas en el lenguaje del cálculo lambda de manera similar a como se abordó en 3.1, también se aborda la representación de la noción de iteración y aglunos mecanismos que nos permiten abstaer el cómputo de las operaciones elementales.

3.2.1. Numerales de Church

Los números naturales son los objetos más básicos para representar expresiones aritméticas. En este trabajo se considera que $\mathbb{N}$ contiene el número 0, por lo que el conjunto de números naturales es

$$\mathbb{N} = \{0,\ 1,\ 2,\ 3,\ \ldots\}$$

En la codificación del álgebra booleana se presenta la representación de valores de verdad como una desición entre dos valores dados. En 3.1.3 se extende la representación de la desición a tres valores y siguiendo el mismo procedimiento se puede extender a n valores. Esta representación no es útil al tratar con los números naturales ya que no se tiene un conjunto finito de valores, sin embargo, si se establece una cota superior para la cantidad de números naturales representables es posible utilizar esta codificación.

Por cuestiones de eficiencia, en las computadoras se limita la cantidad de naturales representables a valores entre 0 y $2^{64} - 1$, por lo tanto, es posible representar números en este rango como abstracciones de 2^{64} variables enlazadas. Utilizar esta codificación no es conveniente ya que las operaciones deberán ser definidas para cada posible combinación de naturales.

La codificación más utilizada para números naturales es la de *numerales de Church*, esta codificación fue propuesta por Alonzo Church en 1941 [9, p. 28].

Definición 3.2.1 (Numerales de Church)**.** El numeral de Church, denotado $\widehat{n}$, asociado al número natural n es

$$\widehat{n} \equiv (\lambda x\ y.x^n\ y) \tag{3.1}$$

Ejemplo 3.2.1. Los primeros 5 numerales son:

$$\begin{aligned}
\widehat{0} &\equiv \lambda x\ y.x^0\ y \equiv \lambda x\ y.y \\
\widehat{1} &\equiv \lambda x\ y.x^1\ y \equiv \lambda x\ y.x\ y \\
\widehat{2} &\equiv \lambda x\ y.x^2\ y \equiv \lambda x\ y.x(x\ y) \\
\widehat{3} &\equiv \lambda x\ y.x^3\ y \equiv \lambda x\ y.x(x(x\ y)) \\
\widehat{4} &\equiv \lambda x\ y.x^4\ y \equiv \lambda x\ y.x(x(x(x\ y)))
\end{aligned}$$

Al manipular numerales de Church, se debe tener cuidado en la manera en que se reducen aplicaciones con otros términos. Con la codificación de valores de verdad es relativamente sencillo experimentar con la manera en la que $\boldsymbol{T}$ y $\boldsymbol{F}$ se combinan y corroborar manualmente que una combinación se reduce a otra. Sin embargo, al aplicar un numeral de Church $\widehat{n}$ a un término cualquiera M, el término resultante de la contracción de dicha aplicación tendría aproximadamente una longitud de $\|M\| \times n$, lo cual resulta inconveniente de escribir en cada paso de una reducción.

Para facilitar el desarrollo de reducciones se muestran algunas propiedades de los numerales de Church y reducciones que involucran términos de la forma $(P^n\ Q)$.

Consideremos un numeral $\widehat{n}$ y términos cualesquiera P y Q.

$$\widehat{n}\,P \ \to_\beta\ \lambda x.P^n\ x \tag{3.2}$$

$$\widehat{n}\,P\,Q \ \to_\beta (\lambda x.P^n\ x)Q \ \to_\beta\ P^n\ Q \tag{3.3}$$

Sea $\widehat{n}$ un numeral de Church, P, Q y R términos cualesquiera y m un número natural. Se aborda la reducción de las aplicaciones $((\widehat{n}\,P\,Q)^m R)$, $((\widehat{n}\,P)^m Q)$ y $(\widehat{n}^m\ P)$, las cuales corresponden a la aplicación $(F^m\ X)$ donde F es el término de las ecuaciones (3.3), (3.2) y (3.1).

$$(\widehat{n}\,P\,Q)^m R \ \twoheadrightarrow_\beta (P^n\ Q)^m R \qquad \text{Por (3.3)} \tag{3.4}$$

Ya que no se hicieron suposiciones adicionales sobre P y Q no podemos asegurar que el término final de la reducción (3.4) pueda ser reducido más.

$$\begin{aligned}
(\widehat{n}\,P)^m Q &\equiv (\widehat{n}\,P)^{m-1}(\widehat{n}\,P\,Q) && \text{Por (2.3)} \\
&\twoheadrightarrow_\beta (\widehat{n}\,P)^{m-1}(P^n\ Q) && \text{Por (3.3)} \\
&\equiv (\widehat{n}\,P)^{m-2}(\widehat{n}\,P(P^n\ Q)) && \text{Por (2.3)} \\
&\twoheadrightarrow_\beta (\widehat{n}\,P)^{m-2}(P^n(P^n\ Q)) && \text{Por (3.3)} \\
&\equiv (\widehat{n}\,P)^{m-2}(P^{2\times n}\ Q) \\
&\quad \ldots && \text{Repitiendo para } m-3, \ldots, m-m \\
&\twoheadrightarrow_\beta (\widehat{n}\,P)^{m-m}(P^{m\times n}\ Q) \\
&\equiv P^{m\times n}\ Q && \text{Por (2.3)}
\end{aligned} \tag{3.5}$$

$$
\begin{aligned}
\widehat{n}^m\, P &\equiv \widehat{n}^{m-1}(\widehat{n}\, P) && \text{Por (2.3)} \\
&\equiv \widehat{n}^{m-2}(\widehat{n}(\widehat{n}\, P)) && \text{Por (2.3)} \\
&\to_\beta \widehat{n}^{m-2}(\lambda x.(\widehat{n}\, P)^n x) && \text{Por (3.2)} \\
&\twoheadrightarrow_\beta \widehat{n}^{m-2}(\lambda x.P^{n\times n}\, x) && \text{Por (3.5)} \\
&\equiv \widehat{n}^{m-2}(\widehat{n\times n}\, P) && \text{Por (3.2)} \\
&\quad \ldots && \text{Repitiendo para } m-3, \ldots, m-m \\
&\twoheadrightarrow_\beta \widehat{n}^{m-m}(\widehat{n^m}\, P) \\
&\equiv (\widehat{n^m}\, P) && \text{Por (2.3)}
\end{aligned}
\tag{3.6}
$$

Consideremos ahora la reducción de aplicaciones de numerales de la forma $(\widehat{n}\,\widehat{m})$.

Cuando $\widehat{n} \equiv \widehat{0}$ se tienen términos $(\widehat{0}\,\widehat{m})$, por la definición (3.1) se tiene que $\widehat{0}$ y $\mathbf{F}$ son sintácticamente equivalentes, en el cuadro 3.2 se mostró que $(\boldsymbol{F}\, M) \twoheadrightarrow_\beta \boldsymbol{I}$ para todo término M, por lo tanto, se asegura que en general $(\widehat{0}\, M) \twoheadrightarrow_\beta \boldsymbol{I}$.

Cuando $\widehat{n} \equiv \widehat{1}$ los pasos de la reducción con $\widehat{m} \in \{\widehat{0}, \widehat{1}, \widehat{2}\}$ son:

Consideremos la reducción de una aplicación de numerales $(\widehat{n}\,\widehat{m})$. Cuando $\widehat{n} \equiv \widehat{0}$ se tiene que para todo natural m, $(\widehat{0}\,\widehat{m}) \twoheadrightarrow_\beta \boldsymbol{I}$ debido a que $\widehat{0} \equiv \boldsymbol{F}$ y $(\boldsymbol{F}\, M) \twoheadrightarrow_\beta \boldsymbol{I}$. Cuando $\widehat{n} \equiv \widehat{1}$ las reducciones siguen siendo manejables, para los casos donde $\widehat{m} \equiv \widehat{0}, \widehat{1}, \widehat{2}$ se obtienen las siguientes reducciones:

$$
\begin{aligned}
\widehat{1}\,\widehat{0} &\to_\beta (\lambda x.\widehat{0^1}\, x) && \text{Por (3.2)} \\
&\twoheadrightarrow_\beta (\lambda x.\widehat{0^1}\, x) && \text{Por (3.6)} \\
&\to_\beta (\lambda x\, y.x^0\, y) && \text{Por (3.2)} \\
&\equiv \widehat{0}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{1}\,\widehat{1} &\to_\beta (\lambda x.\widehat{1^1}\, x) && \text{Por (3.2)} \\
&\twoheadrightarrow_\beta (\lambda x.\widehat{1^1}\, x) && \text{Por (3.6)} \\
&\to_\beta (\lambda x\, y.x^1\, y) && \text{Por (3.2)} \\
&\equiv \widehat{1}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{1}\,\widehat{2} &\to_\beta (\lambda x.\widehat{2^1}\, x) && \text{Por (3.2)} \\
&\twoheadrightarrow_\beta (\lambda x.\widehat{2^1}\, x) && \text{Por (3.6)} \\
&\to_\beta (\lambda x\, y.x^2\, y) && \text{Por (3.2)} \\
&\equiv \widehat{2}
\end{aligned}
$$

Para cualquier numeral $\widehat{m}$:

$$
\widehat{1}\,\widehat{m} \to_\beta \lambda x.\widehat{m^1}\, x \twoheadrightarrow_\beta \lambda x.\widehat{m^1}\, x \to_\beta \lambda x\, y.x^m\, y \equiv \widehat{m}
$$

Cuando se considera $\widehat{n} \equiv \widehat{2}$ las reducciones siguen los mismos pasos que en el caso anterior:

$$\widehat{2}\,\widehat{m} \to_\beta \lambda x.\widehat{m}^2\, x \twoheadrightarrow_\beta \lambda x.\widehat{m^2}\, x \to_\beta \lambda x\, y.x^{m^2}\, y \equiv \widehat{m^2}$$

Lo cual nos lleva a concluír que en el caso general, para cualesquiera numerales $\widehat{n}$ y $\widehat{m}$:

$$\widehat{n}\,\widehat{m} \to_\beta \lambda x.\widehat{m}^n\, x \twoheadrightarrow_\beta \lambda x.\widehat{m^n}\, x \to_\beta \lambda x\, y.x^{m^n}\, y \equiv \widehat{m^n} \tag{3.7}$$

Es curioso observar como la reducción de una aplicación sencilla entre dos numerales nos permite computar una operación aritmética relativamente compleja como la exponenciación. Este resultado pudiera parecer también preocupante, si la codificación de la operación n^m es tan sencillo como reducir la aplicación $(\widehat{m}\,\widehat{n})$, ¿Cómo se implementan operaciones mas simples como la suma y la multiplicación?.

En lo que resta de la sección se presentan procedimientos sistemáticos para codificar las operaciones elementales de la aritmética.

3.2.2. Operaciones elementales

En la ecuación (3.7) se muestra como la aplicación de dos numerales se relaciona directamente con la operación de exponenciación. La primera aproximación a la codificación de las operaciones aritméticas seguirá un procedimiento inverso a cómo se dió con este resultado.

Las operaciones que se codificarán son la suma, la multiplicación y la exponenciación. Estas tres operaciones son binarias, es decir, a partir de dos números calculan otro. Para comenzar a codificar este tipo de operaciones consideremos una operación aritmética binaria $\odot$ que realiza algún cálculo.

La convención para nombrar numerales será utilizada para las operaciones aritméticas, por lo tanto $m \odot n$ se codifica como $(\widehat{\odot}\,\widehat{m}\,\widehat{n})$. Ya que $\widehat{\odot}$ es una abstracción que espera ser aplicada a dos numerales, se propone que la operación codificada tenga la forma $\widehat{\odot} \equiv (\lambda \widehat{m}\,\widehat{n}.M)$, donde M es un término que al reducir la aplicación $(\widehat{\odot}\,\widehat{a}\,\widehat{b})$ es equivalente a $(\lambda x\, y.x^{a \odot b}\, y)$.

En el caso de la codificación de la exponenciación, denotada $\widehat{\uparrow}$, se tiene que $(\widehat{\uparrow}\,\widehat{m}\,\widehat{n})$ debe reducirse a

$$\lambda x\, y.x^{m^n}\, y$$

Se puede derivar el término $\widehat{\uparrow}$ observando que

$$\begin{aligned} \lambda x\, y.x^{m^n}\, y &\equiv \widehat{m^n} && \text{Por (3.1)} \\ &=_\beta \widehat{n}\,\widehat{m} && \text{Por (3.7)} \\ &=_\beta \lambda x\, y.\widehat{n}\,\widehat{m}\, x\, y \end{aligned}$$

Por lo tanto, la codificación de la exponenciación es

$$\widehat{\uparrow} \equiv \lambda m\, n.\lambda x\, y.n\, m\, x\, y \tag{3.8}$$

En el caso de la codificación de la multiplicación, denotada $\widehat{\times}$, se tiene que $(\widehat{\times}\,\widehat{m}\,\widehat{n})$ debe reducirse a

$$\lambda x\ y.x^{n\times m}\ y$$

Para derivar el término $\widehat{\times}$ se observa que

$$\begin{aligned}
\lambda x\ y.x^{n\times m}\ y &=_\beta \lambda x\ y.(\widehat{m}\ x)^n\ y && \text{Por (3.5)}\\
&=_\beta \lambda x\ y.\widehat{n}(\widehat{m}\ x)y && \text{Por (3.3)}
\end{aligned}$$

Por lo tanto, la codificación de la multiplicación es

$$\widehat{\times} \equiv \lambda m\ n.\lambda x\ y.n(m\ x)y \tag{3.9}$$

Finalmente, con la operación de adición, denotada $\widehat{+}$, se tiene que $(\widehat{+}\ \widehat{m}\ \widehat{n})$ debe reducirse a

$$\lambda x\ y.x^{n+m}\ y$$

Para derivar el término $\widehat{+}$ se observa que

$$\begin{aligned}
\lambda x\ y.x^{n+m}\ y &\equiv \lambda x\ y.x^n(x^m\ y) && \text{Por (2.3)}\\
&=_\beta \lambda x\ y.x^n(\widehat{m}\ x\ y) && \text{Por (3.3)}\\
&=_\beta \lambda x\ y.\widehat{n}\ x(\widehat{m}\ x\ y) && \text{Por (3.3)}
\end{aligned}$$

Por lo tanto, la codificación de la adición es

$$\widehat{+} \equiv \lambda m\ n.\lambda x\ y.n\ x(m\ x\ y) \tag{3.10}$$

Las codificaciones (3.8), (3.9) y (3.10) fueron construídas a partir de las reducciones mostradas en (3.5), (3.6) y (3.7), las cuales a su vez fueron obtenidas a partir del abuso de notación definido en (2.3) el cual refleja la estructura de los numerales de Church, por lo tanto, las codificaciones mostradas se basan únicamente en la estructura de los numerales. Sin embargo, las operaciones de adición, multiplicación y exponenciación no son únicamente cálculos independientes que permiten expresar expresiones aritméticas. Estas tres operaciones se encuentran conceptualmente relacionadas.

$$\begin{array}{lcrcr}
\widehat{+} & \equiv & \lambda m\ n.\lambda x\ y.n\ x(m\ x\ y) & \equiv & (\lambda m.(\lambda n.(\lambda x.(\lambda y.((n\ x)((m\ x)y))))))\\
\widehat{\times} & \equiv & \lambda m\ n.\lambda x\ y.n(m\ x)y & \equiv & (\lambda m.(\lambda n.(\lambda x.(\lambda y.((n(m\ x))y)))))\\
\widehat{\uparrow} & \equiv & \lambda m\ n.\lambda x\ y.n\ m\ x\ y & \equiv & (\lambda m.(\lambda n.(\lambda x.(\lambda y.(((n\ m)x)y)))))
\end{array}$$

Figura 3.1: Codificaciones de adición, multiplicación y exponenciación

En la figura 3.1 se puede apreciar la diferencia estructural entre las codificaciones definidas, las similitudes se encuentran coloreadas en gris.

La segunda columna muestra las tres codificaciones escritas de manera compacta, se observa que en los tres casos, la aparición de n se encuentra antes que la aparición de m. Debido a que los numerales de

Church son abstracciones, al reducir la aplicación de una operación a dos numerales $\widehat{m}$ y $\widehat{n}$, la estructura del resultado en su forma normal se basará principalmente en la estructura de $\widehat{n}$. Esto no es muy relevante en el caso de $\widehat{+}$ y $\widehat{\times}$ ya que las operaciones son conmutativas, por lo tanto, no es importante si se intercambian las apariciones de n y m, sin embargo, la exponenciación no es una operación conmutativa, el numeral base y el numeral exponente juegan papeles diferentes en la operación.

La tercera columna muestra las tres codificaciones escritas sin abuso de notación, se observa que la aparición de m se agrupa con las variables x y y de manera similar a las ecuaciones (3.1), (3.2) y (3.3), las cuales fueron utilizadas en las ecuaciones (3.5), (3.6) y (3.7).

La figura nos permiten razonar sobre la manera en como se β-reduce la aplicación de las operaciones, sin embargo, las similitudes en la estructura de las codificaciones no refleja las similitudes de las operaciones, por lo que es difícil razonar sobre las operaciones a partir de su definición.

En la sección 3.1, las codificaciones desarrolladas se basaron en una relación fundamental entre los valores de verdad y una desición, esto permitió construir abstracciones componibles que facilitaron codificar y razonar sobre las expresiones booleanas, a tal grado que se estableció una correspondencia directa entre las expreciones condicionales de los lenguajes de programación y las operaciones booleanas.

Para lograr este mismo efecto con la codificación de la aritmética, se deben hacer observaciones más fundamentales sobre la estructura de los numerales de Church y las nociones de las operaciones aritméticas.

Los números naturales nacieron a la par de la necesidad humana de *contar*. De manera similar a la analogía presentada al inicio de la sección 3.1.1 se plantea la siguiente situación hipotética:

Una persona omnisciente y muda llamada P puede decirme la cantidad de objetos en el mundo si le planteo una pregunta con una respuesta contable y le doy un martillo y un clavo; la cantidad de objetos va a corresponder a la cantidad de veces que P golpea el clavo con el martillo. En este planteamiento irreal e hipotético, no es necesario conocer la estructura del número, sólo es necesario tener a alguien que pueda contar y proveer dos objetos sabiendo que la persona va a realizar algo con el primero sobre el segundo (en este caso, golpear con el martillo al clavo). El procedimiento que realiza esta persona puede representar valores numéricos si nunca podemos conocer a los números naturales.

Detrás del concepto de contar, está el concepto de *repetición*, la estructura de los numerales de Church se puede interpretar como la analogía entre repetición y número.

Sea P un término lambda el cual puede ser aplicado a una pregunta Q, al β-reducir $(P\,Q)$ se obtiene una repetición R la cual al ser aplicada a una acción A y un objeto O se β-reduce a realizar la acción A sobre O y repetir el procedimiento con el resultado hasta haber realizado cierta cantidad de acciones:

$$P\,Q \twoheadrightarrow_\beta R,$$

$$R\,A\,O \twoheadrightarrow_\beta \underbrace{A(A(\ldots(A(A\,O))\ldots))}_{n \text{ veces}}$$

Para fines prácticos no es necesario conocer la estructura de P ni de Q, lo importante es que A se realice cierta cantidad de veces sobre O. Por lo tanto R es un término lambda de la forma

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

La cual corresponde a la estructura de los numerales de Church.

Teniendo una justificación conceptual e informal para considerar a los números como repeticiones podemos estudiar las operaciones aritméticas a partir de esta perspectiva.

Al inicio de esta sección se construyeron las codificaciones de las operaciones aritméticas en un orden peculiar. Primero la exponenciación, después la multiplicación y al final la adición. Esto es bastante raro debido a que la exponenciación suele ser considerada una operación más compleja que la multiplicación y a su vez esta más compleja que la adición, la estructura de las codificaciones parece aumentar en complejidad entre menos complejas son las operaciones que describen.

La percepción de complejidad de operaciones aritméticas se remonta a la manera en cómo se enseña la aritmética en la educación básica. Después de aprender a contar, se aprende a sumar y después a multiplicar. A pesar de ser en un inicio un proceso de memorización, el acto de sumar y multiplicar números pequeños termina siendo un acto trivial, empleando algoritmos y heurísticas de estimación cuando los números son grandes. En el caso de la exponenciación, los computólogos suelen adquirir esta misma capacidad cuando se trata de operaciones de la forma 2^n debido a la repetida utilización de números en base 2. Sin embargo, esta percepción tiene también una justificación algorítmica.

La operación de multiplicación puede ser definida en función de la operación de adición. Sean m y n dos números naturales, la operación $m \times n$ es equivalente a sumar m consigo mismo n veces.

$$m \times n = \underbrace{m + m + \ldots + m}_{n \text{ veces}} = \sum_{i=1}^{n} m \tag{3.11}$$

De manera análoga, la operación de exponenciación puede ser definida en función de la operación de multiplicación. Sean m y n dos números naturales, la operación m^n es equivalente a multiplicar m consigo mismo n veces.

$$m^n = \underbrace{m \times m \times \ldots \times m}_{n \text{ veces}} = \prod_{i=1}^{n} m \tag{3.12}$$

De esta manera, una operación compleja como la exponenciación se define en términos de una operación más fundamental como la multiplicación. Esta observación trae a colación la pregunta, ¿Cuál es la operación aritmética más fundamental?.

La respuesta a esta pregunta no es fácil de encontrar, se pudiera pensar que la adición es la operación más fundamental, sin embargo, la adición puede ser definida en función de la operación unaria sucesor y esta a su vez es un caso particular de la adición. Sea $+_1$ el operador unario sucesor

$$m + n = \underbrace{+_1 +_1 \ldots +_1}_{n \text{ veces}} m = \underbrace{1 + 1 + \ldots + 1}_{n \text{ veces}} + m$$

En la segunda aproximación de las codificaciones, se considera que el operador de sucesor es más fundamental que la adición debido a que es fácil codificar la sucesión sin basarse en resultados previos.

La definición de la operación de sucesor consiste en "añadir" una variable x a un número, ya que

$$\widehat{n} \equiv \lambda x\, y.\, \underbrace{x(x(\ldots(x\, y)))}_{n \text{ apariciones de } x}$$

solo se necesita obtener el cuerpo del numeral con la aplicación $(\widehat{n}\ x\ y)$ y aplicar x al resultado de la reducción. La codificación del operador $\widehat{+_1}$ es

$$\widehat{+_1} \equiv \lambda n.\lambda x\ y.x(n\ x\ y) \tag{3.13}$$

Para demostrar que esta definición es correcta, consideremos la reducción de la aplicación de $\widehat{+_1}$ en un numeral cualquiera $\widehat{n}$:

$$\begin{aligned}
\widehat{+_1}\ \widehat{n} &\equiv (\lambda n.\lambda x\ y.x(n\ x\ y))\widehat{n} && \text{Por (3.13)} \\
&\to_\beta \lambda x\ y.x(\widehat{n}\ x\ y) \\
&\twoheadrightarrow_\beta \lambda x\ y.x(x^n\ y) && \text{Por (3.3)} \\
&\equiv \lambda x\ y.x^{n+1}\ y && \text{Por (2.3)} \\
&\equiv \widehat{n+1}
\end{aligned} \tag{3.14}$$

Ahora se debe plantear una manera de aplicar el concepto de repetición de sucesores para obtener la adición. La operación $\widehat{+}$ deberá tomar dos numerales $\widehat{m}$ y $\widehat{n}$ y repetir $\widehat{n}$ veces la operación de sucesor sobre $\widehat{m}$. Ya que la aplicación de $\widehat{+_1}$ a un numeral, resulta en un numeral basta con aplicar $\widehat{n}$ al operador $\widehat{+_1}$ y al numeral $\widehat{m}$. Por ejemplo

$$\begin{aligned}
(\widehat{3}\ \widehat{+_1}\ \widehat{4}) &\twoheadrightarrow_\beta \widehat{+_1}(\widehat{+_1}(\widehat{+_1}\ \widehat{4})) && \text{Por (3.3)} \\
&\twoheadrightarrow_\beta \widehat{+_1}(\widehat{+_1}\ \widehat{5}) && \text{Por (3.14)} \\
&\twoheadrightarrow_\beta \widehat{+_1}\ \widehat{6} && \text{Por (3.14)} \\
&\twoheadrightarrow_\beta \widehat{7} && \text{Por (3.14)}
\end{aligned}$$

En general, para cualesquiera $\widehat{m}$ y $\widehat{n}$, la aplicación $(\widehat{n}\ \widehat{+_1}\ \widehat{m})$ se reduce a:

$$\begin{aligned}
\widehat{n}\ \widehat{+_1}\ \widehat{m} &\twoheadrightarrow_\beta \widehat{+_1}^n\ \widehat{m} && \text{Por (3.3)} \\
&\equiv \widehat{+_1}^{n-1}(\widehat{+_1}\ \widehat{m}) && \text{Por (2.3)} \\
&\twoheadrightarrow_\beta \widehat{+_1}^{n-1}\ \widehat{m+1} && \text{Por (3.14)} \\
&\ldots \\
&\twoheadrightarrow_\beta \widehat{+_1}^{n-n}\ \widehat{m+n} \\
&\equiv \widehat{m+n} && \text{Por (2.3)}
\end{aligned} \tag{3.15}$$

La codificación del operador $\widehat{+}$ es

$$\widehat{+} \equiv \lambda m\ n.n\ \widehat{+_1}\ m \tag{3.16}$$

Para codificar la operación de multiplicación y exponenciación se puede seguir el mismo patrón: un numeral $\widehat{n}$ determina una cantidad de repeticiones, es aplicado a una operación unaria que será aplicada n veces a un término. Hay dos detalles importantes que considerar, primero, cómo convertir una codificación de una operación binaria a unaria y qué valor aplicar al final.

En el caso de la codificación de la multiplicación se debe convertir a $\widehat{+}$ en una operación unaria, por

la definición de multiplicación (3.11) se tiene que

$$m + m + \ldots + m = m + m + \ldots + m + 0 = (m + (m + \ldots + (m + 0)\ldots))$$

Esto es, se repite la aplicación de una abstracción que toma un numeral y computa la suma de el numeral y m, cierta cantidad de veces, comenzando con el numeral $\widehat{0}$. Para construír la versión unaria de $\widehat{+}$ se puede plantear la abstracción:

$$\lambda n.(\widehat{+}\, n\, \widehat{m})$$

Sin embargo hay una manera más conveniente de escribir esta abstracción. Si consideramos la definición (3.16) y la aplicamos únicamente a un numeral, se reduce a

$$\widehat{+}\,\widehat{m} \;\equiv\; (\lambda m\, n.n\, \widehat{+}_1\, m)\widehat{m} \;\to_\beta\; \lambda n.n\, \widehat{+}_1\, \widehat{m}$$

El cual al ser aplicado a algun numeral $\widehat{n}$ será reducido a un término β-convertible a $(\widehat{+}\,\widehat{m}\,\widehat{n})$. Por lo tanto, la aplicación $(\widehat{n}(\widehat{+}\,\widehat{m})\widehat{0})$ computa la multiplicación de $\widehat{m}$ y $\widehat{n}$. Por ejemplo

$$\begin{aligned}
\widehat{3}(\widehat{+}\,\widehat{4})\widehat{0} &\twoheadrightarrow_\beta (\widehat{+}\,\widehat{4})^{\widehat{3}}\widehat{0} && \text{Por (3.3)}\\
&\equiv (\widehat{+}\,\widehat{4})^{\widehat{2}}(\widehat{+}\,\widehat{4}\,\widehat{0}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{+}\,\widehat{4})^{\widehat{2}}\widehat{4} && \text{Por (3.15)}\\
&\equiv (\widehat{+}\,\widehat{4})(\widehat{+}\,\widehat{4}\,\widehat{4}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{+}\,\widehat{4})\widehat{8} && \text{Por (3.15)}\\
&\equiv (\widehat{+}\,\widehat{4})^{0}(\widehat{+}\,\widehat{4}\,\widehat{8}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{+}\,\widehat{4})^{0}\widehat{12} && \text{Por (3.15)}\\
&\equiv \widehat{12} && \text{Por (2.3)}
\end{aligned}$$

En general, para cualesquiera $\widehat{m}$ y $\widehat{n}$, la aplicación $(\widehat{n}(\widehat{+}\,\widehat{m})\widehat{0})$ se reduce a:

$$\begin{aligned}
\widehat{n}(\widehat{+}\,\widehat{m})\widehat{0} &\twoheadrightarrow_\beta (\widehat{+}\,\widehat{m})^{n}\widehat{0} && \text{Por (3.3)}\\
&\equiv (\widehat{+}\,\widehat{m})^{n-1}(\widehat{+}\,\widehat{m}\,\widehat{0}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{+}\,\widehat{m})^{n-1}\widehat{m+0} && \text{Por (3.15)}\\
&\equiv (\widehat{+}\,\widehat{m})^{n-2}(\widehat{+}\,\widehat{m}\,\widehat{m+0}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{+}\,\widehat{m})^{n-2}\widehat{m\times 2+0} && \text{Por (3.15)}\\
&\ldots\\
&\twoheadrightarrow_\beta (\widehat{+}\,\widehat{m})^{n-n}\widehat{m\times n+0}\\
&\equiv \widehat{m\times n} && \text{Por (2.3)}
\end{aligned} \tag{3.17}$$

La codificación del operador $\widehat{\times}$ es

$$\widehat{\times} \;\equiv\; \lambda m\, n.n(\widehat{+}\, m)\widehat{0} \tag{3.18}$$

De manera análoga, se utiliza la definición de exponenciación (3.12) para definir su codificación en función de $\widehat{\times}$. Sean $\widehat{m}$ y $\widehat{n}$ dos numerales cualesquiera, la aplicación $(\widehat{\uparrow}\,\widehat{m}\,\widehat{n})$ debe repetir la multiplicación de la base $\widehat{m}$ una cantidad de veces determinada por el exponente $\widehat{n}$. La codificación es muy similar a la de multiplicación, sólo que utilizando como término final el numeral $\widehat{1}$ ya que $\prod_{i=1}^{n} m = 1 \times \prod_{i=1}^{n} m$. Para corroborar que la aplicación $(\widehat{n}(\widehat{\times}\,\widehat{m})\widehat{1})$ computa la exponenciación de $\widehat{m}$ a la $\widehat{n}$ se desarrolla el siguiente ejemplo

$$
\begin{aligned}
\widehat{3}(\widehat{\times}\,\widehat{4})\widehat{1} &\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{4})^{\widehat{3}}\widehat{1} && \text{Por (3.3)}\\
&\equiv (\widehat{\times}\,\widehat{4})^{\widehat{2}}(\widehat{\times}\,\widehat{4}\,\widehat{1}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{4})^{\widehat{2}}\widehat{4} && \text{Por (3.17)}\\
&\equiv (\widehat{\times}\,\widehat{4})^{\widehat{1}}(\widehat{\times}\,\widehat{4}\,\widehat{4}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{4})^{\widehat{1}}\widehat{16} && \text{Por (3.17)}\\
&\equiv (\widehat{\times}\,\widehat{4})^{\widehat{0}}(\widehat{\times}\,\widehat{4}\,\widehat{16}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{4})^{\widehat{0}}\widehat{64} && \text{Por (3.17)}\\
&\equiv \widehat{64}
\end{aligned}
$$

En general, para cualesquiera $\widehat{m}$ y $\widehat{n}$, la aplicación $(\widehat{n}(\widehat{\times}\,\widehat{m})\widehat{1})$ se reduce a:

$$
\begin{aligned}
\widehat{n}(\widehat{\times}\,\widehat{m})\widehat{1} &\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{m})^{\widehat{n}}\widehat{1} && \text{Por (3.3)}\\
&\equiv (\widehat{\times}\,\widehat{m})^{\widehat{n-1}}(\widehat{\times}\,\widehat{m}\,\widehat{1}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{m})^{\widehat{n-1}}\widehat{m \times 1} && \text{Por (3.17)}\\
&\equiv (\widehat{\times}\,\widehat{m})^{\widehat{n-2}}(\widehat{\times}\,\widehat{m}\,\widehat{m \times 1}) && \text{Por (2.3)}\\
&\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{m})^{\widehat{n-2}}\widehat{m^2 \times 1} && \text{Por (3.17)}\\
&\ldots\\
&\twoheadrightarrow_\beta (\widehat{\times}\,\widehat{m})^{\widehat{n-n}}\widehat{m^n \times 1}\\
&\equiv \widehat{m^n}
\end{aligned}
\tag{3.19}
$$

La codificación del operador $\widehat{\uparrow}$ es

$$
\widehat{\uparrow} \equiv \lambda m\, n.n(\widehat{\times}\,m)\widehat{1} \tag{3.20}
$$

Comparando esta segunda aproximación de las codicicaciones de $\widehat{+}$, $\widehat{\times}$ y $\widehat{\uparrow}$ se pueden observar relaciones tanto en estructura como en significado.

En la tabla 3.2 se puede apreciar la diferencia estrucural entre las codificaciones definidas, las similitudes menos importantes se encuentran coloreadas en gris.

La segunda columna muestra las tres codificaciones escritas de manera compacta, se observa que en los tres casos el átomo n corresponde al operando derecho de la operación y es el que determina la cantidad de veces que se aplicará un procedimiento. Coloreados con azúl se encuentran los valores iniciales a los que se aplica el procedimiento, estos corresponden al caso trivial de la operación, es decir,

$$\begin{array}{lcll} \widehat{+} & \equiv & \lambda m\, n.n\widehat{+}_1 m & \equiv & (\lambda m.(\lambda n.((n\ \widehat{+}_1)m))) \\ \widehat{\times} & \equiv & \lambda m\, n.n(\widehat{+}\, m)\widehat{0} & \equiv & (\lambda m.(\lambda n.((n(\widehat{+}\, m))\widehat{0}))) \\ \widehat{\uparrow} & \equiv & \lambda m\, n.n(\widehat{\times}\, m)\widehat{1} & \equiv & (\lambda m.(\lambda n.((n(\widehat{\times}\, m))\widehat{1}))) \end{array}$$

Figura 3.2: Codificaciones de adición, multiplicación y exponenciación

si n es cero, entonces el resultado de la suma es m, el de la multiplicación es 0 y el de la exponenciación es 1.

En la tercer columna se encuentran las codificaciones escritas sin abuso de notación, coloreado en magenta están los términos a los que n es aplicado primero, esto es, los términos que serán aplicados una y otra vez. Estos términos se encuentran en función de la operación anterior (en orden de menor a mayor complejidad). Al ver las definiciones se puede saber que la expoenciación es repetición de multiplicaciones con caso base 1, la multiplicación es repetición de adiciones con caso base 0 y la adición es repetición de sucesiones con caso base m.

Teniendo codificaciones definidas de manera compacta y elegante, solo queda preguntarnos cómo obtener el resto de las operaciones aritméticas elementales, es decir, la sustracción, la división y para completar las inversas, el logaritmo y la raíz.

Estas operaciones inversas pueden ser vistas de manera similar a la adición, multiplicación y exponenciación, solo que en lugar de añadir aplicaciones, eliminar aplicaciones. Esto se puede lograr con la operación *predecesor*, definida como una operación unaria cerrada en los naturales como

$$-_1 0 = 0$$
$$-_1 n = n - 1$$

La estructura de los numerales de Church favorece los mecanismos que añaden aplicaciones. En la codificación del sucesor fué relativamente sencillo "reenlazar" las variables ligadas en $\widehat{n}$ de tal manera que sólo se necesitaba aplicar x a $(x^n\ y)$ para obtener $(x^{n+1}\ y)$. Sin embargo, para codificar el predecesor es necesario "eliminar" una x de la aplicación y no hay manera sencilla de lograr esto.

Henk Barendreght, en el artículo titulado "The Impact of Lambda Calculus in Logic and Computer Science" [4], menciona que la codificación del predecesor en el cálculo lambda fue un problema abierto. Alonzo Church pudo codificar la adición, la multiplicación y la exponenciación, sin embargo, la función predecesor resultaba ser extremadamente difícil de encontrar con sus numerales.

Stephen Kleene, estudiante de Alonzo Church, encontró la solución de la misteriosa codificación del predecesor. Sin embargo se tuvo que auxiliar de una representación alternativa para los números naturales. De acuerdo a [4], Kleene hizo uso de una codificación de pares de números $\langle n-1, n\rangle$. Se inicia con $\langle 0, 0\rangle$ y el sucesor de $\langle a, b\rangle$ es $\langle b, b+1\rangle$. Cuando Kleene le llevó la propuesta a Church, este ya se había convencido que el cálculo lambda era un sistema demasiado débil para representar el predecesor; es entonces que Church, habiendo aprendido que el predecesor era definible en el cálculo lambda, se convenció de que todas las funciones que eran intuitivamente computables, eran definibles en el cálculo lambda.

Para definir la codificación del predecesor, no se hará uso de la técnica de Kleene, pero si se introducirá

otra notación para los números naturales.

Lo que hace que la estructura de los numerales de Church no sea adecuada es que no se tiene una manera sencilla de remover aplicaciones. Sin embargo, podemos considerar una modificación a los numerales de Church, de tal manera que el mecanismo para quitar y añadir aplicaciones sea sencillo.

Sea $\widehat{n}$ un numeral de Church, su estructura $(\lambda x\ y.x^n\ y)$ se modifica para que una de las apariciones de x se enlace a una variable diferente, por ejemplo z, de tal manera que la cantidad de x sumada a la cantidad de z sea el número representado. Consideremos que el numeral modificado $\widehat{n}'$ tiene la última x de $\widehat{n}$ como z, su definición sería:

$$\widehat{n}' \equiv \lambda x\ y\ z.x^{n-1}(z\ y)$$

En esta nueva codificación, los primeros cinco números son codificados como

$$\begin{aligned}
\widehat{0}' &\equiv \lambda x\ y\ z.y \\
\widehat{1}' &\equiv \lambda x\ y\ z.z\ y \\
\widehat{2}' &\equiv \lambda x\ y\ z.x(z\ y) \\
\widehat{3}' &\equiv \lambda x\ y\ z.x(x(z\ y)) \\
\widehat{4}' &\equiv \lambda x\ y\ z.x(x(x(z\ y)))
\end{aligned}$$

La clave de utilizar esta representación modificada está en observar que es fácil pasar de $\widehat{n}'$ a $\widehat{n-1}$. Sea $\widehat{n}'$ un numeral en la codificación modificada, la reducción de la aplicación $(\widehat{n}'\ x\ y\ \boldsymbol{I})$ resulta en $(x^{n-1}\ y)$.

$$\begin{aligned}
\widehat{n}'\ x\ y\ \boldsymbol{I} &\equiv (\lambda x\ y\ z.x^{n-1}(z\ y))x\ y\ \boldsymbol{I} \\
&\twoheadrightarrow_\beta x^{n-1}(\boldsymbol{I}\ y) \\
&\rightarrow_\beta x^{n-1}\ y
\end{aligned}$$

De tal manera que la definición del predecesor $\widehat{-}_1$ para la codificación de Church puede ser escrita

$$\widehat{-}_1 \equiv \lambda n.\lambda x\ y.\mathcal{T}[n \mapsto n']\ x\ y\ \boldsymbol{I}$$

Donde $\mathcal{T}[n \mapsto n']$ es una transformación que a partir de n obtiene el mísmo número pero con la codificación modificada. Con la cual es sencillo encontrar $n-1$.

Para construír el término $\mathcal{T}[n \mapsto n']$ se debe encontrar una manera de contar desde 0 hasta n en la codificación modificada. Esto se puede lograr utilizando la interpretación de los numerales de Church como operadores de repetición. Si se construye una codificación del sucesor $\widehat{+}'_1$ para los numerales modificados, entonces $\mathcal{T}[n \mapsto n']$ puede ser definida como n aplicaciones de $\widehat{+}'$ con el caso base $\widehat{0}'$, es decir

$$\mathcal{T}[n \mapsto n'] \equiv n\ \widehat{+}'_1\ \widehat{0}'$$

El problema de codificar el predecesor se reduce ahora a la construcción del sucesor de un número con la codificación modificada. Esta construcción resulta ser casi tan sencilla como la codificación de $\widehat{+}_1$.

Primero se analiza cómo cambia la estructura de la codificación de $\widehat{n}'$ a $\widehat{\widehat{n+1}}'$:

$$\begin{array}{lcrcr}
\widehat{+}_1'\,\widehat{0}' & \equiv & \widehat{+}_1'(\lambda x\,y\,z.y) & \twoheadrightarrow_\beta & \lambda x\,y\,z.z\,y \\
\widehat{+}_1'\,\widehat{1}' & \equiv & \widehat{+}_1'(\lambda x\,y\,z.z\,y) & \twoheadrightarrow_\beta & \lambda x\,y\,z.x(z\,y) \\
\widehat{+}_1'\,\widehat{2}' & \equiv & \widehat{+}_1'(\lambda x\,y\,z.x(z\,y)) & \twoheadrightarrow_\beta & \lambda x\,y\,z.x(x(z\,y)) \\
\widehat{+}_1'\,\widehat{3}' & \equiv & \widehat{+}_1'(\lambda x\,y\,z.x(x(z\,y))) & \twoheadrightarrow_\beta & \lambda x\,y\,z.x(x(x(z\,y)))
\end{array}$$

Cuando se computa el sucesor de $\widehat{0}'$, la única variable enlazada es y y el resultado es $z\,y$, por lo tanto, la y será sustituída por $(z\,y)$ en el sucesor. Cuando se computa el sucesor de $\widehat{1}'$, se tienen variables enlazadas z y y, si y es sustituído por $(z\,y)$, la z deberá ser sustituída por x en el sucesor. Cuando se computa el sucesor de $\widehat{2}'$, se tienen variables enlazadas x, z y y, si se suponen las sustituciones de los otros dos casos, la x deberá ser sustituída por x en el sucesor. Para corroborar que estas sustituciones son correctas, se considera la aplicación $(\widehat{3}'\,x\,(z\,y)\,x)$:

$$\begin{aligned}
\widehat{3}'\,x(z\,y)x &\twoheadrightarrow_\beta x^2(x(z\,y)) \\
&\equiv x^3(z\,y)
\end{aligned}$$

Y en general, aplicar estos términos a un numeral modificado $\widehat{n}'$ se reduce a

$$\begin{aligned}
\widehat{n}'\,x(z\,y)x &\twoheadrightarrow_\beta x^{n-1}(x(z\,y)) \\
&\equiv x^n(z\,y)
\end{aligned}$$

Término que corresponde al cuerpo del numeral $\widehat{4}'$. Por lo tanto, la codificación de la operación $\widehat{+}_1'$ es:

$$\widehat{+}_1' \equiv \lambda n'.\lambda x\,y\,z.n'\,x(z\,y)x$$

Con estas piezas, la codificación de la operación predecesor se define como

$$\widehat{-}_1 \equiv \lambda n.\lambda x\,y.(n\,\widehat{+}_1'\,\widehat{0}')x\,y\,\boldsymbol{I} \tag{3.21}$$

Para corroborar que la codificación computa el resultado deseado, consideremos los casos $\widehat{n} \equiv \widehat{0}$ y $\widehat{n} \equiv \widehat{k}$ en la reducción de la aplicación $(\widehat{-}_1\,\widehat{n})$:

$$\begin{aligned}
\widehat{-}_1\,\widehat{0} &\to_\beta \lambda x\,y.(\widehat{0}\,\widehat{+}_1'\,\widehat{0}')x\,y\,\boldsymbol{I} \\
&\twoheadrightarrow_\beta \lambda x\,y.(\boldsymbol{I}\widehat{0}')x\,y\,\boldsymbol{I} \\
&\to_\beta \lambda x\,y.\widehat{0}'\,x\,y\,\boldsymbol{I} \\
&\equiv \lambda x\,y.(\lambda x\,y\,z.y)x\,y\,\boldsymbol{I} \\
&\twoheadrightarrow_\beta \lambda x\,y.y \\
&\equiv \widehat{0}
\end{aligned}$$

$$\begin{aligned}
\widehat{-}_1\,\widehat{k} &\to_\beta \lambda x\,y.(\widehat{k}\,\widehat{+}'_1\,\widehat{0}')x\,y\,\boldsymbol{I}\\
&\twoheadrightarrow_\beta \lambda x\,y.\widehat{+}'^{k}_1\,\widehat{0}'\,x\,y\,\boldsymbol{I}\\
&\twoheadrightarrow_\beta \lambda x\,y.\widehat{k}'\,x\,y\,\boldsymbol{I}\\
&\equiv \lambda x\,y.(\lambda x\,y\,z.x^{k-1}(z\,y))x\,y\,\boldsymbol{I}\\
&\twoheadrightarrow_\beta \lambda x\,y.x^{k-1}(\boldsymbol{I}\,y)\\
&\to_\beta \lambda x\,y.x^{k-1}\,y\\
&\equiv \widehat{k-1}
\end{aligned}$$

Teniendo la codificación del predecesor se puede plantear una codificación de la resta, después de todo, restar n de m es aplicar la función predecesor n veces a m.

$$\widehat{-} \equiv \lambda m\,n.n\,\widehat{-}_1\,m \tag{3.22}$$

Para corroborar que esta definición es correcta, se reduce la aplicación $(\widehat{-}\,\widehat{m}\,\widehat{n})$ la cuál deberá resultar en la codificación del número $m-n$.

$$\begin{aligned}
\widehat{-}\,\widehat{m}\,\widehat{n} &\equiv (\lambda m\,n.n\,\widehat{-}_1\,m)\widehat{m}\,\widehat{n}\\
&\twoheadrightarrow_\beta \widehat{n}\,\widehat{-}_1\,\widehat{m}\\
&\twoheadrightarrow_\beta \widehat{-}^n_1\,\widehat{m}\\
&\equiv \widehat{-}^{n-1}_1\,(\widehat{-}_1\,\widehat{m})\\
&\twoheadrightarrow_\beta \widehat{-}^{n-1}_1\,\widehat{m-1}\\
&\dots\\
&\twoheadrightarrow_\beta \widehat{-}^{n-n}_1\,\widehat{m-n}\\
&\equiv \widehat{m-n}
\end{aligned}$$

La idea de definir operaciones aritméticas complejas en función de otras más sencillas también puede ser aplicada a la definición de la división. Sin embargo, no podemos definir m/n como $(\widehat{n}(\widehat{-}\,\widehat{m})\widehat{0})$ ya que el procedimiento de reducción consistiría en primero calcular $m-0=m$, después calcular $m-m=0$, después calcular $m-0=m$, y así sucesivamente hasta dejar de repetir el procedimiento. En pocas palabras, $(\widehat{n}(\widehat{-}\,\widehat{m})\widehat{0})$ se reduce a $\widehat{m}$ cuando $\widehat{n}$ es impar y a $\widehat{0}$ cuando $\widehat{n}$ es par.

La idea de definir la división como repetición de restas se puede interpretar considerando el siguiente ejemplo. El resultado de dividir a 12 en 3 es 4 porque 4 es la *cantidad* de veces que hay que restarle al 12 el 3 hasta llegar al cero:

$$12-3-3-3-3=12-4\times 3=12-12=0$$

Esto presenta un aumento de complejidad en la codificación de la operación, no solo se debe restar hasta llegar a $\widehat{0}$, también se debe mantener un conteo de la cantidad de veces que se ha restado.

Para el caso de la función logaritmo, el problema es similar a la división. El resultado de calcular el

logaritmo base 3 de 81 es 4 porque 4 es la cantidad de veces que hay que dividir al 81 en 3 hasta llegar al 1 (el caso trivial para la división):

$$(((81/3)/3)/3)/3 = 81/3^4 = 81/81 = 1$$

Para el caso de la función raíz, el problema es aún más grande que con la división y el logaritmo. Ya que $\sqrt[n]{m}$ es calculada como la base a la que debemos elevar por n para obtener m. Poniendo el mismo ejemplo que con el logaritmo, la idea de la raíz es partir del 81 y el 4 y calcular el número al que se tiene que dividir 81, 4 veces, hasta llegar al 1. El problema con esto es que no se puede reducir el problema a uno más sencillo basándonos únicamente en contar y en la operación de división. Sin embargo, es posible utilizar métodos de aproximación y calcular $\sqrt[4]{81}$ intentando divisiones entre 1 y fallar, luego entre 2 y fallar, luego entre 3 y encontrar que es la respuesta.

En la siguiente subsección se construye la manera en la que se podrán plantear mecanismos más complejos de cómputo y lograr definir las operaciones de división, logaritmo y raíz.

3.2.3. Iteración

La técnica de utilizar los números naturales como mecanismo de repetición es de mucha utilidad. En el diseño de algoritmos, la repetición es usualmente representada con mecanismos de *iteración*, usualmente en cada paso de la iteración una o más variables en el contexto del algoritmo cambian, hasta obtener en una de ellas el resultado final.

Por ejemplo, un algoritmo para computar el factorial de un número n puede ser expresado de manera iterativa como:

Algoritmo 3.6 Factorial de n

Entrada: $n \in \mathbb{N}$
Salida: $n!$
  $r \leftarrow copia(n)$
  $a \leftarrow 1$
  mientras $r \neq 0$ **hacer**
    $a \leftarrow a \times r$
    $r \leftarrow r - 1$
  fin mientras
  regresar a

El mecanismo utilizado en este algoritmo para iterar es el *mientras*, éste se acompaña con una condición, si la condición se satisface, el cuerpo del *mientras* es ejecutado, de lo contrario, se detiene la iteración y se prosigue con el resto de los pasos del algoritmo.

Por fortuna, este tipo de algoritmos pueden ser codificados de manera sencilla en el cálculo lambda. En lugar de codificar el *mientras*, se puede utilizar $\widehat{n}$ para repetir un procedimiento, después de todo, el algoritmo inicia con $r = n$ y en cada iteración r disminuye en 1, por lo que se realizan n iteraciones. La variable a inicia en 1 y en cada iteración es multiplicada por r, al terminar los pasos del algoritmo, el valor de a es el resultado $n!$.

La clave para codificar este algoritmo es determinar los valores que representan el cómputo. Las variables r y a describen el estado del cómputo en el caso trivial $n = 0$. Cuando $n \neq 0$, en cada iteración, toda la información del cómputo sigue estando en las variables r y a, aún más, los valores de estas

variables describen una propiedad interesante del algoritmo: Antes y después de cada iteración, se cumple que $n! = a \times r!$.

Ya que $\widehat{n}$ será el mecanismo de iteración, se deben determinar dos cosas: el término que será aplicado repetidas veces P y el término inicial B, de tal manera que $(\widehat{n}\ P\ B)$ se β-reduzca al estado final del cómputo.

El estado del cómputo consiste de la codificación de r y a como numerales de Church. Lo que debemos codificar para representar el algoritmo es una manera de crear un estado a partir de dos números y una manera de obtener el primer y el segundo valor de un estado, es decir, el *constructor* y los *selectores*. Ya que son únicamente dos valores, se pueden utilizar las codificaciones $\boldsymbol{T}$ y $\boldsymbol{F}$ para *decidir* el valor del estado que se desea obtener.

Sean $\widehat{r}$ y $\widehat{a}$ dos numerales, el término $M \equiv \lambda p.p\,\widehat{r}\,\widehat{a}$ puede representar al estado, de tal manera que $(M\,\boldsymbol{T}) \twoheadrightarrow_\beta \widehat{r}$ y $(M\,\boldsymbol{F}) \twoheadrightarrow_\beta \widehat{a}$.

Para construír un estado, basta con codificar un término lambda que al ser aplicado a dos numerales, se reduzca a un término como M. Se define el constructor S de un estado como

$$S \equiv \lambda n_1\, n_2.\lambda p.p\, n_1\, n_2 \tag{3.23}$$

Para seleccionar de un determinado valor a partir de un estado, se plantean abstracciones lambda que al ser aplicadas a un término como M, se reduzcan al valor deseado. Se definen los selectores S_r y S_a de un estado como

$$S_r \equiv \lambda s.s\,\boldsymbol{T} \tag{3.24}$$

$$S_a \equiv \lambda s.s\,\boldsymbol{F} \tag{3.25}$$

Con estas codificaciones, se facilita la escritura de los términos P y B. El estado inicial del cómputo es $r = n$ y $a = 1$, por lo que, conociéndo un numeral $\widehat{n}$, el término B se codifica como

$$B \equiv S\,\widehat{n}\,\widehat{1} \tag{3.26}$$

El término P debe ser una abstracción que sea aplicada a un estado y sea reducida a un estado, ya que en el cálculo lambda no hay una noción de asignación, en cada repetición de la aplicación de P se crea un estado nuevo con sus valores en función del estado anterior. Si $\widehat{r}$ y $\widehat{a}$ son los valores del estado previo, el término P debe reducirse a un estado en donde el primer elemento sea $(\widehat{-_1}\,\widehat{r})$ y el segundo sea $(\widehat{\times}\,\widehat{a}\,\widehat{r})$. Utilizando los selectores S_r y S_a, el término P se codifica como

$$P \equiv \lambda s.S(\widehat{-_1}(S_r\ s))(\widehat{\times}(S_a\ s)(S_r\ s)) \tag{3.27}$$

La codificación completa del algoritmo factorial, utilizando las definiciones (3.23), (3.24) y (3.25), se define

$$\widehat{!} \equiv \lambda n.n(\lambda s.S(\widehat{-_1}(S_r\ s))(\widehat{\times}(S_a\ s)(S_r\ s)))(S\ n\ \widehat{1}) \tag{3.28}$$

Para poder integrar esta definición y componer expresiones algebraicas con las codificaciones de las operaciones elementales, la aplicación de $\widehat{!}$ a un numeral $\widehat{n}$ debe β-reducirse al numeral $\widehat{n!}$, sin embargo, al obtener la forma normal de $(\widehat{!}\,\widehat{n})$ el resultado es un estado equivalente a $(S\,\widehat{0}\,\widehat{1})$ si $\widehat{n} =_\beta \widehat{0}$ o a $(S\,\widehat{1}\,\widehat{n!})$ en otro caso. Por lo tanto $\widehat{!}$ debe estar codificado de tal manera que después de computar el algoritmo,

seleccione el segundo elemento del estado resultante.

$$\widehat{!} \equiv \lambda n.S_a(n(\lambda s.S(\widehat{-}_1(S_r\ s))(\widehat{\times}(S_a\ s)(S_r\ s)))(S\ n\ \widehat{1})) \tag{3.29}$$

Lamentablemente, no siempre es posible expresar algoritmos en donde todo el estado del cómputo se resuma en dos valores. Sin embargo, utilizando una extensión similar a 3.1.3 se puede generalizar la codificación de algoritmos cuyos estados tienen n componentes.

Sean ${}_nv_1, {}_nv_2, \ldots, {}_nv_n$, términos de la forma

$${}_nv_i \equiv \lambda x_1 \ldots x_n.x_i$$

El constructor ${}_nS$ de un estado con n valores se define como

$${}_nS \equiv \lambda x_1 \ldots x_n.\lambda p.p\ x_1 \ldots x_n \tag{3.30}$$

El selector del i-ésimo valor de un estado, se define

$${}_nS_i \equiv \lambda s.s\ v_i \tag{3.31}$$

La codificación de las operaciones aritméticas de división, logaritmo y raíz, serán basadas en algoritmos similares al del factorial. Para tener la habilidad de escribir algoritmos aritméticos, es necesario complementar los numerales de Church con predicados, por ejemplo, para determinar si un numeral es cero o si dos numerales son iguales.

El primer predicado que se define es el que a partir de un numeral $\widehat{n}$, determina si es el $\widehat{0}$. Este predicado es muy utilizado ya que en muchos algoritmos, la condición de paro es cuando un valor numérico tiene el valor cero. A partir de la estructura de $\widehat{n} \equiv (\lambda x\ y.x^n\ y)$ se puede encontrar una manera de aplicarle dos términos P y Q a $\widehat{n}$, tal que al reducirse resulte en $\boldsymbol{T}$ si $n = 0$ y a $\boldsymbol{F}$ si $n > 0$. El $\widehat{0}$ no tiene aplicaciones internas y símplemente es reducido al segundo término al que fue aplicado, por lo tanto $(\widehat{n}\ P\ \boldsymbol{T})$ debe reducirse a $\boldsymbol{T}$ cuando $n = 0$. Cuando el numeral es mayor a cero, la primera aplicación de $(P\ \boldsymbol{T})$ debe reducirse a $\boldsymbol{F}$ y las siguientes aplicaciones deben ser $(P\ \boldsymbol{F})$ y reducirse también a $\boldsymbol{F}$. Un término que al ser aplicado a cualquier término es reducido a $\boldsymbol{F}$ es $(\boldsymbol{K}\ \boldsymbol{F})$. Por lo tanto, la codificación de este predicado se define

$$\widehat{0?} \equiv \lambda n.n(\boldsymbol{K}\ \boldsymbol{F})\boldsymbol{T} \tag{3.32}$$

Para corroborar que este predicado es correcto al ser aplicado a un numeral, se reducen las siguientes aplicaciones:

$$
\begin{aligned}
\widehat{\mathrm{o}}_?\,\widehat{\mathrm{o}} &\equiv (\lambda n.n(\boldsymbol{K}\,\boldsymbol{F})\boldsymbol{T})\widehat{\mathrm{o}} \\
&\to_\beta \widehat{\mathrm{o}}(\boldsymbol{K}\,\boldsymbol{F})\boldsymbol{T} \\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{\mathrm{o}}\boldsymbol{T} \\
&\equiv \boldsymbol{T} \\
\widehat{\mathrm{o}}_?\,\widehat{n} &\equiv (\lambda n.n(\boldsymbol{K}\,\boldsymbol{F})\boldsymbol{T})\widehat{n} \\
&\to_\beta \widehat{n}(\boldsymbol{K}\,\boldsymbol{F})\boldsymbol{T} \\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{n}\boldsymbol{T} \\
&\equiv (\boldsymbol{K}\,\boldsymbol{F})^{n-1}(\boldsymbol{K}\,\boldsymbol{F}\,\boldsymbol{T}) \\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{n-1}\boldsymbol{F} \\
&\equiv (\boldsymbol{K}\,\boldsymbol{F})^{n-2}(\boldsymbol{K}\,\boldsymbol{F}\,\boldsymbol{F}) \\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{n-2}\boldsymbol{F} \\
&\ldots \\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{n-n}\boldsymbol{F} \\
&\equiv \boldsymbol{F}
\end{aligned}
$$

El siguiente predicado que se define es el que a partir de dos numerales $\widehat{m}$ y $\widehat{n}$, determina si $m \leq n$. La codificación de este predicado se basa en la observación de al restar $\widehat{n}$ de $\widehat{m}$ el resultado será $\widehat{\mathrm{o}}$ si $\widehat{m}$ es menor o igual a $\widehat{n}$. Por lo tanto, la codificación de este predicado se define

$$\widehat{\leq}_? \;\equiv\; \lambda m\,n.\widehat{\mathrm{o}}_?(\widehat{-}\,m\,n) \tag{3.33}$$

Ya que estos predicados son reducidos a valores booleanos cuando se aplican a numerales de Church, pueden ser combinados utilizando las operaciones del álgebra booleana. Otros predicados pueden codificarse haciéndo uso de propiedades numéricas.

- Si $n \leq m$, entonces $m \geq n$:

$$\widehat{\geq}_? \;\equiv\; \lambda m\,n.\widehat{\leq}_?\,n\,m \tag{3.34}$$

- Si $m \leq n$ y $m \geq n$, entonces $m = n$:

$$\widehat{=}_? \;\equiv\; \lambda m\,n.\,\boldsymbol{\wedge}\,(\widehat{\leq}_?\,m\,n)(\widehat{\geq}_?\,m\,n) \tag{3.35}$$

- Si $m \not\leq n$, entonces $m > n$:

$$\widehat{>}_? \;\equiv\; \lambda m\,n.\boldsymbol{\neg}(\widehat{\leq}_?\,m\,n) \tag{3.36}$$

- Si $m \not\geq n$, entonces $m < n$:

$$\widehat{<}_? \;\equiv\; \lambda m\,n.\boldsymbol{\neg}(\widehat{\geq}_?\,m\,n) \tag{3.37}$$

A la vez, a partir de estos predicados se pueden definir otros términos muy utilizados en algoritmos aritméticos:

- Si $m < n$, entonces $min(m, n) = m$, de lo contrario $min(m, n) = n$:

$$\widehat{min} \equiv \lambda m\, n.\, \prec (\widehat{\leq}_? \, m \, n) m \, n \tag{3.38}$$

- Si $m > n$, entonces $max(m, n) = m$, de lo contrario $max(m, n) = n$:

$$\widehat{max} \equiv \lambda m\, n.\, \prec (\widehat{>}_? \, m \, n) m \, n \tag{3.39}$$

Para codificaciones de $\widehat{min}$ y $\widehat{max}$ que se reduzcan correctamente al ser aplicados a más de dos numerales, se puede utilizar una técnica como la mostrada en 3.1.6.

Con estos nuevos términos, es más amena la codificación de algoritmos, de hecho, la manera de codificarlos es casi tan sencillo como programar los algoritmos en lenguajes aptos para la programación funcional como `Lisp`, `ML` o `Haskell`.

La estrategia general para la codificación de las operaciones faltantes se basa en la observación de que la sustracción es la función inversa de la adición, la división es la función inversa de la multiplicación y el logaritmo y la raíz son las inversas de la exponenciación. Sean m, n y k tres números naturales

$$\begin{aligned} m - n = x &\iff m = x + n \\ m/n = x &\iff m = x \times n \\ \log_n m = x &\iff m = n^x \\ \sqrt[n]{m} = x &\iff x^n \end{aligned}$$

Estas operaciones deben de ser tratadas con mucho cuidado ya que no son operaciones cerradas, es decir, existen números naturales m y n tal que, para alguna operación $\odot$ de las cuatro mencionadas, $m \odot n$ no es un número natural. Por ejemplo, con la sustracción se pueden calcular números negativos, con la división números racionales, con la raíz números reales y con el logaritmo no únicamente números reales, también el valor $-\infty$. Por lo tanto, las codificaciones que se definen, serán versiones discretas, cerradas y por lo tanto inexactas de las que usualmente se utilizan.

El algoritmo en el que se basan estas cuatro operaciones considera dos números m y n; una operación inversa $\odot$; una condición de trivialidad err; y un valor de trivialidad t. La idea del algoritmo es regresar t cuando $err(m, n)$ es verdadero, de lo contrario iterar a partir de $x = 0$, calculando el resultado $x \odot n$ hasta obtener un valor mayor o igual a m, en donde x será el cálculo de la operación.

Para la sustracción la condición de trivialidad es $m \leq n$ y el valor de trivialidad 0. Para la división la condición de trivialidad es $m < n$ y el valor de trivialidad 1. Para el logaritmo la condición de trivialidad es $m = 0$ y el valor de trivialidad 0. Para la raíz la condición de trivialidad es $n = 0$ y el valor de trivialidad 1.

Una limitación que tiene la iteración en base a los numerales es que se debe conocer la cantidad de veces que se repetirá un proceso, en el caso del algoritmo 3.7, la condición de paro es verificada de manera dinámica, mientras cuando los cálculos se están realizando. Sin embargo, es posible establecer una cota superior a la cantidad de pasos en base a las propiedades de las operaciones establecidas.

En el caso de la sustracción, $m - n$ debe ser un número natural, y ya que la iteración sucede cuando $m > n$, entonces $x \leq m$. En el caso de la división, m/n debe ser un número natural, y ya que la iteración

Algoritmo 3.7 Cálculo de $m \odot^{-1} n$

Entrada: $m,\ n,\ t \in \mathbb{N},\ \odot\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N},\ err\colon \mathbb{N} \times \mathbb{N} \to \{\text{verdadero, falso}\}$
Salida: $m \odot^{-1} n$

```
si err(m, n) entonces
    regresar t
de lo contrario
    x ← 0
    a ← x ⊙ n
    mientras a < m hacer
        a ← x ⊙ n
        x ← x + 1
    fin mientras
    regresar a
fin si
```

sucede cuando $m \geq n$, entonces $x \leq m$. En el caso del logaritmo, $\log_n m$ debe ser un número natural, y ya que la iteración sucede cuando $m \geq n$, entonces $x \leq m$. En el caso de la raíz, ya que m, n y x siempre serán naturales, y la iteración sucede cuando $n \neq 0$, entonces $x \leq m$. Por lo tanto, se puede utilizar a m para determinar la cantidad de iteraciones, de tal manera que cuando la condición $a < m$ se cumpla, siempre se reduzca al valor de a. El algoritmo 3.8 es una modificación de 3.7 para que su codificación sea más directa.

Algoritmo 3.8 Cálculo de $m \odot^{-1} n$

Entrada: $m,\ n,\ t \in \mathbb{N},\ \odot\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N},\ err\colon \mathbb{N} \times \mathbb{N} \to \{\text{verdadero, falso}\}$
Salida: $m \odot^{-1} n$

```
si err(m, n) entonces
    regresar t
de lo contrario
    x ← 0
    a ← x ⊙ n
    para iteraciones hasta m hacer
        si a < m entonces
            a ← x ⊙ n
            x ← x + 1
        de lo contrario
            a ← a
        fin si
    fin para
    regresar a
fin si
```

Para acortar la definición de la codificación del algoritmo de inversa, se definen algunos términos auxiliares:

$$\begin{aligned} \mathbf{:} &\equiv {}_2S \\ \boldsymbol{A} &\equiv {}_2S_1 \\ \boldsymbol{X} &\equiv {}_2S_2 \end{aligned}$$

La codificación del algoritmo 3.8 es

$$\begin{aligned}\widehat{\odot}^{-1} \equiv\ & \lambda \odot\ e\ t.\lambda m\ n.\ \prec (e\ m\ n) \\ & t \\ & (\boldsymbol{X}(m(\lambda s.\ \prec(\widehat{<}_?\ (\boldsymbol{A}\ s)m) \\ & \qquad (:\ (\odot(\boldsymbol{X}\ s)n)(\widehat{+}_1(\boldsymbol{X}s))) \\ & \qquad (:\ (\boldsymbol{A}\ s)(\boldsymbol{X}\ s))) \\ & (:\ (\odot\ \widehat{0}\ n)\widehat{0})))\end{aligned} \tag{3.40}$$

La definición es difícil de leer y comprender si se escribe en un solo renglón, por ello, se utilizaron varios renglones para escribir el término, la convención para escribirlo fué: los saltos de renglón se dan en cada término condicional para que la condición esté en el mismo renglón que $\prec$, el consecuente en el siguiente renglón horizontalmente alineado con la condición y la alternativa en el siguiente renglón horizontalmente alineado con el consecuente. También hay un salto de renglón en la aplicación de m, de tal manera que los dos términos a los que es aplicado estén alineados horizontalmente.

Utilizando el término $\widehat{\odot}^{-1}$ y las condiciones y valores de trivialidad mencionados anteriormente, se definen las codificaciones de las funciones inversas

$$\widehat{-} \equiv \widehat{\odot}^{-1}\ \widehat{+}\ \widehat{\leq}_?\ \widehat{0} \tag{3.41}$$

$$\widehat{/} \equiv \widehat{\odot}^{-1}\ \widehat{\times}\ \widehat{<}_?\ \widehat{1} \tag{3.42}$$

$$\widehat{\log} \equiv \widehat{\odot}^{-1}\ \widehat{\uparrow}(\lambda m\ n.\widehat{0}_?\ m)\widehat{0} \tag{3.43}$$

$$\widehat{\text{root}} \equiv \widehat{\odot}^{-1}(\lambda x\ n.\widehat{\uparrow}\ n\ x)(\lambda m\ n.\widehat{0}_?\ n)\widehat{1} \tag{3.44}$$

El algoritmo 3.8 describe un método de aproximación bastante pobre e ineficiente, sin embargo, es posible codificar otras maneras de calcular estas operaciones utilizando las técnicas vistas hasta el momento, siempre y cuando se establezca una cota superior antes de realizar las iteraciones.

3.2.4. Hiperoperaciones

Las definiciones (3.16), (3.18), (3.20) describen de manera clara y consisa la relación entre la adición, la multiplicación y la exponenciación. Conociendo una cantidad de repeticiones, una operación de agregación y el valor neutro de dicha agregación, fué posible definir una operación aritmética en función de otra operación más simple, hasta tener el caso base con la adición.

En el artículo *Mathematics and Computer Science: Coping with Finiteness* [16], Donald E. Knuth introduce la notación de flecha para expresar números finitos gigantescos. La notación de flecha ya fué utilizada en el término de la exponenciación, ya que, de acuerdo a Knuth, $x \uparrow n = x^n$.

La definición de la notación de flecha es la siguiente:

$$x \uparrow n = x^n$$

$$x \uparrow^k n = x \underbrace{\uparrow \ldots \uparrow}_{k \text{ flechas}} n = \underbrace{(x \overbrace{\uparrow \ldots \uparrow}^{k-1 \text{ flechas}} (x \overbrace{\uparrow \ldots \uparrow}^{k-1 \text{ flechas}} (\cdots \overbrace{\uparrow \ldots \uparrow}^{k-1 \text{ flechas}} x) \cdots))}_{n \text{ veces}}$$

Esta notación introduce una secuencia infinita de operaciones cuya definición es recursiva y consistente con las definiciones de la adición, multiplicación y exponenciación presentadas.

Sea $\widehat{\mathcal{H}(i)}$ el i-ésimo elemento de esta secuencia

$$\begin{aligned} \widehat{\mathcal{H}(1)} &\equiv \widehat{\uparrow} \equiv (\lambda m\, n.n(\widehat{\times} m)\widehat{1}) \\ \widehat{\mathcal{H}(i)} &\equiv \widehat{\uparrow}_i \equiv (\lambda m\, n.n(\widehat{\mathcal{H}(i-1)}\, m)\widehat{1}) \end{aligned} \tag{3.45}$$

El valor neutro siempre es $\widehat{1}$ ya que, para todo $k > 1$

$$x \uparrow^k 1 = x \uparrow^{k-1} 1$$

El primer paso para codificar estas secuencia es generalizar la estructura del término $\widehat{\uparrow}$. Ya que en la definición (3.45) lo único que cambia en la estructura es la operación previa, se puede colocar una variable enlazada f que denote la operación anterior. De esta manera, la codificación de un término que dada una codificación de $\mathcal{H}(i)$ es reducido a $\mathcal{H}(i+1)$ es

$$\lambda f.\lambda m\, n.n(f\, m)\widehat{1}$$

Así, se definen los términos $\widehat{\uparrow}_i$

$$\begin{aligned} \widehat{\uparrow}_1 &\equiv (\lambda f.\lambda m\, n.n(f\, m)\widehat{1})\widehat{\times} \twoheadrightarrow_\beta \lambda m\, n.n(\widehat{\times} m)\widehat{1} \\ \widehat{\uparrow}_2 &\equiv (\lambda f.\lambda m\, n.n(f\, m)\widehat{1})\widehat{\uparrow}_1 \twoheadrightarrow_\beta \lambda m\, n.n(\widehat{\uparrow}_1 m)\widehat{1} \\ \widehat{\uparrow}_3 &\equiv (\lambda f.\lambda m\, n.n(f\, m)\widehat{1})\widehat{\uparrow}_2 \twoheadrightarrow_\beta \lambda m\, n.n(\widehat{\uparrow}_2 m)\widehat{1} \\ &\ldots \\ \widehat{\uparrow}_i &\equiv (\lambda f.\lambda m\, n.n(f\, m)\widehat{1})\widehat{\uparrow}_{i-1} \twoheadrightarrow_\beta \lambda m\, n.n(\widehat{\uparrow}_{i-1} m)\widehat{1} \end{aligned}$$

Este procedimiento es correcto, sin embargo, en caso que se requiera utilizar el término $\widehat{\uparrow}_{1000}$ se deberán de escribir manualmente las definiciones de $\widehat{\uparrow}_{999}$ hasta $\widehat{\uparrow}_1$ lo cuál es inconveniente y tedioso. Con las técnicas que se han desarrollado previamente, se puede construír un término que dado un numeral $\widehat{n}$, se β-reduzca al n-ésimo operador de la secuencia, es decir, codificar $\widehat{\mathcal{H}}$, tal que $(\widehat{\mathcal{H}}\, \widehat{n}) \twoheadrightarrow_\beta \widehat{\mathcal{H}(n)}$.

$$\widehat{\mathcal{H}} \equiv \lambda n.n(\lambda f.\lambda m\, n.n(f\, m)\widehat{1})\widehat{\times} \tag{3.46}$$

Para corroborar que la ecuación (3.46) es correcta, se computan las reducciones

$$
\begin{aligned}
\widehat{\mathscr{H}}\,\widehat{1} &\equiv (\lambda n.n(\lambda f.\lambda m\, n.n(f\ m)\widehat{1})\widehat{\times})\widehat{1} \\
&\to_\beta \widehat{1}(\lambda f.\lambda m\, n.n(f\ m)\widehat{1})\widehat{\times} \\
&\twoheadrightarrow_\beta (\lambda f.\lambda m\, n.n(f\ m)\widehat{1})\widehat{\times} \\
&\to_\beta \lambda m\, n.n(\widehat{\times}\ m)\widehat{1} \\
&\equiv \widehat{\uparrow} \\
&\equiv \widehat{\uparrow}_1
\end{aligned}
$$

$$
\begin{aligned}
\widehat{\mathscr{H}}\,\widehat{i} &\equiv (\lambda n.n(\lambda f.\lambda m\, n.n(f\ m)\widehat{1})\widehat{\times})\widehat{i} \\
&\to_\beta \widehat{i}(\lambda f.\lambda m\, n.n(f\ m)\widehat{1})\widehat{\times} \\
&\twoheadrightarrow_\beta (\lambda f.\lambda m\, n.n(f\ m)\widehat{1})^{i}\widehat{\times} \\
&\equiv (\lambda f.\lambda m\, n.n(f\ m)\widehat{1})^{i-1}((\lambda f.\lambda m\, n.n(f\ m)\widehat{1})\widehat{\times}) \\
&\to_\beta (\lambda f.\lambda m\, n.n(f\ m)\widehat{1})^{i-1}(\lambda m\, n.n(\widehat{\times}\ m)\widehat{1}) \\
&\equiv (\lambda f.\lambda m\, n.n(f\ m)\widehat{1})^{i-1}\widehat{\uparrow}_1 \\
&\ldots \\
&\twoheadrightarrow_\beta (\lambda f.\lambda m\, n.n(f\ m)\widehat{1})^{i-i}\widehat{\uparrow}_i \\
&\equiv \widehat{\uparrow}_i
\end{aligned}
$$

Ejemplos de reducciones concretas no serán dados debido a la naturaleza de la secuencia, tan solo $5 \uparrow\uparrow 3$ tiene 2185 dígitos y $5 \uparrow\uparrow 4$ tiene $1335740483872137 \times 10^{2169}$ dígitos.

3.3. Procesos recursivos

Para complementar el mecanismo de iteración presentado en la sección 3.2.3, se plantea la manera de codificar algoritmos que describen procesos *recursivos*.

La recursividad está detrás de una grán cantidad de definiciones, problemas y algoritmos en matemáticas y ciencias de la computación. La idea básica de la recursividad es plantear un concepto en términos de sí mismo [11].

Un ejemplo de definición recursiva es el de términos lambda. Una aplicación es un término y se compone de otros dos términos, mientras que una abstracción es un término y se compone de un átomo y otro término. Incluso la definición de los números naturales, los cuales nos permiten iterar, es recursiva, ya que $(x^n\ y)$ es definido como un abuso de notación de $(x(x^{n-1}\ y))$.

Hay problemas clásicos cuyas soluciones son también recursivas, por ejemplo, el rompecabezas de *las torres de Hanoi* o el problema de *Flavio Josefo* [11]. Sus soluciones consisten en suponer que el problema ya fue resuelto para una versión más simple y resolver la diferencia del problema simple al original. De esta manera, se reduce el problema hasta llegar a una versión muy simple, en donde la solución es "trivial".

Los algoritmos con definiciones recursivas en algunos casos son más concisos que sus contrapartes iterativas. En particular cuando los algoritmos manipulan estructuras definidas de manera recursiva,

su especificación suele seguir un patrón similar a la de las estructuras que manipula. Por ejemplo, los procedimientos para encontrar los subtérminos de un término lambda o calcular su longitud, por definición (2.1.3,2.1.4), son recursivos.

El mecanismo de iteración presentado en este trabajo está más asociado al concepto de iteración en matemáticas que en computación. La estructura de los numerales de Church, capturan la idea de la aplicación de una *función iterativa* cuyo dominio y rango son el mismo conjunto, de tal manera que estas funciones se pueden componer consigo mismas.

En computación, el concepto de iteración es usualmente asociado a la manera en cómo se expresa un procedimiento; la distinción entre la iteración y otros mecanismos para codificar algoritmos se vuelve entonces en una cuestión lingüística, describiendo así la iteración con palabras como *repetir mientras* o *repetir para*. Al traducir un *programa simbólico* a una secuencia de instrucciones que una máquina abstracta (como la máquina de Turing) o real (como las computadoras) pueda entender, la diferencia entre un procedimiento recursivo y uno iterativo se evapora [3, p. 73] [27].

3.3.1. Procedimientos v.s. procesos

La codificación (3.28) del algoritmo 3.6 cumple con la descripción mencionada de procedimiento recursivo. Se tiene un valor numérico que se desea calcular y para encontrarlo se emplea una abstracción que parte de un estado del cómputo con una solución parcial, esta abstracción sólo debe realizar dos operaciones aritméticas para encontrar otra solución parcial más cercana a la respuesta y delegar el trabajo de computar el resto a otra abstracción que realizará lo mismo.

Un algoritmo que describe de manera más precisa la manera con como (3.28) se reduce es:

Algoritmo 3.9 Procedimiento factorial(n, r, a)

Entrada: $n,\ r,\ a \in \mathbb{N}$, inicialmente $r = n$ y $a = 1$
Salida: $n! = a \times r!$
  si $r = 0$ **entonces**
    regresar a
  de lo contrario
    regresar factorial$(n, r - 1, a \times r)$
  fin si

A pesar de poder escribir un procedimiento iterativo y otro recursivo para el cálculo del factorial, los dos algoritmos describen el mismo *proceso computacional*. Estos procesos no son definidos a partir del lenguaje utilizado para describir el algoritmo, si no a partir de las acciones que se realizan para computar el resultado [2]. Tanto 3.6 como 3.9 como la codificación (3.28) describen el mismo proceso computacional.

Un tercer algoritmo para el cálculo del factorial es 3.10. A pesar de ser expresado como un procedimiento recursivo al igual que el algoritmo 3.9, no describe el mismo proceso computacional.

Los dos procesos mostrados comparten ciertas características:

- Realizan la misma cantidad de multiplicaciones y restas en cada paso iterativo/recursivo;
- Realizan una cantidad de operaciones proporcional al n;
- En cada paso iterativo/recursivo, una solución parcial al problema es calculada.

Algoritmo 3.10 Procedimiento factorial(n)

Entrada: $n \in \mathbb{N}$
Salida: $n!$
  si $n = 0$ **entonces**
    regresar 1
  de lo contrario
    regresar $n \times \text{factorial}(n-1)$
  fin si

La diferencia fundamental entre estos los dos procesos es que en los algoritmos 3.6, 3.9 y en la codificación (3.28), en cada paso se conoce el estado completo del cómputo; mientras que en el algoritmo 3.10, al realizar el paso recursivo, se pierde la información de lo que ya se ha computado.

Así como se distinguen los procedimientos recursivos de los iterativos por la manera en como son expresados. Los procesos también se pueden distinguir en recursivos e iterativos. En general, un proceso iterativo es aquel en donde el estado del cómputo puede ser capturado por una cantidad fija de valores, junto con una regla fija que describe como estos valores evolucionan a lo largo del cómputo. Por otro lado, los procesos recursivos suspenden el cálculo de las operaciones hasta tener todos los valores necesarios para computar el resultado [2].

Desde un aspecto operativo, el proceso recursivo del factorial multiplica una vez que el subproblema ha sido resuelto, mientras que el proceso iterativo del factorial, multiplica conforme los subproblemas son resueltos.

3.3.2. Derivación de un mecanismo de recursividad

Las técnicas para la codificación de algoritmos que se han tratado hasta este punto, sirven para aquellos que describen un proceso iterativo. En esta sección se aborda la manera en la que se pueden codificar algoritmos que describen procesos recursivos. Para desarrollar esta técnica, se considera como ejemplo la definición recursiva de la función factorial:

$$n! = \begin{cases} 1 & n = 0; \\ n \times (n-1)! & n > 0. \end{cases}$$

Todas las componentes de ésta definición están codificadas, ya sea como expresiones booleanas o como expresiones aritméticas. Una pseudo-definición de esta codificación es:

$$\widehat{!} \equiv \lambda n.(\widehat{\prec}\ (\widehat{0?}\ n)\ \widehat{1}\ (\widehat{\times}\ n(\widehat{!}(\widehat{-}_1 n)))) \tag{3.47}$$

El problema con esta definición es que antes de definir $\widehat{!}$, se hace referencia a ella en $(\widehat{!}(\widehat{-}_1 n))$, por lo que no es posible establecer el valor del término factorial antes de terminar de escribir su definición.

Hay algunos trucos que se pueden implementar para simular que se tiene definida la codificación de factorial antes de definirla. Por ejemplo, si $\widehat{!}$ fuera una abstracción la cual espera ser aplicada a sí misma y a un número, pudiera definirse exactamente como (3.47), pero con una variable enlazada f que será

sustituída por el término $\widehat{!}$ dentro de la definición.

$$\begin{aligned}\widehat{!} \equiv\ & \lambda f\, n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\\ & \widehat{1}\\ & (\widehat{\times}\, n(f(\widehat{-_1} n))))\end{aligned} \tag{3.48}$$

El problema con (3.48) es que si se reduce $(\widehat{!\,!}\,\widehat{n})$ y $n \neq \mathrm{o}$, ocurre lo siguiente:

$$\widehat{!\,!}\,\widehat{n} \twoheadrightarrow_\beta (\widehat{\times}\,\widehat{n}(\widehat{!}(\widehat{-_1}\,\widehat{n})))$$

La multiplicación de $\widehat{n}$ debe realizarse con otro numeral, sin embargo, $\widehat{!}$ espera ser aplicado a $\widehat{!}$ y a $(\widehat{-_1}\,\widehat{n})$, sin embargo, en la reducción el término $\widehat{!}$ no es aplicado a sí mismo.

$$(\,\widehat{\times}\,\widehat{n}\,(\,\widehat{!}\ \underbrace{[\ ]}_{\text{Debe de ir }\widehat{!}}\ (\,\widehat{-_1}\,\widehat{n}\,))\,)$$

Para escapar de este problema, se debe de aplicar $(f\ f(\widehat{-_1}\widehat{n}))$ en el cuerpo de la definición. El factorial modificado es

$$\begin{aligned}\widehat{!} \equiv\ & \lambda f\, n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\\ & \widehat{1}\\ & (\widehat{\times}\, n(f\ f(\widehat{-_1} n))))\end{aligned} \tag{3.49}$$

Esta definición se escapa del problema de la definición recursiva, puede reducirse a la codificación de la función factorial aplicando

$$\widehat{!\,!} \twoheadrightarrow_\beta\ \lambda n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\widehat{1}(\widehat{\times}\, n(\widehat{!\,!}(\widehat{-_1} n))))$$

En efecto, no se necesita el nombrar la abstracción con un símbolo como $\widehat{!}$ para expresar la codificación del factorial. La reducción anterior es exactamente la misma a

$$\begin{aligned}&((\lambda f\, n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\\ &\qquad \widehat{1}\\ &\qquad (\widehat{\times}\, n(f\ f(\widehat{-_1} n)))))\\ &\ (\lambda f\, n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\\ &\qquad \widehat{1}\\ &\qquad (\widehat{\times}\, n(f\ f(\widehat{-_1} n))))))\end{aligned} \twoheadrightarrow_\beta \begin{aligned}&(\lambda n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\\ &\qquad \widehat{1}\\ &\qquad (\widehat{\times}\, n\\ &\qquad\ ((\lambda f\, n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\\ &\qquad\qquad \widehat{1}\\ &\qquad\qquad (\widehat{\times}\, n(f\ f(\widehat{-_1} n)))))\\ &\qquad\ (\lambda f\, n.(\prec\ (\widehat{\mathrm{o}_?}\ n)\\ &\qquad\qquad \widehat{1}\\ &\qquad\qquad (\widehat{\times}\, n(f\ f(\widehat{-_1} n)))))\\ &\qquad\ (\widehat{-_1} n)))))\end{aligned}$$

3.3.3. Combinador genérico de recursividad

A pesar de tener este problema resuelto, la solución no es buena. Esta técnica obliga a que en cada algoritmo recursivo que se codifique, cada aplicación recursiva se deba tener el primer argumento aplicado a sí mismo. Lo que se necesita para tener un buen mecanismo de recursividad es separar la auto-aplicación de una codificación recursiva de la definición misma. Lo ideal es poder codificar un algoritmo similar a la definición (3.48) y mediante algún procedimiento genérico, hacer que se auto-aplique el procedimiento a si mismo.

La "factorización" del mecanismo de recursión se puede lograr combinando la idea de la auto-aplicación de la ecuación (3.49) y la idea de escribir los términos recursivos como en la ecuación (3.48).

En (3.48) se esperaba que $\widehat{!}$ sea aplicado a sí mismo, sin embargo en su definición solo aplica f a $(\widehat{-}_1\ n)$, por lo que es conveniente tratar de reducir la aplicación de $\widehat{!}$ a $(\widehat{!}\,\widehat{!})$, de tal manera que la reducción sea

$$\begin{aligned}\widehat{!}(\widehat{!}\,\widehat{!}) &\equiv (\lambda f\ n.(\prec (\widehat{0}_?\ n)\widehat{1}(\widehat{\times}\ n(f(\widehat{-}_1 n)))))(\widehat{!}\,\widehat{!})\\ &\to_\beta \lambda n.(\prec (\widehat{0}_?\ n)\widehat{1}(\widehat{\times}\ n(\widehat{!}\,\widehat{!}(\widehat{-}_1 n))))\end{aligned}$$

Esto hace que la codificación funcione en el primer paso recursivo. Sin embargo, $(\widehat{!}\,\widehat{!}(\widehat{-}_1 n))$ será reducido de tal manera que el siguiente paso recursivo no se aplique $\widehat{!}$ a si mismo.

$$\begin{aligned}\widehat{!}\,\widehat{!}(\widehat{-}_1 n) &\equiv (\lambda f\ n.(\prec (\widehat{0}_?\ n)\widehat{1}(\widehat{\times}\ n(f(\widehat{-}_1 n)))))\widehat{!}(\widehat{-}_1 n)\\ &\to_\beta (\lambda n.(\prec (\widehat{0}_?\ n)\widehat{1}(\widehat{\times}\ n(\widehat{!}(\widehat{-}_1 n)))))(\widehat{-}_1 n)\end{aligned}$$

El término $(\widehat{!}(\widehat{-}_1 n))$ debería de ser $(\widehat{!}\,\widehat{!}(\widehat{-}_1 n))$ de nuevo para que el siguiente paso recursivo funcione.

En el ejemplo 1.1.3 se mostró un término que es útil para este tipo de situaciones:

$$\omega \equiv \lambda x.x\ x$$

La propiedad interesante de ω es que $(\omega\ \omega) \twoheadrightarrow_\beta (\omega\ \omega)$, que es justo lo que deseamos en nuestro mecanismo recursivo, el combinador que se necesita es uno similar a ω, llamado ω' tal que $(\omega'\ \omega') \twoheadrightarrow_\beta (\widehat{!}(\omega'\ \omega'))$.

Suponiendo que $\widehat{!}$ ya es una variable enlazada, el combinador ω' debe ser una abstracción que espera ser aplicada a si misma, tiene la forma $\lambda\omega'.M$. Ya que la reducción de ω' aplicada a sí misma resulta en $(\widehat{!}(\omega'\ \omega'))$, M debe de ser la aplicación $(\widehat{!}\,N)$, para completar la regla de reducción $N \equiv (\omega'\ \omega')$.

$$\omega' \equiv \lambda\omega'.\widehat{!}(\omega'\ \omega') =_\alpha \lambda x.\widehat{!}(x\ x)$$

Para completar la definición del mecanismo de recursividad se plantea un combinador que espere ser aplicado a un término como $\widehat{!}$ de (3.48) e internamente aplique ω' a sí misma. Este mecanismo es llamado combinador $\boldsymbol{Y}$.

$$\boldsymbol{Y} \equiv \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)) \tag{3.50}$$

Al aplicar el combinador $\boldsymbol{Y}$ a la definición (3.48) del factorial, se obtiene que

$$\begin{aligned}\widehat{\boldsymbol{Y}!} &=_\beta \widehat{!(\boldsymbol{Y}!)} \\ &\equiv (\lambda f\, n.(\prec (\widehat{\mathrm{o}_?}\, n)\widehat{1}(\widehat{\times}\, n(f(\widehat{-_1} n)))))(\widehat{\boldsymbol{Y}!}) \\ &\to_\beta \lambda n.(\prec (\widehat{\mathrm{o}_?}\, n)\widehat{1}(\widehat{\times}\, n(\widehat{\boldsymbol{Y}!}(\widehat{-_1} n))))\end{aligned}$$

Al reducir la aplicación de este término en un numeral mayor a cero se obtiene

$$\begin{aligned}&(\lambda n.(\prec (\widehat{\mathrm{o}_?}\, n) \\ &\quad \widehat{1} \qquad\qquad\qquad\qquad \twoheadrightarrow_\beta \quad (\widehat{\times}\, \widehat{n}(\widehat{\boldsymbol{Y}!}(\widehat{-_1}\widehat{n}))) \\ &\quad (\widehat{\times}\, n(\widehat{\boldsymbol{Y}!}(\widehat{-_1} n))))\, \widehat{n}\end{aligned}$$

Es entonces que $(\widehat{\boldsymbol{Y}!})$ puede ser β-convertido nuevamente a $(\widehat{!(\boldsymbol{Y}!)})$ para continuar al siguiente paso recursivo de la misma manera.

3.3.4. Combinadores de punto fijo

Un *punto fijo* de un operador o función es un objeto el cuál no cambia cuando el operador le es aplicado. Por ejemplo, la función $f(x) = x^2$ tiene dos puntos fijos 0 y 1, ya que $0^2 = 0$ y $1^2 = 1$. Hay operadores que no tienen punto fijo, por ejemplo el sucesor de un número, ya que $n + 1 \neq n$ para toda n.

El teorema de punto fijo 3.3.1 es uno de los resultados básicos en el cálculo lambda.

Teorema 3.3.1 (Teorema de punto fijo)**.** $\forall F\, \exists X : F\, X =_\beta X$

Demostración. Sea $W \equiv \lambda x.F(x\, x)$ y $X \equiv W\, W$. Entonces

$$X \equiv (\lambda x.F(x\, x))W \to_\beta F(W\, W) \equiv F\, X$$

□

Esta demostración es algo peculiar ya que se inicia con X y se reduce este término a $(F\, x)$, en lugar de partir hacer el proceso inverso. Sin embargo, la reducción presentada es válida de acuerdo a la definición de β-convertibilidad.

El combinador $\boldsymbol{Y}$ derivado en 3.3.3 pertenece a una clase de combinadores interesantes llamados *combinadores de punto fijo*. Estos combinadores tienen la propiedad de que al ser aplicados a cualquier término F "encuentran" un punto fijo para F [13, p. 34], es decir, si M es un combinador de punto fijo, entonces, para toda F

$$F(M\, F) =_\beta M\, F \tag{3.51}$$

A partir de la ecuación (3.50), se puede corroborar que $\boldsymbol{Y}$ es un combinador de punto fijo de la siguiente manera

$$\begin{aligned} \boldsymbol{Y}\, F &\equiv (\lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x)))F \\ &\to_\beta (\lambda x.F(x\,x))(\lambda x.F(x\,x)) \\ &\to_\beta F((\lambda x.F(x\,x))(\lambda x.F(x\,x))) \\ &=_\beta F(\boldsymbol{Y}\, F) \end{aligned} \tag{3.52}$$

Existen combinadores con una propiedad más fuerte que (3.51). Alan Turing descubrió el combinador $\boldsymbol{\Theta}$ en 1937, su definicón es

$$\boldsymbol{\Theta} \equiv U\,U \quad \text{donde} \quad U \equiv \lambda u\,x.x(u\,u\,x) \tag{3.53}$$

La propiedad interesante de $\boldsymbol{\Theta}$ es que puede computar puntos fijos únicamente con la β-reducción. Se corrobora esto de la siguiente manera

$$\begin{aligned} \boldsymbol{\Theta}\, F &\equiv U\,U\,F \\ &\to_\beta (\lambda x.x(U\,U\,x))F \\ &\to_\beta F(U\,U\,F) \\ &\equiv F(\boldsymbol{\Theta}\, F) \end{aligned} \tag{3.54}$$

Esta propiedad no la tiene $\boldsymbol{Y}$, para realizar el último paso del desarrollo (3.52), se tuvo que hacer una reducción inversa.

La utilidad de los combinadores de punto fijo va más allá que el de permitir la codificación de procesos recursivos. Estos combinadores son especialmente útiles para resolver el siguiente tipo de problema:

Sea Z un término lambda, con variables libres f y $\vec{x}$, encuentra el término F tal que

$$F\,\vec{M} =_\beta Z\left[f := F\right]\left[\vec{x} := \vec{M}\right]$$

En este planteamiento F puede no aparecer en Z y la solución es el mismo término a que si apareciera. Una instancia de este problema puede ser una reformulación de la codificación recursiva del factorial:

Ejemplo 3.3.1. Sea $Z \equiv (\prec (\widehat{\mathrm{o}}?n)\hat{1}(\hat{\times}\, n(f(\hat{-}_1\, n))))$, encuentra el término $\hat{!}$ tal que

$$\hat{!}\,\hat{n} =_\beta Z\left[f := \hat{!}\right]\left[n := \hat{n}\right]$$

Solución. Por la regla (β) se tiene que

$$\hat{!}\,\hat{n} =_\beta (\lambda f\, n.\prec (\widehat{\mathrm{o}}?n)\hat{1}(\hat{\times}\, n(\hat{!}(\hat{-}_1\, n))))\,\hat{!}\,\hat{n}$$

Por la regla (ν) se tiene que

$$\hat{!} =_\beta (\lambda f\, n.\prec (\widehat{\mathrm{o}}?n)\hat{1}(\hat{\times}\, n(\hat{!}(\hat{-}_1\, n))))\,\hat{!}$$

Esta ecuación tiene la forma $A =_\beta (B\,A)$, al considerar a A de la forma $(\boldsymbol{Y}\, B)$, se cumple la propiedad de (3.51) y por lo tanto

$$\hat{!} \equiv \boldsymbol{Y}(\lambda f\, n.\prec (\widehat{\mathrm{o}}?n)\hat{1}(\hat{\times}\, n(f(\hat{-}_1\, n))))$$

□

En general, la solución para este tipo de problemas es

$$F \equiv \mathbf{Y}(\lambda f\, \vec{x}.Z) \tag{3.55}$$

3.4. Estructuras recursivas

Utilizando combinadores de punto fijo como $\mathbf{Y}$ y $\mathbf{\Theta}$ y la solución general (3.55) se pueden codificar procedimientos recursivos como el algoritmo 3.10 del factorial. Sin embargo, los procesos recursivos se tornan más interesantes cuando la información que manipulan es también recursiva. En esta sección se muestran técnicas para codificar estructuras recursivas en el cálculo lambda.

Una variedad de lenguajes de programación, cuentan con un operador o función fundamental para la construcción de estructuras compuestas. Este operador usualmente es llamado *cons*, el cual es una abreviación de la palabra "construír en memoria". Este operador toma dos objetos a y d y construye en memoria un objeto que contiene a ambos.

Matemáticamente, el objeto resultante de aplicar el operador *cons* es un par ordenado. La notación que se utiliza en este trabajo para escribir el par conformado por a y d es $\langle a : d \rangle$.

Un par ordenado no es más que un estado con dos elementos, para codificarse en el cálculo lambda, se deben plantear los mecanismos para construír pares y obtener sus elementos. A continuación se construyen las definiciones del constructor p y los selectores a y d para el par ordenado (basadas en las definiciones (3.30) y (3.31)).

$$p \equiv \lambda a\, d.\lambda q.q\, a\, d \tag{3.56}$$

$$a \equiv \lambda c.c(\lambda a\, d.a) \tag{3.57}$$

$$d \equiv \lambda c.c(\lambda a\, d.d) \tag{3.58}$$

Estas tres ecuaciones cumplen con las reducciones

$$(a(p\, M\, N)) \twoheadrightarrow_\beta M$$

$$(d(p\, M\, N)) \twoheadrightarrow_\beta N$$

para cualesquiera términos lambda M y N. Para corroborar esto, se desarrollan las reducciones:

$$\begin{aligned} a(p\, M\, N) &\equiv (\lambda c.c(\lambda a\, d.a))(p\, M\, N) \\ &\to_\beta (p\, M\, N)(\lambda a\, d.a) \\ &\equiv ((\lambda a\, d.\lambda q.q\, a\, d)\, M\, N)(\lambda a\, d.a) \\ &\twoheadrightarrow_\beta (\lambda q.q\, M\, N)(\lambda a\, d.a) \\ &\to_\beta (\lambda a\, d.a) M\, N \\ &\twoheadrightarrow_\beta M \end{aligned} \tag{3.59}$$

$$\begin{aligned} d(p\,M\,N) &\equiv (\lambda c.c(\lambda a\,d.d))(p\,M\,N) \\ &\to_\beta (p\,M\,N)(\lambda a\,d.d) \\ &\equiv ((\lambda a\,d.\lambda q.q\;a\;d)\,M\,N)(\lambda a\,d.d) \\ &\twoheadrightarrow_\beta (\lambda q.q\,M\,N)(\lambda a\,d.d) \\ &\to_\beta (\lambda a\,d.d)M\,N \\ &\twoheadrightarrow_\beta N \end{aligned} \tag{3.60}$$

En el resto de esta sección se abordan diferentes maneras en las que se puede emplear la estructura del par para construír estructuras más complejas.

3.4.1. Listas

Las listas son secuencias de valores, en donde cada valor en la lista tiene una posición fija. La codificación de las listas en el cálculo lambda se asemeja a la lista enlazada comunmente estudiada en estructuras de datos. Si consideramos que el par ordenado contiene como primer elemento un numeral de Church y como segundo elemento otro par ordenado, se puede representar una lista de números, en donde el último par contiene como segundo elemento una codificación que represente el valor nulo $\emptyset$.

$$\langle \widehat{n}_1 : \langle \widehat{n}_2 : \langle \widehat{n}_3 : \langle \ldots \langle \widehat{n}_k : \emptyset \rangle \ldots \rangle \rangle \rangle \rangle$$

Denotada de manera abreviada como

$$\langle \widehat{n}_1,\ \widehat{n}_2,\ \widehat{n}_3,\ \ldots,\ \widehat{n}_k \rangle$$

La codificación de $\emptyset$ debe elegirse con mucho cuidado. En los algoritmos que manipulan listas es crucial determinar cuando se ha llegado al final de la lista, por lo tanto se hace uso de un predicado para determinar si un determinado objeto es un par o es el valor nulo (de manera similar a la comparación de un número con el cero en los algoritmos aritméticos).

Así como el predicado $\widehat{0}?$ fue construído asumiendo que sería aplicado a un numeral, el término $\emptyset$ se construye asumiendo que será aplicado a una lista de números. Formalmente una lista de numeros, o es un par cuyo primer elemento es un numeral de Church y cuyo segundo elemento es otra lista, o es el término nulo (lista con cero elementos).

$$\mathscr{L} \longrightarrow \langle \widehat{n} : \mathscr{L} \rangle \mid \emptyset \tag{3.61}$$

Por lo tanto, se espera que el predicado $\emptyset?$ sea aplicado a un término de la forma $(\lambda q.q\,\widehat{n}\,\mathscr{L})$ o al término $\emptyset$, de tal manera que

$$\begin{aligned} \emptyset?(\lambda q.q\,\widehat{n}\,\mathscr{L}) &\twoheadrightarrow_\beta \boldsymbol{F} \\ \emptyset?\emptyset &\twoheadrightarrow_\beta \boldsymbol{T} \end{aligned}$$

Una manera de convertir un término par a $\boldsymbol{F}$ es reduciendo la aplicación $(\boldsymbol{K}\,\boldsymbol{F}(\lambda q.q\,\widehat{n}\,\mathscr{L}))$, sin embargo, este resultado es el mismo para cualquier valor al que se aplique $(\boldsymbol{K}\,\boldsymbol{F})$ y el objetivo es poder discriminar entre un par y $\emptyset$. Se puede considerar un término similar a $(\boldsymbol{K}\,\boldsymbol{F})$ pero que cancele los siguientes dos términos a los que sea aplicado:

$$(\lambda x\,y.\boldsymbol{F})M\,N \twoheadrightarrow_\beta \boldsymbol{F}$$

De esta manera, aplicar un par a este término resulta en la reducción

$$\begin{aligned}(\lambda q.q\,\widehat{n}\,\mathscr{L})(\lambda x\,y.\boldsymbol{F}) &\to_\beta (\lambda x\,y.\boldsymbol{F})\widehat{n}\,\mathscr{L}\\ &\twoheadrightarrow_\beta \boldsymbol{F}\end{aligned}$$

Por lo tanto, la codificación de $\emptyset_?$ que es reducida a falso al ser aplicada a un par es

$$\emptyset_? \equiv (\lambda l.l(\lambda x\,y.\boldsymbol{F})) \tag{3.62}$$

Con esta definición, la propiedad que debe cumplir la codificación de $\emptyset$ es $(\emptyset_?\,\emptyset) \twoheadrightarrow_\beta \boldsymbol{T}$, lo cuál resulta ser el término $(\boldsymbol{K}\,\boldsymbol{T})$ ya que

$$\begin{aligned}\emptyset_?\,(\boldsymbol{K}\,\boldsymbol{T}) &\equiv (\lambda l.l(\lambda x\,y.\boldsymbol{F}))(\boldsymbol{K}\,\boldsymbol{T})\\ &\to_\beta \boldsymbol{K}\,\boldsymbol{T}(\lambda x\,y.\boldsymbol{F})\\ &\twoheadrightarrow_\beta \boldsymbol{T}\end{aligned}$$

Por lo tanto

$$\emptyset \equiv \boldsymbol{K}\,\boldsymbol{T} \tag{3.63}$$

Observación (Sobre predicados)**.** La aplicación de un predicado como $\widehat{o}_?$ o $\emptyset_?$ a un término lambda M no necesariamente se β-reduce a una codificación de valor de verdad. Estos predicados son construídos para ser aplicados a numerales o listas respectivamente y el resultado de reducir otro tipo de términos no es de importancia para la codificación de algoritmos.

Con estas codificaciones se pueden construír algoritmos que manipulen listas de números. Consideremos el algoritmo 3.11 que dada una lista de números compute una lista con la misma cantidad de elementos pero con cada número de la lista original multiplicado por 2. En cada paso recursivo se verifica que la lista $\mathscr{L}$ no sea el valor nulo, en cuyo caso, será un par cuyo primer elemento es n y segundo elemento es otra lista $\mathscr{L}'$; se construye otro par con el primer elemento multiplicado por 2 y con el segundo elemento el resultado de realizar este mismo proceso con $\mathscr{L}'$.

El predicado $\emptyset_?$ es utilizado con la condicional booleana $\prec$ sobre la lista $\mathscr{L}$. Cuando la lista sea la codificación de nulo, el resultado es nulo; de lo contrario, se asume que $\mathscr{L}$ es un par y se construye el par correspondiente utilizando los términos p, a y d. La codificación resultante es (3.64).

Algoritmo 3.11 Procedimiento recursivo dobles($\mathscr{L}$)

Entrada: $\mathscr{L} \in \{\langle n_1, \ldots, n_k\rangle \mid n \in \mathbb{N}\} \cup \emptyset$
Salida: $\langle 2 \times n_1, \ldots, 2 \times n_k\rangle$ ó $\emptyset$
  si $\mathscr{L} = \emptyset$ **entonces**
    regresar $\emptyset$
  de lo contrario
    $\langle n : \mathscr{L}'\rangle \leftarrow \mathscr{L}$
    regresar $\langle 2 \times n : \text{dobles}(\mathscr{L}')\rangle$
  fin si

$$\begin{aligned} \mathbf{Y}(\lambda f\, \mathscr{L}.\, \prec (\emptyset_?\, \mathscr{L}) \\ \emptyset \\ (p\ (\widehat{\times}\,\widehat{2}\,(a\, \mathscr{L})) \\ (f\ (d\, \mathscr{L})))) \end{aligned} \tag{3.64}$$

Teniendo esta codificación, es relativamente fácil generalizarla para que a cada número en la lista se le aplique algún término g dado, el cual puede ser reducido a otro número.

$$\begin{aligned} \mathbf{Y}(\lambda f\, g\, \mathscr{L}.\, \prec (\emptyset_?\, \mathscr{L}) \\ \emptyset \\ (p\ (g(a\, \mathscr{L})) \\ (f\ g(d\, \mathscr{L})))) \end{aligned} \tag{3.65}$$

Aún más, se puede generalizar la estructura del resultado al abstraer el término final $\emptyset$ y el constructor del par p de la siguiente manera.

$$\begin{aligned} \mathbf{Y}(\lambda f\, p\, \emptyset\, g\, \mathscr{L}.\, \prec (\emptyset_?\, \mathscr{L}) \\ \emptyset \\ (p\ (g(a\, \mathscr{L})) \\ (f\ p\, \emptyset\, g(d\, \mathscr{L})))) \end{aligned} \tag{3.66}$$

Si $\mathscr{F}$ es la abstracción (3.66), $\mathscr{M}$ es la abstracción (3.65), $\mathscr{D}$ es la abstracción (3.64) y $\mathscr{L}$ es el término lambda que codifica la lista $\langle n_1, \ldots, n_k\rangle$, entonces

$$\mathscr{F}\, p\, \emptyset(\widehat{\times}\,\widehat{2})\mathscr{L} =_\beta \mathscr{M}(\widehat{\times}\,\widehat{2})\mathscr{L} =_\beta \mathscr{D}\, \mathscr{L}$$

Consideremos ahora el algoritmo 3.12 que dada una lista de números compute una lista únicamente con los números de la lista original que son impares. La estructura de este algoritmo es similar al algoritmo 3.11 pero la alternativa de la primer condicional verifica además si el primer elemento n de $\mathscr{L}$ es impar, en cuyo caso n será elemento de un par resultante; de lo contrario n es ignorado y no se

construye un par en este paso recursivo.

Algoritmo 3.12 Procedimiento recursivo impares($\mathscr{L}$)

Entrada: $\mathscr{L} \in \{\langle n_1, \ldots, n_k\rangle \mid n \in \mathbb{N}\} \cup \emptyset$
Salida: $\langle n'_1, \ldots, n'_{k'}\rangle$ ó $\emptyset \mid n'_i \in \mathscr{L},\ n'_i$ impar

```
si 𝓛 = ∅ entonces
    regresar ∅
de lo contrario
    ⟨n : 𝓛'⟩ ← 𝓛
    si n impar entonces
        regresar ⟨n : impares(𝓛')⟩
    de lo contrario
        regresar impares(𝓛')
    fin si
fin si
```

La codificación de este algoritmo es bastante directa también, sin embargo, se necesita codificar un predicado que determine si un numeral de Church es impar o no. Un algoritmo recursivo puede ser elegido para la codificación del predicado impar, sin embargo, es más fácil en este caso utilizar la estructura de los numerales para lograr el resultado deseado. Sea $\widehat{imp_?}$ la codificación del predicado impar, se deben de satisfacer las siguientes reducciones:

$$\begin{aligned}
\widehat{imp_?}(\lambda x\ y.y) &\twoheadrightarrow_\beta \boldsymbol{F}\\
\widehat{imp_?}(\lambda x\ y.x\ y) &\twoheadrightarrow_\beta \boldsymbol{T}\\
\widehat{imp_?}(\lambda x\ y.x(x\ y)) &\twoheadrightarrow_\beta \boldsymbol{F}\\
\widehat{imp_?}(\lambda x\ y.x(x(x\ y))) &\twoheadrightarrow_\beta \boldsymbol{T}\\
&\ldots
\end{aligned}$$

Una manera sencilla de codificar este término es

$$\widehat{imp_?} \equiv \lambda n.n\ \neg\ \boldsymbol{F}$$

De esta manera

$$\begin{aligned}
\widehat{imp_?}(\lambda x\ y.y) &\twoheadrightarrow_\beta \boldsymbol{F}\\
\widehat{imp_?}(\lambda x\ y.x\ y) &\twoheadrightarrow_\beta \neg\ \boldsymbol{F}\\
\widehat{imp_?}(\lambda x\ y.x(x\ y)) &\twoheadrightarrow_\beta \neg(\neg\ \boldsymbol{F})\\
\widehat{imp_?}(\lambda x\ y.x(x(x\ y))) &\twoheadrightarrow_\beta \neg(\neg(\neg\ \boldsymbol{F}))\\
&\ldots
\end{aligned}$$

Con este nuevo predicado, la codificación del algoritmo 3.12 es (3.67).

$$
\begin{aligned}
&\boldsymbol{Y}(\lambda f\, \mathscr{L}.\ \prec (\emptyset_?\, \mathscr{L}) \\
&\qquad \emptyset \\
&\qquad (\prec (\widehat{imp}_?(a\, \mathscr{L})) \\
&\qquad\quad (p(a\, \mathscr{L})(f(d\, \mathscr{L}))) \\
&\qquad\quad (f(d\mathscr{L}))))
\end{aligned}
\tag{3.67}
$$

Esta abstracción también es posible generalizarla con pocas modificaciones para que, además de una lista considere un predicado $g_?$ que sea reducido a una codificación booleana cuando sea aplicado a un numeral de Church. El término (3.68) muestra este término más general.

$$
\begin{aligned}
&\boldsymbol{Y}(\lambda f\, g_?\, \mathscr{L}.\ \prec (\emptyset_?\, \mathscr{L}) \\
&\qquad \emptyset \\
&\qquad (\prec (g_?(a\, \mathscr{L})) \\
&\qquad\quad (p(a\, \mathscr{L})(f\ g_?(d\, \mathscr{L}))) \\
&\qquad\quad (f\ g_?(d\mathscr{L}))))
\end{aligned}
\tag{3.68}
$$

Si $\mathscr{F}$ es la abstracción (3.68), $\mathscr{I}$ es la abstracción (3.67) y $\mathscr{L}$ es el término lambda que codifica la lista $\langle n_1,\ \ldots\ ,\ n_k\rangle$, entonces

$$\mathscr{F}\ \widehat{imp}_?\ \mathscr{L} =_\beta \mathscr{I}\ \mathscr{L}$$

Algoritmos de procesamiento de listas más complejos pueden ser codificados ya sea utilizando (3.65), (3.66), (3.68) o términos con una estructura similar. Algo que es importante notar de las generalizaciones planteadas en los ejemplos es que la lista $\mathscr{L}$ puede tener términos lambda que no sean numerales, al utilizar los algoritmos genéricos sólo se debe tener cuidado con que la forma de los términos en $\mathscr{L}$ sea conocida para g y $g_?$.

3.4.2. Árboles

Los árboles son estructuras no-lineales jerárquicas compuestos de *vértices* (también llamados *nodos*) y *aristas* que establecen una relación entre dos vértices.

Utilizando pares y listas es posible construír árboles. Consideremos el árbol mostrado en la figura 3.3.

La definición usual de un árbol como el mostrado en la figura es como un conjunto de vértices y un conjunto de aristas. Sin embargo, una definición recursiva puede ser expresada como. Un árbol $\mathscr{T}$ puede ser o un vértice v acompañado de una lista de árboles $\mathscr{L}_\mathscr{T}$ o un valor nulo $\emptyset$, es decir

$$\mathscr{T} \longrightarrow (v,\ \langle \mathscr{T}_1,\ \ldots\ ,\ \mathscr{T}_k\rangle) \mid \emptyset$$

Esta definición nos permite codificar árboles como el valor nulo o como un par cuyo primer elemento sea un vértice v y cuyo segundo elemento sea una lista de los vértices en los que v incide. El vértice

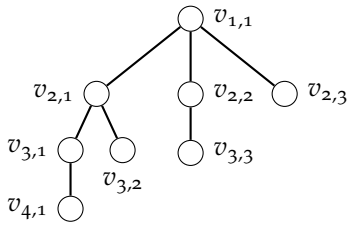


Figura 3.3: Ejemplo de árbol

puede ser representado por cualquier término lambda, en ocaciones es útil asociar un término a cada vértice del árbol para construír, por ejemplo, árboles de numerales.

El árbol de la figura 3.3 se representa de la siguiente manera:

$$\begin{aligned}\langle v_{1,1} : \langle\langle v_{2,1} : \langle\langle v_{3,1} : \langle\langle v_{4,1} : \emptyset\rangle\rangle\rangle\\ \langle v_{3,2} : \emptyset\rangle\rangle\rangle\\ \langle v_{2,2} : \langle\langle v_{3,3} : \emptyset\rangle\rangle\rangle\\ \langle v_{2,3} : \emptyset\rangle\rangle\rangle\end{aligned}$$

La agrupación entre un vértice y sus subárboles se puede realizar con la estructura par, de tal manera que

$$\mathcal{T} \equiv \langle v : \langle \mathcal{T}_1, \ldots, \mathcal{T}_k\rangle\rangle =_\beta \langle v, \mathcal{T}_1, \ldots, \mathcal{T}_k\rangle$$

Por lo que el predicado $\emptyset_?$ funciona para diferenciar árboles nulos de árboles con vértices y si $\mathcal{T}$ es un árbol no nulo, entonces $(a\,\mathcal{T})$ se reduce al vértice del árbol y $(d\,\mathcal{T})$ se reduce a los subárboles del árbol.

Esta similitud en la representación de árboles y listas corresponde con la codificación de listas como "pares de pares" y la codificación de árboles como "listas de listas". Para mostrar las posibilidades que estas relaciones nos permiten se considera el algoritmo 3.13, el cual calcula la cantidad de vértices en un árbol. El procedimiento suma calcula la suma de los elementos de una lista de numeros y el procedimiento map es el que es codificado por (3.65).

Algoritmo 3.13 Procedimiento recursivo cuenta($\mathcal{T}$)

Entrada: $\mathcal{T} \in \{\langle v : \langle \mathcal{T}_1, \ldots, \mathcal{T}_k\rangle\rangle\} \cup \emptyset$
Salida: n = cantidad de vértices en $\mathcal{T}$
  si $\mathcal{T} = \emptyset$ **entonces**
    regresar 0
  de lo contrario
    $\langle v : \mathcal{L}_\mathcal{T}\rangle \leftarrow \mathcal{T}$
    regresar $1 + \text{fold}(+, 0, \text{cuenta}, \mathcal{L}_\mathcal{T})$
  fin si

La codificación del procedimiento fold es el término (3.66). Por lo que el algoritmo 3.13 se define en el cálculo lambda como

$$\begin{aligned}&\boldsymbol{Y}(\lambda f\,\mathcal{T}.\ \prec(\emptyset_?\,\mathcal{T})\\&\qquad\widehat{0}\\&\qquad(\widehat{+}_1(\mathcal{F}\,\widehat{+}\,f(d\,\mathcal{T}))))\end{aligned}$$

3.4.3. Gráficas

Las gráficas son una generalización de árboles en donde se admiten *ciclos*, es decir, a partir de un vértice, se pudieran encontrar dos caminos de aristas para llegar a otro vértice. Consideremos la gráfica mostrada en la figura 3.4.

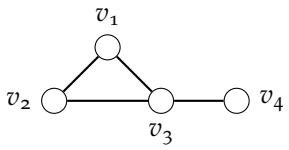


Figura 3.4: Ejemplo de gráfica

Una posible codificación para las gráficas es como *lista de adyacencia*: Una gráfica $\mathcal{G}$ es representada como una lista con un elemento por vértice; cada elemento de esta lista es un par $\langle v : \mathcal{L}_v \rangle$ donde v es un término que representa al vértice y $\mathcal{L}_v$ es una lista con un elemento por vértice en el que v incida; cada elemento de $\mathcal{L}_v$ es un término que representa a un vértice.

La gráfica de la figura 3.4 se representa como lista de adyacencia de la siguiente manera:

$$\begin{aligned}&\langle\langle v_1 : \langle v_2,\ v_3\rangle\rangle\\&\ \langle v_2 : \langle v_1,\ v_3\rangle\rangle\\&\ \langle v_3 : \langle v_1,\ v_2,\ v_4\rangle\rangle\\&\ \langle v_4 : \langle v_3\rangle\rangle\rangle\end{aligned}$$

La única restricción que se debe incorporar a la codificación es que los vértices puedan ser distinguidos entre sí, esto se puede lograr por ejemplo, utilizando numerales de Church para representar vértices.

Ya que las gráficas son codificadas utilizando términos lambda conocidos, los algoritmos que manipular gráficas pueden ser codificados de manera similar a los presentados en las secciones 3.4.2 y 3.4.1.

3.4.4. Términos lambda

En esta sección se aborda el mecanismo mediante el cual es posible codificar términos lambda dentro del cálculo lambda y de esta manera, tener las herramientas para codificar algoritmos que manipulen términos lambda como la sustitución, la α-contracción, la β-contracción, encontrar los subtérminos de un término, etc.

El hecho de poder codificar términos lambda en el cálculo lambda no introduce problema algúno. Este técnica de representar un lenguaje en sí mismo es común, por ejemplo, en compiladores de `C` escritos

en `C` o en intérpretes de `Lisp` escritos en `Lisp`.

Partiendo de la definición 2.1.1 de término lambda, se modifica el conjunto V para que en lugar de ser $\{v_0,\ v_{00},\ ...\}$, sea $\{0,\ 1,\ ...\}$, es decir, que los átomos sean numeros naturales; además en lugar de construír las abstracciones y aplicaciones a partir de símbolos, se construyen a partir de pares.

Definición 3.4.1 (Términos lambda)**.** El conjunto Λ' tiene elementos que son pares y números naturales. Λ' es el conjunto más pequeño que satisface:

$$n \in \mathbb{N} \implies n \in \Lambda' \tag{a}$$
$$M \in \Lambda',\ n \in \mathbb{N} \implies \langle n : M\rangle \in \Lambda' \tag{b}$$
$$M,\ N \in \Lambda' \implies \langle M : N\rangle \in \Lambda' \tag{c}$$

El problema con esta definición de términos lambda es que no es posible distinguir al término $(\lambda x.x)$ de $(x\ x)$. Además, la mayoría de los algoritmos que manipulan términos lambda es crucial poder diferenciar cuando un término M es un átomo, una abstracción o una aplicación. Para arreglar la definición 3.4.1 se pueden utilizar números que etiqueten cada tipo de término, asignando el número 1 a los átomos, el 2 a las abstracciones y el 3 a las aplicaciones, se tiene una correspondencia uno a uno entre Λ y Λ'.

Definición 3.4.2 (Términos lambda)**.** El conjunto Λ' tiene elementos que son pares. Λ' es el conjunto más pequeño que satisface:

$$n \in \mathbb{N} \implies \langle 1 : n\rangle \in \Lambda' \tag{a}$$
$$M \in \Lambda',\ n \in \mathbb{N} \implies \langle 2 : \langle n : M\rangle\rangle \in \Lambda' \tag{b}$$
$$M,\ N \in \Lambda' \implies \langle 3 : \langle M : N\rangle\rangle \in \Lambda' \tag{c}$$

Con esta definición, el término $(\lambda x.x)$ se puede representar como $\langle 2 : \langle\langle 1 : n\rangle : \langle 1 : n\rangle\rangle\rangle$ y el término $(x\ x)$ se puede representar como $\langle 3 : \langle\langle 1 : n\rangle : \langle 1 : n\rangle\rangle\rangle$.

Codificaciones adecuadas para los constructores y selectores de esta representación de términos lambdas se basan en las técnicas abordadas a lo largo de este capítulo.

Sea $\mathcal{T}$ la codificación de un término lambda, los predicados atomo$_?$, abstraccion$_?$ y aplicacion$_?$ permiten determinar la clase de término que es $\mathcal{T}$:

$$\text{atomo}_? \equiv \lambda\mathcal{T}.\widehat{=}_?\,\widehat{1}(a\,\mathcal{T}) \tag{3.71}$$
$$\text{abstraccion}_? \equiv \lambda\mathcal{T}.\widehat{=}_?\,\widehat{2}(a\,\mathcal{T}) \tag{3.72}$$
$$\text{aplicacion}_? \equiv \lambda\mathcal{T}.\widehat{=}_?\,\widehat{3}(a\,\mathcal{T}) \tag{3.73}$$

Utilizando estos tres predicados y el término $\prec$ se puede codificar un término que funcione como una estructura de control similar a $\prec$ que a partir de un término codificado $\mathcal{T}$ y cuatro términos M_1, M_2, M_3 y M_4 compute M_1 si $\mathcal{T}$ es un átomo, M_2 si es una abstracción, M_3 si es una aplicación y M_4 en otro caso:

$$\text{ramifica} \equiv \lambda\mathcal{T}\ x\ y\ z\ e.\ \prec(\text{atomo}_?\,\mathcal{T})x(\prec(\text{abstraccion}_?\,\mathcal{T})y(\prec(\text{aplicacion}_?\,\mathcal{T})z\ e)) \tag{3.74}$$

Los constructores para cada clase de término lambda siguen la definición 3.4.2:

$$\text{atomo} \equiv \lambda n.p\,\widehat{1}\,n \tag{3.75}$$
$$\text{abstraccion} \equiv \lambda n\,x.p\,\widehat{2}(p\,n\,x) \tag{3.76}$$
$$\text{aplicacion} \equiv \lambda x\,y.p\,\widehat{3}(p\,x\,y) \tag{3.77}$$

Los constructores en (3.75) deben satisfacer los predicados en 3.71, se corrobora esto con los siguientes desarrollos.

$$\begin{aligned}
\text{atomo}_?(\text{atomo}\,\widehat{n}) &\equiv (\lambda\mathcal{T}.\widehat{=}_?\,\widehat{1}(a\,\mathcal{T}))(\text{atomo}\,\widehat{n}) \\
&\to_\beta \widehat{=}_?\,\widehat{1}(a(\text{atomo}\,\widehat{n})) \\
&\equiv \widehat{=}_?\,\widehat{1}(a((\lambda n.p\,\widehat{1}\,n)\widehat{n})) \\
&\to_\beta \widehat{=}_?\,\widehat{1}(a(p\,\widehat{1}\,\widehat{n})) \\
&\twoheadrightarrow_\beta \widehat{=}_?\,\widehat{1}\,\widehat{1} \\
&\twoheadrightarrow_\beta \boldsymbol{T}
\end{aligned}$$

$$\begin{aligned}
\text{atomo}_?(\text{aplicacion}\,M\,N) &\equiv (\lambda\mathcal{T}.\widehat{=}_?\,\widehat{1}(a\,\mathcal{T}))(\text{aplicacion}\,M\,N) \\
&\to_\beta \widehat{=}_?\,\widehat{1}(a(\text{aplicacion}\,M\,N)) \\
&\equiv \widehat{=}_?\,\widehat{1}(a((\lambda x\,y.p\,\widehat{3}(p\,x\,y))M\,N)) \\
&\twoheadrightarrow_\beta \widehat{=}_?\,\widehat{1}(a(p\,\widehat{3}(p\,M\,N))) \\
&\twoheadrightarrow_\beta \widehat{=}_?\,\widehat{1}\,\widehat{3} \\
&\twoheadrightarrow_\beta \boldsymbol{F}
\end{aligned}$$

Corroborar que las otras combinaciones de aplicaciones se reducen de manera correcta implica un desarrollo similar.

Los selectores para cada clase de término lambda se codifican en función de la cantidad de elementos que conforman al término. Los átomos tienen un selector, las abstracciones dos y las aplicaciones dos. Debido a que las abstracciones comparten la misma estructura, sus selectores son los mismos.

$$\text{atomo}_n \equiv \lambda x.d\,x \tag{3.78}$$
$$\text{abstraccion}_v \equiv \lambda x.a(d\,x) \tag{3.79}$$
$$\text{abstraccion}_t \equiv \lambda x.d(d\,x) \tag{3.80}$$
$$\text{aplicacion}_o \equiv \lambda x.a(d\,x) \tag{3.81}$$
$$\text{aplicacion}_a \equiv \lambda x.d(d\,x) \tag{3.82}$$

La técnica de etiquetar pares para poder determinar el "tipo" de estructura es muy utilizada en la implementación de lenguajes de programación con verificación dinámica de tipos. La flexibilidad de la técnica permite codificar todos los objetos que se han abordado en este trabajo como pares etiquetados,

los valores de verdad pueden etiquetarse con el 4, los numerales de Church con el 5, los pares con el 6 y así sucesivamente. De esta manera la codificación de algoritmos en el cálculo lambda puede ser más robusta, evitando errores que puedan surgir al reducir los términos.

Apéndice A

Programación de codificaciones

El siguiente código es una implementación literal de los combinadores S, K e I descritos en 2.2.3; los valores de verdad y las operaciones booleanas descritas en 3.1; los numerales de Church y las operaciones aritméticas descritas en 3.2.

Se presentan dos implementaciones, en `Scheme` y en `Haskell`, sin embargo, es sencillo adaptar el código a lenguajes de programación que cuenten con lambdas (también llamadas clausuras).

A.1. Scheme

En este listado de código se ha estilizado la palabra `lambda` con λ.

```
;;
;; Combinadores SKI
;;
(define I (λ (x) x))
(define K (λ (x) (λ (y) x)))
(define S (λ (x) (λ (y) (λ (z) ((x z) (y z))))))

;;
;; Álgebra booleana
;;
(define T K)
(define F (λ (x) (λ (y) y)))
(define IF (λ (p) (λ (m) (λ (n) ((p m) n)))))
(define NOT (λ (p) (((IF p) F) T)))
(define OR (λ (p1) (λ (p2) (((IF p1) T)(((IF p2) T) F)))))
(define AND (λ (p1) (λ (p2) (((IF p1) (((IF p2) T) F) F)))))

;;
;; Aritmética
;;
(define N:0 F)
(define N:0? (λ (n) ((n (K F)) T)))
(define SUCC (λ (n) (λ (x) (λ (y) (x ((n x) y))))))
```

```
(define N:1 (SUCC N:0))
(define N:2 (SUCC N:1))
(define N:3 (SUCC N:2))
(define N:4 (SUCC N:3))
(define + (λ (m) (λ (n) ((n SUCC) m))))
(define * (λ (m) (λ (n) ((n (+ m)) N:0))))
(define ^ (λ (m) (λ (n) ((n (* m)) N:1))))
(define N:0* (λ (x) (λ (y) (λ (z) y))))
(define SUCC* (λ (n) (λ (x) (λ (y) (λ (z) (((n x) (z y)) x))))))
(define PRED (λ (n) (λ (x) (λ (y) (((((n SUCC*) N:0*) x) y) I)))))
(define - (λ (m) (λ (n) ((n PRED) m))))
```

A.2. Haskell

En este listado de código se ha estilizado el caracter \ como λ y la secuencia -> como $\rightarrow$.

```
--
-- Combinadores SKI
--
C_I x = x
C_K x y = x
C_S x y z = x z (y z)

--
-- Álgebra booleana
--
B_T = C_K
B_F x y = y
B_If p m n = p m n
B_Not p = B_If p B_F B_T
B_Or p q = B_If p B_T (B_If q B_T B_F)
B_And p q = B_If p (B_If q B_T B_F) B_F

---
--- Aritmética
---
N_0 = B_F
N_0p n = n (C_K B_F) B_T
N_Succ n = λx y → x (n x y)
N_1 = N_Succ N_0
N_2 = N_Succ N_1
N_3 = N_Succ N_2
N_4 = N_Succ N_3
N_Sum m n = n N_Succ m
N_Mul m n = n (N_Sum m) N_0
N_Exp m n = n (N_Mul m) N_1
N_02 x y z = y
N_Succ2 n = λx y z → n x (z y) x
N_Pred n = λx y → n N_Succ2 N_02 x y C_I
N_Sub m n = n N_Pred m
```

Apéndice B

Intérprete del cálculo lambda

El siguiente código es una implementación de un intérprete del cálculo lambda, incluye un analizador sintáctico, un evaluador y procedimientos específicos para generar código de LaTeX a partir de la representación interna del intérprete.

La implementación está escrita en Racket y se divide en cuatro módulos.

B.1. Módulo principal

```
#lang racket/base

(require racket/match)

(provide
 atom          atom?         atom-symbol
 application   application?  application-applicator  application-applicand
 abstraction   abstraction?  abstraction-argument    abstraction-body
 command       command?      command-name            command-arguments
 assert!
 *terminal-delimiters* *initial-delimiters*
 terminal-delimiter?   initial-delimiter?    delimiter?
 consume-whitespace
 read-number read-identifier read-brackets read-command/atom
 read-λ read-parentheses read-expr
 apply/identity
 *metacommands* *commands*
 eval-expr
 notation-abuse?
 *command-formats*
 fmt:atom-pre fmt:atom-post
 fmt:command-name-pre fmt:command-name-post
 fmt:other-pre fmt:other-post
 fmt:left-paren fmt:right-paren fmt:left-brack fmt:right-brack
 fmt:application-sep fmt:abstraction-sep fmt:command-sep
```

```
 fmt:λ fmt:dot
 format-atom format-application format-abstraction format-command format-expr
 format-list format-other
 print-expr)

(struct atom (symbol) #:transparent)
(struct application (applicator applicand) #:transparent)
(struct abstraction (argument body) #:transparent)
(struct command (name arguments) #:transparent)

(define (assert! ok? who message . formats)
  (if ok? #t (apply error who message formats)))

(define *terminal-delimiters* '(#\} #\] #\) #\. #\, ))

(define *initial-delimiters* '(#\{ #\[ #\( #\λ #\\ ))

(define (terminal-delimiter? ch)
  (or (eof-object? ch) (member ch *terminal-delimiters* char=?)))

(define (initial-delimiter? ch)
  (and (not (eof-object? ch)) (member ch *initial-delimiters* char=?)))

(define (delimiter? ch)
  (or (terminal-delimiter? ch)
      (initial-delimiter? ch)
      (char-whitespace? ch)))

(define (consume-whitespace [in (current-input-port)]
                            [ch (peek-char in)])
  (when (and (char? ch) (char-whitespace? ch))
    (read-char in)
    (consume-whitespace in (peek-char in))))

(define (read-number [in (current-input-port)]
                     [ch (peek-char in)])
  (consume-whitespace in ch)
  (let loop ([lis null]
             [n   (peek-char in)])
    (cond [(and (char? n) (char-numeric? n))
           (read-char in)
           (loop (cons n lis) (peek-char in))]
          [(null? lis)
           (error 'read-number "expecting digit but got `~a'" n)]
          [else
           (assert! (delimiter? n) 'read-number
                    "missing delimiter, got `~a'" n)
```

```
            (string->number (list->string (reverse lis)))])))

  (define (read-identifier [in (current-input-port)]
                           [ch (peek-char in)])
    (consume-whitespace in ch)
    (let loop ([lis null]
               [x   (peek-char in)])
      (cond [(not (delimiter? x))
             (read-char in)
             (loop (cons x lis) (peek-char in))]
            [(null? lis)
             (error 'read-identifier "expecting a non-delimiter but got `~a'" x)]
            [else
             (assert! (delimiter? x) 'read-identifier
                      "missing delimiter, got `~a'" x)
             (string->symbol (list->string (reverse lis)))])))

  (define (read-brackets [in (current-input-port)]
                         [ch (peek-char in)])
    (consume-whitespace in ch)
    (let ([ch (read-char in)])
      (assert! (and (char? ch) (char=? ch #\[)) 'read-brackets
               "expecting `[' but got `~a'" ch))
    (let loop ([lis null]
               [arg (read-expr in)])
      (cond [(null? arg)
             (let ([ch (read-char in)])
               (cond [(eof-object? ch)
                      (error 'read-brackets "missing matching bracket")]
                     [(char=? ch #\,)
                      (loop lis (read-expr in))]
                     [(char=? ch #\])
                      (reverse lis)]
                     [else
                      (error 'read-brackets "missing matching bracket")]))]
            [else
              (loop (cons arg lis) (read-expr in))])))

  (define (read-command/atom [in (current-input-port)]
                             [ch (peek-char in)])
    (define id (read-identifier in ch))
    (consume-whitespace in)
    (let ([ch (peek-char in)])
      (cond [(and (char? ch) (char=? ch #\[))
             (command id (read-brackets in ch))]
            [else
             (atom id)])))
```

```
(define (read-λ [in (current-input-port)]
                [ch (peek-char in)])
  (consume-whitespace in ch)
  (let ([ch (read-char in)])
    (assert! (and (char? ch) (or (char=? ch #\\ ) (char=? ch #\λ))) 'read-λ
             "expecting `λ' or `\\' but got `~a'" ch))
  (let loop ([lis (list (read-command/atom in))]
             [ch  (peek-char in)])
    (cond [(char=? ch #\.)
           (read-char in)
           (foldr abstraction (read-expr in) (reverse lis))]
          [else
           (loop (cons (read-command/atom in) lis) (peek-char in))])))

(define (apply/identity acc x)
  (if (null? acc) x (application acc x)))

(define (read-parentheses [in (current-input-port)]
                          [ch (peek-char in)])
  (consume-whitespace in ch)
  (let ([ch (read-char in)])
    (assert! (and (char? ch) (char=? ch #\( )) 'read-parentheses
             "expecting `(' but got `~a'" ch))
  (let ([expr (read-expr in)])
    (assert! (or (application? expr) (abstraction? expr)) 'read-parentheses
             "expected an application or an abstraction but got ~a" expr)
    (let ([ch (read-char in)])
      (assert! (and (char? ch) (char=? ch #\) )) 'read-parentheses
               "missing matching parentheses"))
    expr))

(define (read-expr [in  (current-input-port)]
                   [ch  (peek-char in)]
                   [acc null])
  (cond [(or (eof-object? ch) (terminal-delimiter? ch))
         acc]
        [(char-whitespace? ch)
         (consume-whitespace in ch)
         (read-expr in (peek-char in) acc)]
        [(char-numeric? ch)
         (read-number in ch)]
        [(char=? ch #\( )
         (let ([expr (read-parentheses in ch)])
           (read-expr in (peek-char in) (apply/identity acc expr)))]
        [(or (char=? ch #\λ) (char=? ch #\\ ))
         (let ([expr (read-λ in ch)])
```

```
            (read-expr in (peek-char in) (apply/identity acc expr)))]
          [else
           (let ([expr (read-command/atom in ch)])
            (read-expr in (peek-char in) (apply/identity acc expr)))]))

(define *metacommands* (make-hasheq))
(define *commands* (make-hasheq))

(define (eval-expr expr)
  (cond [(atom? expr)
         expr]
        [(application? expr)
         (application (eval-expr (application-applicator expr))
                      (eval-expr (application-applicand expr)))]
        [(abstraction? expr)
         (abstraction (eval-expr (abstraction-argument expr))
                      (eval-expr (abstraction-body expr)))]
        [(command? expr)
         (cond [(hash-ref *metacommands* (command-name expr) #f)
                => (λ (proc) (apply proc (command-arguments expr)))]
               [(hash-ref *commands* (command-name expr) #f)
                => (λ (proc) (apply proc (map eval-expr (command-arguments
                    expr))))]
               [else
                (error 'eval-expr "unknown command ~a called with arguments ~a"
                       (command-name expr) (command-arguments expr))])]
        [else
         expr]))

(define fmt:atom-pre (make-parameter ""))
(define fmt:atom-post (make-parameter ""))
(define fmt:application-sep (make-parameter " "))
(define fmt:λ (make-parameter "λ"))
(define fmt:dot (make-parameter "."))
(define fmt:abstraction-sep (make-parameter " "))
(define fmt:command-name-pre (make-parameter ""))
(define fmt:command-name-post (make-parameter ""))
(define fmt:command-sep (make-parameter ","))
(define fmt:other-pre (make-parameter ""))
(define fmt:other-post (make-parameter ""))
(define fmt:left-paren (make-parameter "("))
(define fmt:right-paren (make-parameter ")"))
(define fmt:left-brack (make-parameter "["))
(define fmt:right-brack (make-parameter "]"))
(define notation-abuse? (make-parameter #f))

(define *command-formats* (make-hasheq))
```

```
(define (format-atom symbol)
 (format "~a~a~a" (fmt:atom-pre) symbol (fmt:atom-post)))

(define (format-application applicator applicand)
  (cond
    [(notation-abuse?)
     (format "~a~a~a"
             (if (abstraction? applicator)
                 (format "~a~a~a"
                         (fmt:left-paren)
                         (format-expr applicator)
                         (fmt:right-paren))
                 (format-expr applicator))
             (fmt:application-sep)
             (if (or (application? applicand)
                     (abstraction? applicand))
                 (format "~a~a~a"
                         (fmt:left-paren)
                         (format-expr applicand)
                         (fmt:right-paren))
                 (format-expr applicand)))]
    [else
     (format "~a~a~a~a~a"
             (fmt:left-paren) (format-expr applicator)
             (fmt:application-sep)
             (format-expr applicand) (fmt:right-paren))]))

(define (format-list lis sep)
  (cond [(null? lis) ""]
        [(null? (cdr lis))
         (format-expr (car lis))]
        [else
         (format "~a~a~a"
                 (format-expr (car lis))
                 sep
                 (format-list (cdr lis) sep))]))

(define (format-abstraction argument body)
  (cond [(notation-abuse?)
         (match body
           [(abstraction x M)
            (format-abstraction
             (if (list? argument) (cons x argument) (list x argument)) M)]
           [_
            (format "~a~a~a~a"
                    (fmt:λ)
```

```
                    (if (list? argument)
                        (format-list (reverse argument) (fmt:abstraction-sep))
                        (format-expr argument))
                    (fmt:dot)
                    (format-expr body))])]
          [else
           (format "~a~a~a~a~a~a"
                   (fmt:left-paren) (fmt:λ)
                   (format-expr argument)
                   (fmt:dot)
                   (format-expr body)
                   (fmt:right-paren))]))

(define (format-command name arguments)
  (cond [(hash-ref *command-formats* name #f)
         => (λ (proc)
              (apply proc arguments))]
        [else
         (format "~a~a~a~a~a~a"
                 (fmt:command-name-pre) name (fmt:command-name-post)
                 (fmt:left-brack)
                 (format-list arguments (fmt:command-sep))
                 (fmt:right-brack))]))

(define (format-other other)
  (format "~a~a~a"
          (fmt:other-pre)
          other
          (fmt:other-post)))

(define (format-expr expr)
  (match expr
    [(atom symbol)                      (format-atom symbol)]
    [(application applicator applicand) (format-application applicator applicand)]
    [(abstraction argument body)        (format-abstraction argument body)]
    [(command name arguments)           (format-command name arguments)]
    [x                                  (format-other x)]))

(define (print-expr expr [out (current-output-port)])
  (display (format-expr expr) out))
```

B.2. Módulo comandos

```
#lang racket/base

(require racket/match)
(require racket/list)
```

```
(require racket/set)

(require "λ-common.rkt")

(provide
 install-procedure
 cmd:abuse
 cmd:unabuse
 fmt:fv-name
 fmt:length-pre fmt:length-post
 fmt:subst-assign)

(define (install-procedure table name procedure)
  (hash-set! table name procedure))

(define (cmd:quote expr)
  (match expr
    [(atom symbol)
     expr]
    [(application applicator applicand)
     (application (cmd:quote applicator)
                  (cmd:quote applicand))]
    [(abstraction argument body)
     (abstraction (cmd:quote argument)
                  (cmd:quote body))]
    [(command name arguments)
     (if (eq? name 'u)
         (cmd:unquote (car arguments))
         (command name
                  (map cmd:quote arguments)))]
    [_
     expr]))

(install-procedure *metacommands* 'q cmd:quote)

(define (cmd:unquote expr)
  (eval-expr expr))

(install-procedure *metacommands* 'u cmd:unquote)

(define (cmd:numeral expr)
  (match expr
    [(abstraction (atom symbol1) (abstraction (atom symbol2) body))
     (let loop ([body body])
       (match body
         [(== (atom symbol2))
          0]
```

```
          [(application (== (atom symbol1)) body)
           (+ 1 (loop body))]
          [_
           (error 'numeral "malformed Church encoding")]))]
     [_
      (error 'numeral "malformed Church encoding")]))

(install-procedure *commands* 'numeral cmd:numeral)

(define (cmd:length expr)
  (match expr
    [(atom symbol) 1]
    [(application applicator applicand)
     (+ (cmd:length applicator)
        (cmd:length applicand))]
    [(abstraction argument body)
     (+ 1 (cmd:length body))]
    [_
     (error 'length "can't compute the length of ~a, it's not a λ term" expr)]))

(install-procedure *commands* 'length cmd:length)

(define fmt:length-pre (make-parameter "|"))
(define fmt:length-post (make-parameter "|"))

(define (fmt:length expr)
 (format "~a~a~a"
          (fmt:length-pre)
          (format-expr expr)
          (fmt:length-post)))

(install-procedure *command-formats* 'length fmt:length)

(define (cmd:free-variables expr)
  (match expr
    [(atom symbol)
     (list expr)]
    [(application applicator applicand)
     (remove-duplicates (append (cmd:free-variables applicator)
                                (cmd:free-variables applicand)))]
    [(abstraction argument body)
     (remove argument (cmd:free-variables body))]
    [_
     (error 'length "can't compute the length of ~a, it's not a λ term" expr)]))

(install-procedure *commands* 'fv cmd:free-variables)
```

```
(define fmt:fv-name (make-parameter "FV"))

(define (fmt:free-variables expr)
  (format "~a~a~a~a"
          (fmt:fv-name)
          (fmt:left-paren)
          (format-expr expr)
          (fmt:right-paren)))

(install-procedure *command-formats* 'fv fmt:free-variables)

(define alphabetic-strings
  (map (λ (n) (string (integer->char n)))
       (range (char->integer #\a) (char->integer #\z))))

(define (atom-from-new-symbol binding-symbols)
  (let ([diff (set-subtract (map string->symbol alphabetic-strings)
                            binding-symbols)])
    (if (not (null? diff))
        (atom (car diff))
        (let loop ([i 0])
          (let ([next-symbols (map (λ (str)
                                     (string->symbol (string-append str
                                         (number->string i))))
                                   alphabetic-strings)])
            (let ([diff (set-subtract next-symbols binding-symbols)])
              (if (not (null? diff))
                  (atom (car diff))
                  (loop (+ i 1)))))))))

(define (subst expr old-atom new-expr binding-symbols)
  (match expr
    [(atom symbol)
     (if (equal? expr old-atom)
         new-expr
         expr)]
    [(application applicator applicand)
     (application (subst applicator old-atom new-expr binding-symbols)
                  (subst applicand old-atom new-expr binding-symbols))]
    [(abstraction argument body)
     (cond
       [(or (equal? argument old-atom)
            (not (member old-atom (cmd:free-variables body))))
        expr]
       [(not (member argument (cmd:free-variables new-expr)))
        (abstraction argument
                     (subst body old-atom new-expr
```

```
                          (set-add binding-symbols (atom-symbol argument))))]
       [else
        (let* ([binding-symbols (set-union binding-symbols
                                           (map atom-symbol (cmd:free-variables
                                               new-expr))
                                           (map atom-symbol (cmd:free-variables
                                               body)))]
               [fresh (atom-from-new-symbol binding-symbols)])
          (abstraction fresh
                       (subst (subst body argument fresh
                                     (set-add binding-symbols (atom-symbol
                                         fresh)))
                              old-atom new-expr
                              (set-add binding-symbols (atom-symbol
                                  fresh)))))]))

(define (cmd:subst expr old-atom new-expr)
  (subst expr old-atom new-expr null))

(install-procedure *commands* 'subst cmd:subst)

(define fmt:subst-assign (make-parameter ":="))

(define (fmt:subst expr old-atom new-expr)
  (if (and (notation-abuse?)
           (or (application? expr) (abstraction? expr)))
      (format "~a~a~a~a~a~a~a"
              (fmt:left-paren)
              (format-expr expr)
              (fmt:right-paren)
              (fmt:left-brack)
              (format-expr old-atom)
              (fmt:subst-assign)
              (format-expr new-expr)
              (fmt:right-brack))
      (format "~a~a~a~a~a~a"
              (format-expr expr)
              (fmt:left-brack)
              (format-expr old-atom)
              (fmt:subst-assign)
              (format-expr new-expr)
              (fmt:right-brack))))

(install-procedure *command-formats* 'subst fmt:subst)

(define (cmd:abuse)
  (notation-abuse? #t))
```

```
(install-procedure *metacommands* 'abuse cmd:abuse)

(define (cmd:unabuse)
  (notation-abuse? #f))

(install-procedure *metacommands* 'unabuse cmd:unabuse)

(define (fmt:well-formed expr)
  (let ([state (notation-abuse?)])
    (cmd:unabuse)
    (let ([str (format-expr expr)])
      (notation-abuse? state)
      str)))

(install-procedure *command-formats* 'wf fmt:well-formed)

(define (cmd:well-formed expr)
  (cmd:quote (command 'wf (list expr))))

(install-procedure *commands* 'wf cmd:well-formed)

(define (fmt:mal-formed expr)
  (let ([state (notation-abuse?)])
    (cmd:abuse)
    (let ([str (format-expr expr)])
      (notation-abuse? state)
      str)))

(install-procedure *command-formats* 'mf fmt:mal-formed)

(define (cmd:mal-formed expr)
  (cmd:quote (command 'mf (list expr))))

(install-procedure *commands* 'mf cmd:mal-formed)

(define (fmt:hole)
  (format "~a \\quad ~a"
          (fmt:left-brack)
          (fmt:right-brack)))

(install-procedure *command-formats* 'hole fmt:hole)

(define (cmd:fill expr1 expr2)
  (match expr1
    [(atom symbol)
     expr1]
```

```
    [(application applicator applicand)
     (application (cmd:fill applicator expr2)
                  (cmd:fill applicand expr2))]
    [(abstraction argument body)
     (abstraction argument (cmd:fill body expr2))]
    [(command name arguments)
     (if (eq? name 'hole)
         expr2
         expr1)]
    [_
     expr1]))

(install-procedure *commands* 'fill cmd:fill)

(define (fmt:fill expr1 expr2)
  (if (and (notation-abuse?)
           (or (abstraction? expr1) (application? expr1)))
      (format "~a~a~a~a~a~a"
              (fmt:left-paren)
              (format-expr expr1)
              (fmt:right-paren)
              (fmt:left-brack)
              (format-expr expr2)
              (fmt:right-brack))
      (format "~a~a~a~a"
              (format-expr expr1)
              (fmt:left-brack)
              (format-expr expr2)
              (fmt:right-brack))))

(install-procedure *command-formats* 'fill fmt:fill)
```

B.3. Módulo interfaz

```
#lang racket/base

(require racket/cmdline)

(require "λ-common.rkt")
(require "λ-commands.rkt")

(define (repl)
  (parameterize ([current-prompt-read
                  (λ ()
                    (display "λ> ")
                    (flush-output)
                    (let ([str (read-line)])
```

```
                    (if (or (eof-object? str) (string=? str "exit"))
                        eof
                        (read-expr (open-input-string str)))))]
                [current-eval
                 (λ (expr)
                   (eval-expr (cdr expr)))]
                [current-print
                 (λ (expr)
                   (print-expr expr)
                   (newline))])
    (display "REPL ready, press any key to start...")
    (flush-output)
    (read-line)
    (read-eval-print-loop)))

(define console-args (command-line #:program "texλ"))

(repl)
```

B.4. Módulo TEX

```
#lang racket/base

(require racket/system)
(require racket/file)
(require racket/cmdline)
(require racket/list)

(require "λ-common.rkt")
(require "λ-commands.rkt")

(fmt:left-paren "\\left( ")
(fmt:right-paren " \\right)")
(fmt:left-brack "\\left[ ")
(fmt:right-brack " \\right]")
(fmt:command-name-pre "\\texttt{")
(fmt:command-name-post "}")
(fmt:application-sep "\\ ")
(fmt:abstraction-sep "\\ ")
(fmt:command-sep "\\texttt{,}")
(fmt:λ " \\λ ")
(fmt:dot ".")
(fmt:fv-name "\\mathrm{FV}")
(fmt:length-pre "\\left\\|")
(fmt:length-post "\\right\\|")

(define (fmt:bold expr)
```

```
    (format "\\boldsymbol{~a}"
            (format-expr expr)))

  (install-procedure *command-formats* 'bold fmt:bold)

  (define (fmt:context expr)
    (if (and (notation-abuse?)
             (or (abstraction? expr) (application? expr)))
        (format "~a~a~a~a \\quad ~a"
                (fmt:left-paren)
                (format-expr expr)
                (fmt:right-paren)
                (fmt:left-brack)
                (fmt:right-brack))
        (format "~a~a \\quad ~a"
                (format-expr expr)
                (fmt:left-brack)
                (fmt:right-brack))))

  (install-procedure *command-formats* 'context fmt:context)

  (define (fmt:prime expr n)
    (format "~a^{~a}"
            (format-expr expr)
            (make-string n #\')))

  (install-procedure *command-formats* 'prime fmt:prime)

  (define (fmt:subscript expr sub)
    (format "~a_{~a}"
            (format-expr expr)
            (format-expr sub)))

  (install-procedure *command-formats* 'subscript fmt:subscript)

  (define (fmt:superscript expr sup)
    (format "~a^{~a}"
            (format-expr expr)
            (format-expr sup)))

  (install-procedure *command-formats* 'superscript fmt:superscript)

  (define (fmt:seq expr)
    (format "\\vec{~a}"
            (format-expr expr)))

  (install-procedure *command-formats* 'seq fmt:seq)
```

```
(define (fmt:dots pos . exprs)
  (let ([head (take exprs pos)]
        [tail (drop exprs pos)])
    (format-list (append head (list "...") tail) ",")))

(install-procedure *command-formats* 'dots fmt:dots)

(define (fmt:subterm expr1 expr2)
  (format "~a \\subset ~a"
          (format-expr expr1)
          (format-expr expr2)))

(install-procedure *command-formats* 'subterm fmt:subterm)

(define (fmt:subterms expr)
  (format "\\mathrm{Sub}\\left( ~a \\right)"
          (format-expr expr)))

(install-procedure *command-formats* 'subterms fmt:subterms)

(define (fmt:left-apply sup expr1 expr2)
  (format-application
   (command 'superscript (list expr1 sup))
   expr2))

(install-procedure *command-formats* 'left-apply fmt:left-apply)

(define (fmt:right-apply sup expr1 expr2)
  (format-application
   expr1
   (command 'superscript (list expr2 (format "\\sim ~a" (format-expr sup))))))

(install-procedure *command-formats* 'right-apply fmt:right-apply)

(define cache-dir
  (make-parameter "./λ-cache/"))

(define (file-extension name ext)
  (string-append name ext))

(define (cache-path filename)
  (string-append (cache-dir) filename))

(define (tex-before-curly in out)
  (let loop [(c   (peek-char in))
             (lis null)]
```

```
      (if (char=? c #\})
          (list->string (reverse lis))
          (begin
            (write-char (read-char in) out)
            (loop (peek-char in) (cons c lis))))))

  (define (tex-process-file filename)
    (call-with-input-file filename
      (λ (in)
        (call-with-output-file (cache-path filename)
          (λ (out)
            (let loop ([braces? #f])
              (cond
                [(eof-object? (peek-char in)) #t]
                [(string=? "\\lc{" (peek-string 4 0 in))
                 (read-string 4 in)
                 (print-expr (eval-expr (read-expr in)) out)
                 (loop #t)]
                [(string=? "\\lc*{" (peek-string 5 0 in))
                 (read-string 5 in)
                 (cmd:abuse)
                 (print-expr (eval-expr (read-expr in)) out)
                 (cmd:unabuse)
                 (loop #t)]
                [(char=? (peek-char in) #\})
                 (if braces? (read-char in) (write-char (read-char in) out))
                 (loop #f)]
                [(string=? "\\input{" (peek-string 7 0 in))
                 (write-string (read-string 7 in) out)
                 (let ([ref (tex-before-curly in out)])
                   (unless (file-exists? (string-append "./λ-cache/" ref ".tex"))
                     (tex-process-file (string-append ref ".tex"))))
                 (loop braces?)]
                [(string=? "\\bibliography{" (peek-string 14 0 in))
                 (write-string (read-string 14 in) out)
                 (let ([bib (tex-before-curly in out)])
                   (unless (file-exists? (string-append "./λ-cache/" bib ".bib"))
                     (copy-file (string-append bib ".bib")
                                (string-append "./λ-cache/" bib ".bib"))))
                 (loop braces?)]
                [else
                 (write-char (read-char in) out)
                 (loop braces?)])))))))

  (define (build-tex-draft entry)
    (system (format "pdflatex -draftmode -interaction=batchmode ~a >/dev/null"
        entry)))
```

```
(define (build-tex-bib entry)
  (system (format "bibtex ~a >/dev/null" entry)))

(define (build-tex-complete entry)
  (system (format "pdflatex -interaction=batchmode -shell-escape ~a >/dev/null"
      entry)))

(define (build-tex-document entry)
  (current-directory (cache-dir))
  (build-tex-draft entry)
  (build-tex-bib entry)
  (build-tex-draft entry)
  (build-tex-complete entry))

(define (texλ entry)
  (when (directory-exists? (cache-dir))
    (delete-directory/files (cache-dir)))
  (make-directory (cache-dir))
  (tex-process-file (file-extension entry ".tex"))
  (build-tex-document entry))

(define console-args
  (command-line #:program "texλ" #:args (entry) entry))

(texλ console-args)
```

Bibliografía

[1] A. S. Troelstra, H. S. *Basic Proof Theory*, 2nd ed. Cambridge Tracts in Theoretical Computer Science 43. Cambridge University Press, 2000.

[2] Abelson, H., and Sussman, G. J. *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, Cambridge, MA, USA, 1996.

[3] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[4] Barendregt, H. The impact of the lambda calculus in logic and computer science, 1997.

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

[6] Bridges, D., and Palmgren, E. Constructive mathematics. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., winter 2013 ed. 2013.

[7] Cardone, F., and Hindley, J. R. Lambda-calculus and combinators in the 20th century. In *Logic from Russell to Church*, D. M. Gabbay and J. Woods, Eds., vol. 5 of *Handbook of the History of Logic*. North-Holland, 2009, pp. 723 – 817.

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

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

[10] Crossley, J. N. *Reminiscences of logicians*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975, pp. 1–62.

[11] Graham, R. L., Knuth, D. E., and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[12] Haskell Brooks Curry, Robert Feys, W. C. *Combinatory Logic, Volume I*. Studies in Logic and the Foundations of Mathematics 22. North-Holland Publishing Company, 1958.

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

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

[15] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[16] Knuth, D. E. Mathematics and computer science: Coping with finiteness. *Science 194*, 4271 (1976), 1235–1242.

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

[18] Mendelson, E. *Introduction to Mathematical Logic*, 5 ed. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2010.

[19] Milner, R. Logic for computable functions – description of a machine implementation. Tech. Rep. CS-TR-72-288, Stanford University, Department of Computer Science, May 1972.

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

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

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

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

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

[25] Scott, D. S. Lambda calculus then and now. In *ACM Turing Centenary Celebration* (2012), ACM-TURING '12, ACM.

[26] Statman, R. The typed λ-calculus is not elementary recursive. In *Foundations of Computer Science, 1977., 18th Annual Symposium on* (Oct 1977), pp. 90–94.

[27] Steele, Guy Lewis, J. Debunking the 'expensive procedure call' myth, or, procedure call implementations considered harmful, or, lambda: The ultimate goto. Tech. Rep. AIM-443, Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, October 1977.

[28] Sussman, G. J., and Steele Jr, G. L. SCHEME : an interpreter for extended lambda calculus. Tech. Rep. AI 349, Massachussetts Institute of Technology (MIT). Cambridge (MA US), 1975.