

El cálculo λ y los fundamentos de la computación

Eduardo Acuña Yeomans

2016

Índice general

Para revisores	1
Introducción	3
1 Noción informal del cálculo λ	5
1.1 Notación funcional	5
1.2 Expresiones	7
1.2.1 Átomos	7
1.2.2 Abstracciones	7
1.2.3 Aplicaciones	8
1.2.4 Extensiones	8
1.2.5 Ejemplos	9
1.3 Operaciones	10
1.3.1 Sustitución	11
1.3.2 Cambio de variable ligada	12
1.3.3 Reducción de aplicaciones	13
1.4 Equivalencias	14
1.4.1 Equivalencia sintáctica	14
1.4.2 α -convertibilidad	15
1.4.3 β -convertibilidad	16
1.4.4 Relaciones de equivalencia	16
2 Formalización del cálculo λ	21
2.1 Términos λ	21
2.1.1 Estructura	26
2.1.2 Clasificación	28
2.1.3 Sustitución de términos	29
2.2 Los cálculos de la conversión λ	33
2.2.1 Teorías formales	33
2.2.2 Teoría $\lambda K\beta$	35
2.2.3 Lógica combinatoria	39

2.3	Teoría de reducción	44
2.3.1	Nociones de reducción	44
2.3.2	Confluencia	50
2.3.3	Estrategias de reducción	54
3	Codificación de objetos	59
3.1	Álgebra Booleana	59
3.1.1	Valores de verdad	60
3.1.2	Expresiones booleanas	62
3.1.3	Extensiones al álgebra booleana	72
3.2	Aritmética	74
3.2.1	Numerales de Church	74
3.2.2	Operaciones elementales	78
3.2.3	Iteración	91
3.2.4	Hiperoperaciones	98
3.3	Procesos recursivos	100
3.3.1	Procedimientos v.s. procesos	101
3.3.2	Derivación de un mecanismo de recursividad	102
3.3.3	Combinador genérico de recursividad	104
3.3.4	Combinadores de punto fijo	105
3.4	Estructuras recursivas	108
3.4.1	Listas	109
3.4.2	Árboles	114
3.4.3	Gráficas	116
3.4.4	Términos λ	116
	Conclusión	121
A	Programación de codificaciones	123
A.1	Scheme	123
A.2	Haskell	124
B	Intérprete del cálculo λ	127
B.1	Módulo principal	127
B.2	Módulo comandos	134
B.3	Módulo interfaz	140
B.4	Módulo $\text{T}_{\text{E}}\text{X}$	141
C	Intérprete Lambda	147

Para revisores

Con respecto a comentarios Para hacer comentarios en el documento propongo que se utilice el entorno que corresponde al nombre del revisor (Gutu, Waissman, Frias, Jesus, Eduardo), en caso que se desee hacer un comentario anónimo, hay un entorno anónimo (Anonimo) también. Los comentarios pueden ser anidados para comentar comentarios. Ejemplo de como se ven y como se escriben:

Olivia Gutú 0.1

Comentario de Olivia

Julio Waissman 0.1

Comentario de Julio

Martín Frías 0.1

Comentario de Martín

Jesús Espinoza 0.1

Comentario de Jesús

Anónimo 0.1

Comentario anónimo

```
\begin{Gutu}
  Comentario de Olivia
  \begin{Waissman}
    Comentario de Julio
    \begin{Frias}
      Comentario de Martín
    \end{Frias}
  \end{Waissman}
\begin{Jesus}
  Comentario de Jesús
```

```

\end{Jesus}
\end{Gutu}
\begin{Anonimo}
Comentario anónimo
\end{Anonimo}

```

Con respecto al contenido Hay muchos aspectos del trabajo que me gustaría mejorar, creo que la redacción pudiera ser mucho más clara y concisa. Además me da la impresión que el trabajo está “sobreestructurado”, presentar los aspectos informales antes de la formalización y las codificaciones me parece lo más apropiado, pero al leer el trabajo desde el inicio al final me quedo con la impresión de que los cambios de capítulo son muy repentinos.

Tengo algunas autocríticas de la manera en como presento el contenido del trabajo pero quisiera esperar a recibir retroalimentación desde otros puntos de vista.

Con respecto a la forma Esto lo puse como última prioridad, lo que le hace falta al trabajo es uniformizar las enumeraciones de términos λ , en ocasiones utilizo letras, en ocasiones las enumero por capítulo. Otra cuestión algo superficial es determinar si es conveniente que los ejemplos sean especificados como bloques de contenido, inicialmente los escribí en su propio entorno de L^AT_EX para poder hacer referencia a ellos, pero creo que casi no hago referencias a ejemplos y al leer el trabajo siento que se pierde el flujo de la lectura.

Gracias por darse el tiempo de leer el trabajo.

—Eduardo

Introducción

El cálculo λ es un lenguaje y herramienta para el estudio del cómputo, ha sido utilizado en una gran variedad de áreas de la computación, desde los aspectos más fundamentales como en establecer los límites del cómputo y la semántica de lenguajes, hasta aspectos muy prácticos como en algoritmos para la inferencia de tipos y demostradores automáticos de teoremas.

Este trabajo presenta una introducción al estudio del cálculo λ puro desde una perspectiva matemática y computacional. El contenido del trabajo es producto de una revisión de la literatura sobre este cálculo y temas relacionados de las ciencias de la computación.

La estructura del trabajo se conforma de tres capítulos y dos apéndices:

- El primer capítulo aborda las ideas elementales del cálculo λ de manera informal relacionando los conceptos de este cálculo con otras áreas de estudio más convencionales de las matemáticas.
- El segundo capítulo presenta la perspectiva matemática del cálculo λ , se aborda la formalización de la sintaxis del lenguaje del cálculo y después haciendo uso de sistemas formales y sistemas de reducción se formalizan el resto de los conceptos introducidos en el primer capítulo.
- El tercer capítulo presenta la perspectiva computacional del cálculo λ , se aborda la codificación de objetos matemáticos y algoritmos del álgebra booleana y la aritmética elemental, después se explora la representación de procesos y estructuras recursivas.
- Los apéndices son implementaciones de programas para ser ejecutados en la computadora. El primer apéndice presenta una manera de programar las codificaciones del tercer capítulo en los lenguajes de programación Scheme y Haskell. El segundo apéndice presenta el intérprete y editor estructural de un lenguaje diseñado para explorar los temas de este trabajo de manera interactiva.

Las aportaciones de este trabajo son las codificaciones de algoritmos desarrolladas en el tercer capítulo y el programa del segundo apéndice.

Capítulo 1

Noción informal del cálculo λ

El cálculo λ fue 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.

En este capítulo se presentan de manera informal los aspectos más elementales del cálculo λ , en la primer sección se describe la notación usada en el cálculo en comparación a la notación matemática convencional; en la segunda sección se describen los objetos matemáticos válidos del cálculo, sus estructuras sintácticas y las expresiones que se pueden construir con estas estructuras; en la tercer sección se describen los mecanismos básicos para manipular y transformar los objetos del cálculo y en la cuarta sección se presentan las nociones de equivalencia y el significado de las expresiones.

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 [14]; y la monografía titulada *The Calculi of Lambda-Conversion* por Alonzo Church [9].

1.1 Notación funcional

El concepto de *lenguaje* es de suma importancia en el cálculo λ , informalmente el lenguaje se puede asociar con el concepto de *notación*, es decir, una manera de escribir cosas. La notación del cálculo λ consiste en una manera alternativa de escribir funciones.

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 da a las funciones, puede ser conveniente representarlas simbólicamente con expresiones, gráficamente con dibujos, numéricamente con tablas o incluso verbalmente con palabras.

Por ejemplo, la función $f : \mathbb{Z} \rightarrow \mathbb{Z}$ definida como $\{(x, 2x + y) \mid y \in \mathbb{Z}\}$ se escribe con la notación convencional como

$$f(x) = 2x + y$$

En la notación funcional usada en el cálculo λ se escribe

$$f = \lambda x.2x + y$$

Para denotar la evaluación de una función para un valor en concreto, por ejemplo el 3, se escribe con la notación convencional como

$$f(3)$$

mientras que en la notación λ se escribe

$$(f \ 3)$$

a pesar de ser similares, la notación λ hace que sea posible denotar funciones anónimas, es decir, funciones que no están asociadas a un nombre como f . Expresar funciones anónimas con la notación convencional es inconveniente ya que al leer $2x + y$ no se puede estar seguros si la variable es x o y , por otra parte, con $\lambda x.2x + y$ no existe esta ambigüedad.

La notación λ es útil para escribir funcionales o funciones de orden superior, es decir, aquellas funciones cuyo dominio o codominio son a su vez funciones, por ejemplo

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

denota una función que toma como argumento una función g y resulta en una función anónima que toma como argumento un valor x y resulta en el doble de la evaluación de g de x . Desarrollando la evaluación de f de $\lambda y.3y$ se tiene que

$$\begin{aligned} (f \ \lambda y.3y) &= ((\lambda g.\lambda x.2(g \ x)) \ \lambda y.3y) \\ &= \lambda x.2((\lambda y.3y) \ x) \\ &= \lambda x.2(3x) \\ &= \lambda x.6x \end{aligned}$$

Una diferencia conceptual importante entre el cálculo λ y el enfoque clásico del estudio de funciones es que este cálculo permite expresar únicamente funciones, evaluaciones y variables. Otras estructuras matemáticas como los números y los conjuntos no son estructuras válidas del

cálculo λ . Las implicaciones de esto son fuertes ya que en la matemática clásica una función es un conjunto y la notación utilizada al tratar funciones es azúcar sintáctico sobre la notación usada en teoría de conjuntos. Es por ello que la definición de función, evaluación de funciones y variables en el cálculo λ se basa únicamente en la sintaxis de las expresiones.

En el estudio del cálculo λ se distinguen dos lenguajes: el lenguaje de las expresiones y el *metalenguaje*. El lenguaje de las expresiones es un *lenguaje formal* que especifica las secuencias de símbolos que representan expresiones válidas, se relaciona con las clases de objetos del cálculo 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.

1.2 Expresiones

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

1.2.1 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 λ , los nombres son parte del metalenguaje y no del lenguaje de las expresiones.

1.2.2 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*.

En el cálculo λ las abstracciones 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 .

Las abstracciones proveen el mecanismo para asociarle un valor a una variable. Esta asociación es similar a la que proveen las funciones en la matemática clásica y las proposiciones en la lógica de primer orden, sin embargo, éstas se basan en conjuntos y en operaciones sobre conjuntos, por ejemplo $f(x) = M$ 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 .

Es posible utilizar la definición de función para describir operaciones y transformaciones de expresiones en el cálculo λ , utilizar lógica de primer orden para aseverar propiedades del cálculo, o utilizar aritmética para cuantificar aspectos de las expresiones del cálculo λ , sin embargo estos objetos matemáticos son usados en el metalenguaje y no como expresiones válidas del cálculo.

1.2.3 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 *evaluación de funciones*. En el cálculo λ se hace una distinción entre la expresión sintáctica de una evaluación como $f(x)$ y el acto de encontrar el valor de asociado de x en f . Usualmente el primero se refiere a aplicaciones y el segundo se refiere a reducciones. Por ejemplo, cuando se tiene la aplicación de una abstracción $(\lambda x. M)$ en alguna 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 número 13.

1.2.4 Extensiones

Cuando se desea representar en el cálculo λ 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 $f: \mathbb{N} \rightarrow \mathbb{N}$ dada por $f(x) = x^2$ primero se deben codificar los números naturales con expresiones del cálculo λ , 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 λ 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 λ debe ser codificado como expresiones, los objetos representados pueden ser entendidos de la misma manera.

1.2.5 Ejemplos

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

Ejemplo 1.2.1. Algunas expresiones del cálculo λ

- | | |
|-----------------------------------------------|-----|
| x | (a) |
| $(\lambda x.x)$ | (b) |
| $(y(\lambda x.x))$ | (c) |
| $((\lambda y.(y(\lambda x.x)))(\lambda w.w))$ | (d) |
| $(\lambda x.(xx))$ | (e) |
| $(\lambda f.(\lambda x.(f(fx))))$ | (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 1.2.2. 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 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 1.2.2, 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 λ :

Ejemplo 1.2.3. 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” [15, pp. 431–437] término originalmente asociado a una paradoja sobre sistemas lógicos [23]. 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 λ 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.

1.3 Operaciones

En el cálculo λ 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 1.2 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.

1.3.1 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 N , este procedimiento se denota

$$M[x := N]$$

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

Ejemplo 1.3.1. Sustituciones sencillas

$$x[x := y] = y \quad (a)$$

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

$$(((w x)y)z) [x := a] [y := b] = (((w a)b)z) \quad (c)$$

$$(x x) [x := (\lambda w. w)] = ((\lambda w. w)(\lambda w. w)) \quad (d)$$

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 λ 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 1.3.2. 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 1.3.2 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 .

1.3.2 Cambio de variable ligada

Para resolver el problema presentado en el ejemplo 1.3.2 se debe considerar otra operación llamada *cambio de variable ligada*. Se parte de la observación que en una expresión del cálculo λ , 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 λ , 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 1.3.3. Procedimiento de sustitución para el ejemplo 1.3.2

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 1.3.3 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 1.3.4. 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

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

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 ;
- $(MN)[x := P]$ 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 ,
 - $(\lambda z.M[y := z][x := N])$ cuando x es una variable libre en M y y es una variable libre en N .

1.3.3 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 1.3.5. Procedimiento de reducir la aplicación de $(\lambda x.x)$ en y

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

Ejemplo 1.3.6. 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 1.3.7. 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 λ 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 λ haciendo solamente uso de las operaciones de *sustitución*, *cambio de variable ligada* y *aplicación de abstracciones*.

1.4 Equivalencias

El cálculo λ 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 λ , 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 λ* y no únicamente de un cálculo λ .

Como se aborda en la sección 1.3, con la operación de sustitución se puede transformar expresiones del cálculo λ 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 λ , 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 λ y las maneras en las que las equivalencias se relacionan entre sí.

1.4.1 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 λ , la equivalencia sintáctica es denotada como

$$M \equiv N$$

cuando M es sintácticamente la misma expresión que N .

Todos los cálculos λ , 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.

1.4.2 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 λ 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.

Ejemplo 1.4.1 (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 λ como árboles.

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

1.4.3 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 llamada 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:

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

1.4.4 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

- Toda expresión M es equivalente a sí misma, es decir, $M \sim M$.
- Si una expresión M es relacionada con una equivalencia a otra expresión N , entonces N también es relacionada a M , es decir $M \sim N \implies N \sim M$.
- 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, $M \sim N, N \sim P \implies M \sim P$.

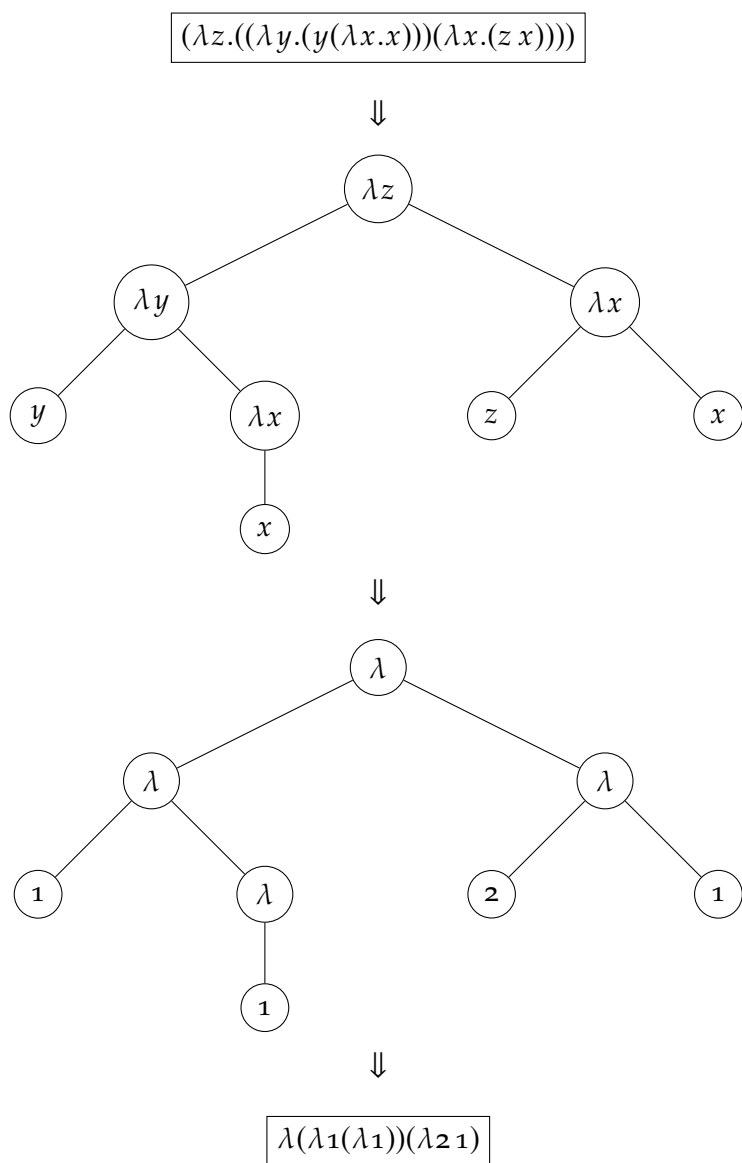


Figura 1.1: Transformación gráfica del ejemplo 1.4.1

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 [13, 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 1.4.1 (Ecuaciones de \sim). Las ecuaciones con \sim que se satisfacen para expresiones del cálculo λ son

$$\begin{aligned} (\lambda x.M) &\sim (\lambda y.M[x := y]) & (\alpha) \\ ((\lambda x.M)N) &\sim M[x := N] & (\beta) \\ M &\sim M & (\rho) \\ M \sim N &\implies N \sim M & (\sigma) \\ M \sim N, N \sim P &\implies M \sim P & (\tau) \end{aligned}$$

Las ecuaciones en la definición 1.4.1 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

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

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 [13, p. 59] como ν , μ y ξ , junto con las reglas de \sim completan la definición declarativa de β -convertibilidad:

Definición 1.4.2. Reglas que debe cumplir \sim para ser $=_\beta$

$$M \sim N \implies (M Z) \sim (N Z) \quad (\nu)$$

$$M \sim N \implies (Z M) \sim (Z N) \quad (\mu)$$

$$M \sim N \implies (\lambda x.M) \sim (\lambda x.N) \quad (\xi)$$

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

Ejemplo 1.4.2. Razonamiento para concluir que $(\lambda f.((\lambda x.(f x))y)) =_\beta (\lambda f.(f y))$

$$1. ((\lambda x.(f x))y) =_\beta (f y) \quad \text{por } \beta$$

$$2. (\lambda f.((\lambda x.(f x))y)) =_\beta (\lambda f.(f y)) \quad \text{por } \xi$$

Es posible incluir aún más reglas de equivalencia cuando se estudia el cálculo λ , 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 λ 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 λ 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 incluir 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 λ , 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 λ

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 λ pueden ser considerados como instancias 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 λ , 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 presenta un tratamiento matemático del cálculo λ con el objetivo de formalizar las ideas presentes en el capítulo 1 y ahondar en el objeto de estudio de este cálculo.

En la primer sección se define el conjunto que tiene como elementos todas las expresiones válidas del cálculo, se presentan propiedades y criterios de clasificación de expresiones y finalmente se formaliza el concepto de sustitución; en la segunda sección se construye una teoría formal del cálculo λ , en donde un conjunto de axiomas y reglas de inferencia permiten plantear razonamientos lógicos para demostrar propiedades del cálculo; finalmente en la tercer sección se formulan nociones de reducción para las expresiones del cálculo usando sistemas de reescritura.

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 [14] 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 λ

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

Los *términos* λ son la formalización de las expresiones descritas en la sección 1.2. El conjunto de todos los términos λ es un lenguaje formal Λ en donde sus elementos son cadenas compuestas de símbolos de un alfabeto [16].

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

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

$$x \in V \Rightarrow x \in \Lambda \quad (a)$$

$$M \in \Lambda, x \in V \Rightarrow (\lambda x.M) \in \Lambda \quad (b)$$

$$M, N \in \Lambda \Rightarrow (MN) \in \Lambda \quad (c)$$

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

- (a) establece que todo elemento de V es un término λ a los cuales se les llama *átomos*;
- (b) establece que las cadenas de la forma $(\lambda x.M)$ son términos λ , donde x es un átomo y M es cualquier término λ , a estos términos se les llama *abstracciones*;
- (c) establece que las cadenas de la forma (MN) son términos λ , donde M y N son términos λ cualesquiera, a estos términos se les llama *aplicaciones*.

En el estudio usual de lenguajes formales [16], Λ pertenece a la clase de lenguajes libres de contexto y puede ser definido de la siguiente manera:

Definición 2.1.2 (Términos λ). El conjunto de términos λ 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 λ , aplicaciones, abstracciones y átomos respectivamente;

símbolos terminales $\{ (,), ., \lambda, v, o \}$, 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 λ ;

reglas de producción

$$T \rightarrow E \mid F \mid A \quad (a)$$

$$A \rightarrow v_o \mid A_o \quad (b)$$

$$F \rightarrow (\lambda A . T) \quad (c)$$

$$E \rightarrow (T T) \quad (d)$$

Dada una secuencia de símbolos M , se pueden utilizar estas dos definiciones para verificar si M es o no un término λ . 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 λ 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 λ 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

$$\begin{aligned} (\lambda v_{00}.v_0 v_{00}), v_0 \in \Lambda &\implies ((\lambda v_{00}.v_0 v_{00})v_0) \in \Lambda; \\ v_{00} \in V, v_0 v_{00} \in \Lambda &\implies (\lambda v_{00}.v_0 v_{00}) \in \Lambda; \\ v_0 v_{00} \notin \Lambda &\therefore ((\lambda v_{00}.v_0 v_{00})v_0) \notin \Lambda. \end{aligned}$$

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

$$\begin{aligned} T &\Rightarrow E \Rightarrow (T T) \Rightarrow (F T) \Rightarrow ((\lambda A . T) T) \\ &\Rightarrow ((\lambda A_0 . T) T) \Rightarrow ((\lambda v_{00} . T) T) \\ &\Rightarrow ((\lambda A . v_0 v_{00}) T). \end{aligned}$$

La sintaxis del cálculo λ 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 λ , esta equivalencia contempla las consideraciones de este listado.
2. Cuando se hace referencia a *cualquier* término λ 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 λ 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) $((MN)P) \equiv MNP$, en general, se considera la aplicación de términos λ con asociación a la izquierda. Se tiene que tener cuidado con respetar esta regla, por ejemplo $(M(N(OP))) \equiv M(N(OP)) \not\equiv MNOP$.
 - (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 $((\lambda x.(MN))O) \equiv (\lambda x.(MN))O \not\equiv \lambda x.(MN)O$ ya que el lado derecho de la equivalencia es sintácticamente equivalente a $(\lambda x.((MN)O))$.
 - (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 [25].

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

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

y la consideración de 4c como

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

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 λ 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 λ complejos resulta conveniente acortarlos. A continuación se muestran ejemplos de términos λ 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 λM 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 λM , denotada como $\|M\|$, es la cantidad de *apariciones* de átomos en M , se determina a partir de la estructura del término λ como:

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

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 λM , su longitud será estrictamente mayor a cero. Una implicación de esta observación es que al “desbaratar” la longitud de un término λ de acuerdo a su estructura, en el caso de que M sea una aplicación o una abstracción, la longitud de sus partes es estrictamente 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 λ 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 λ considera también la estructura del término, de tal manera que para una proposición P sobre un término λM , 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 puede *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 $z \notin M$ 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 $M \equiv \lambda x.x y(\lambda z.y) \equiv (\lambda x.((x y)(\lambda z.y)))$, 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 λ 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 λ y las propiedades que tienen los términos de acuerdo a su clasificación.

Al considerar las apariciones de átomos en un término λ , 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 $x \in M$ 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 1.3.3 la distinción entre una variable libre y una ligada era importante ya que las variables libres nunca son sustituidas 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 λ .

Definición 2.1.7 (Variables libres). El conjunto de variables libres de un término λM se denota $FV(M)$ y se define de manera inductiva como:

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

Cuando $FV(M) = \emptyset$ 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 $((\lambda y.x)\lambda x.y)$.

- $FV(x(\lambda x.x y z)) = \{x, y, z\}$;

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

En ocasiones es importante distinguir los términos λ 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^0 al conjunto

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

La notación Λ^0 se puede generalizar para identificar diferentes subconjuntos de Λ a partir de las variables libres de los términos λ :

$$\Lambda^0(\vec{x}) = \{M \in \Lambda \mid FV(M) \subseteq \{\vec{x}\}\}$$

De tal manera que:

$$\Lambda^0 = \Lambda^0(\emptyset)$$

Si consideramos un término M con variables libres, se puede encontrar otro término $N \in \Lambda^0$ similar a M , al cual se le llama clausura de M .

Definición 2.1.9 (Clausura). La clausura de un término λM con $FV(M) \neq \emptyset$ es un término λ

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

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

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

En la sección 1.3.1 se dió una descripción informal pero precisa de la sustitución en el cálculo λ . A continuación se presenta la definición utilizando los conceptos introducidos en este capítulo.

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:

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

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 $x \not\equiv y, x \notin \text{FV}(P)$

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

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

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

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

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

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

Las siguientes equivalencias de sustituciones múltiples son importantes para resaltar la relevancia de los detalles de la definición 2.1.10.

- $M[x := y][y := P]$

Esta operación es equivalente a $M[x := P]$ siempre y cuando $y \notin \text{FV}(M)$, de otro modo, M tiene variables sustituibles y que no serán sustituidas con $M[x := P]$.

- $M[x := y][y := x]$

Esta operación es equivalente a M siempre y cuando $y \notin \text{FV}(M)$, el argumento es el mismo que el punto anterior.

- $M[y := Q][x := P]$

Esta operación debe analizarse por casos ya que P y Q pudieran o no contener apariciones de x y y :

- Es equivalente a $M[x := P][y := Q]$ cuando $y \notin \text{FV}(P)$ y $x \notin \text{FV}(Q)$ ya que ni la primer operación no introduce variables libres x no afecta el resultado de la segunda operación y ya que la segunda operación no introduce variables libres y no afecta invertir el orden de las operaciones.
- Cuando la segunda operación no introduce variables libres y pero la primera operación si introduce variables libres x es equivalente a $M[x := P][y := Q[x := P]]$,

es decir, se sustituyen las variables libres x de Q por las que se sustituirían con la operación original y se procede como en el punto anterior.

- $M[x := Q][x := P]$

Esta operación es equivalente a $M[x := Q][x := P]$, debido a que la sustitución de x por P tiene efecto únicamente cuando Q introduce variables x libres, por lo tanto, cambiar las x libres de Q por P antes de hacer la primera operación produce el resultado correcto.

Lema 2.1.1. Si $(y\ x) \notin FV(L)$ 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 λ 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 $\lambda M \equiv (xy)$.

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

El caso análogo con la sustitución es

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

2.2 Los cálculos de la conversión λ

El objetivo principal de esta subsección es presentar una formalización del cálculo λ 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 $\lambda K\beta$, 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 $\phi: \mathcal{F}^n \rightarrow \mathcal{F}$ con $n \geq 1$. Si se consideran n fórmulas A_1, \dots, A_n tal que

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

Se dice que la secuencia $\langle A_1, \dots, A_n, B \rangle$ 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

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

Observación. En la literatura se pueden encontrar 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 $\langle A_1, \dots, A_{n+1} \rangle$ 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 [14, 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 teorías que formalizan los cálculos λ , los objetos que se relacionan son términos λ . 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.

De acuerdo a Barendregt [5, p. 22], el objeto de estudio del cálculo λ es el conjunto de términos λ módulo convertibilidad. Que este conjunto sea el objeto de estudio de una teoría λ del cálculo λ significa que cada elemento de Λ módulo convertibilidad, denotado $\Lambda / =_{\lambda}$, es distinto y representa una clase de términos λ considerados en la teoría λ como equivalentes. Cuando $\lambda \vdash M = N$ se dice que M y N son términos *convertibles*, también denotado $M =_{\lambda} N$.

Habiendo definido una teoría λ , el interés de estudiarla es

- determinar los términos que son convertibles en λ ;
- estudiar las propiedades que comparten dos términos convertibles;
- modificar a λ y comparar la teoría modificada con la original.

La comparación entre teorías usualmente consiste en partir de una teoría λ , modificar sus fórmulas, axiomas o reglas para obtener otra teoría λ' y determinar si λ y λ' 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 λ puede implicar cambiar la definición de sus términos, es decir, utilizar un lenguaje formal diferente al de λ 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 λ cuyo lenguaje de términos es el conjunto Λ^0 para que los términos sean escritos con el índice de DeBruijn mostrado en 1.4.1 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 λ con términos Λ de tal manera que se consideren únicamente los términos cerrados Λ^0 si puede tener fuertes implicaciones en la convertibilidad de la teoría ya que habrá términos λ no admitidos en fórmulas.

Otra manera de modificar una teoría λ 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 $\langle A_1, \dots, A_n, B \rangle$ de ϕ , ϕ es derivable en λ si y sólo si

$$\lambda, \{A_1, \dots, A_n\} \vdash B \quad (2.5)$$

Cuando añadir una regla ϕ a una teoría λ no cambia el conjunto de teoremas se dice que la regla es *admisibile*, 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 $\langle A_1, \dots, A_n, B \rangle$ de ϕ :

$$(\lambda \vdash A_1), \dots, (\lambda \vdash A_n) \implies (\lambda \vdash B) \quad (2.6)$$

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 λ 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 λ .

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:

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

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

$$\lambda x.M = \lambda y.M[x := y] \qquad \forall y \notin \text{FV}(M) \qquad (\alpha)$$

$$(\lambda x.M)N = M[x := N] \qquad (\beta)$$

$$M = M \qquad (\rho)$$

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

$$\frac{M = N}{Z M = Z N} \qquad (\mu)$$

$$\frac{M = N}{M Z = N Z} \qquad (\nu)$$

$$\frac{M = N}{\lambda x.M = \lambda x.N} \qquad (\xi)$$

$$\frac{M = N \quad N = P}{M = P} \qquad (\tau)$$

$$\frac{M = N}{N = M} \qquad (\sigma)$$

Consideremos la convertibilidad en $\lambda K\beta$ de los términos λ

$$M \equiv (\lambda f.x((\lambda y.y f)\lambda z.z))w$$

$$N \equiv x w$$

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

La teoría λ puede ser modificada de muchas maneras, en este trabajo se considera una extensión no trivial a esta teoría con interpretaciones y consecuencias interesantes.

El concepto de igualdad de funciones usado en la mayoría de las ramas de la matemática tiene una propiedad conocida como *extensionalidad*, esta propiedad de las relaciones de equivalencia hace referencia a las características externas de los objetos que compara, en el caso de las funciones, esto implica que para funciones f y g con el mismo dominio

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

Contrario a esta propiedad de las funciones, en la computación, el tema central son los

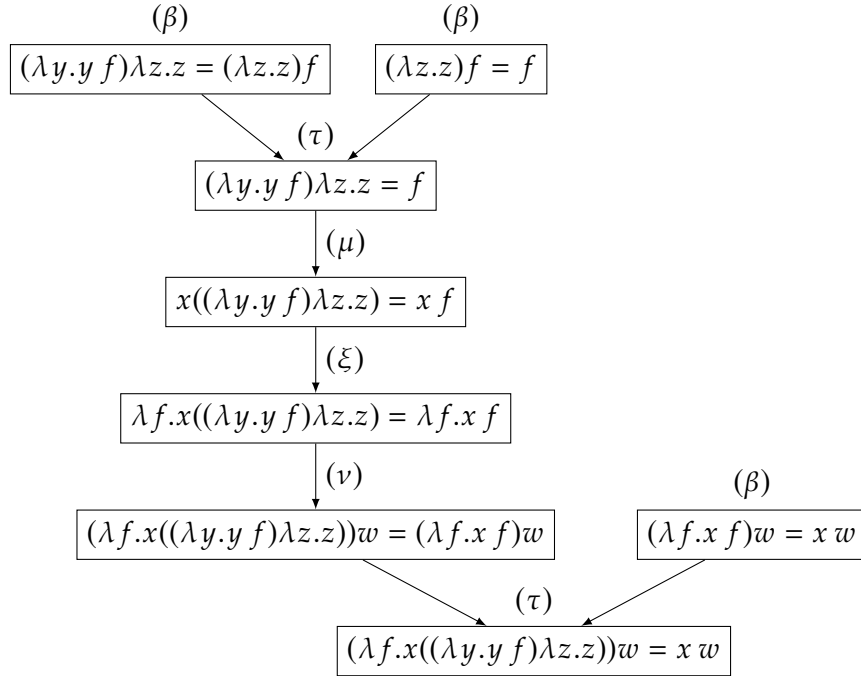


Figura 2.1: Árbol de deducción para demostrar la convertibilidad entre dos términos

procedimientos y procesos que describen los programas o algoritmos, cuyas igualdades son *intensionales*, es decir, si dos programas computan la misma función matemática, no necesariamente se dice que son el mismo programa ya que uno pudiera ser, por ejemplo, más eficiente que otro y la eficiencia es una característica interna de cada algoritmo, independiente de los resultados que computa.

La teoría λ también es intensional ya existen dos términos λF y G tales que para todo término X la aplicación $(F X)$ es convertible a $(G X)$ pero $F \neq_{\lambda} G$.

Consideremos por ejemplo, dos términos $F \equiv y$ y $G \equiv \lambda x.y x$. Claramente $F \neq_{\lambda} G$, sin embargo al aplicar estos dos términos a un término cualquiera M se tiene que

$$\begin{aligned}
 G X &=_{\lambda} (\lambda x.y x)X \\
 &=_{\lambda} y X \\
 &=_{\lambda} F X
 \end{aligned}$$

Cuando se plantea formalizar un cálculo λ que sea extensional, surge la pregunta, ¿Qué es demostrable en la teoría extensional que no es demostrable en λ . A continuación se presentan tres diferentes agregados a la teoría λ las cuales incluyen la propiedad de extensionalidad y que han sido propuestas en la literatura [14, 5]. Las teorías extendidas son llamadas $\lambda\zeta$, $\lambda + ext$ y $\lambda\eta$ de acuerdo a la regla que se añade a la definición 2.2.2.

Definición 2.2.3 (Reglas de extensionalidad). Cada una de las siguientes reglas nos permite añadir a λ la propiedad de extensionalidad.

Reglas de inferencia

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

$$\frac{M P = N P}{M = N} \quad \forall P \in \Lambda \quad (\text{ext})$$

Axiomas

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

Hindley y Seldin en [14, p. 77] presentan las reglas (ζ) y (ext) como equivalentes, explicadas de manera informal como “Si M y N tienen el mismo efecto sobre un objeto, entonces $M = N$ ”. En este trabajo se consideran dos teorías modificadas a partir de λ :

- La teoría $\lambda\zeta$ es igual que λ pero incluyendo también la regla ζ ;
- La teoría $\lambda\eta$ es igual que λ pero incluyendo también el axioma η .

El ejemplo presentado para introducir la propiedad de extensionalidad es precisamente el axioma η , en [5, p. 32] se presenta el siguiente teorema y demostración.

Teorema 2.2.1. Las teorías $\lambda\zeta$ y $\lambda\eta$ son equivalentes en reglas.

Demostración. Primero, se demuestra que la fórmula η es derivable en $\lambda\zeta$:

1. Aplicando β , se tiene que $(\lambda x. M x)x =_{\lambda\zeta} (M x)$ si $x \notin \text{FV}(M)$;
2. Aplicando ζ , se tiene que $(\lambda x. M x) =_{\lambda\zeta} M$.

Después, se demuestra que la regla ζ es derivable en $\lambda\eta$:

1. Se parte de la primicia $(M x) =_{\lambda\eta} (N x)$ si $x \notin \text{FV}(M)$;
2. Aplicando ξ , se tiene que $(\lambda x. M x) =_{\lambda\eta} (\lambda x. N x)$;
3. Aplicando η , se tiene que $M =_{\lambda\eta} N$.

□

En [14, 5] se le da un tratamiento más extenso a diferentes teorías formales del cálculo λ y otros sistemas similares, incluyendo aquellas con la propiedad de extensionalidad.

2.2.3 Lógica combinatoria

En esta sección se introduce una teoría similar en objetivos y contemporánea al cálculo λ , llamada lógica combinatoria. Creada inicialmente por Moses Schönfinkel pero desarrollada extensamente por Haskell Curry en la década de 1930, la lógica combinatoria tiene tanto en común con el cálculo λ , que comparten una gran cantidad de definiciones, teoremas y extensiones.

En este trabajo no se desarrolla la lógica combinatoria, pero se presenta la teoría $CL\omega$ que formaliza este sistema para su comparación con $\lambda K\beta$.

Los términos de la lógica combinatoria CL se definen de la siguiente manera

Definición 2.2.4 (Términos CL). Sea V el conjunto de variables $\{v_o, v_{oo}, v_{ooo}, \dots\}$ y C el conjunto de constantes atómicas $\{I, K, S\}$. El conjunto de términos de la lógica combinatoria, denotado CL se define de manera inductiva de la siguiente manera:

$$x \in V \implies x \in CL \quad (a)$$

$$x \in C \implies x \in CL \quad (b)$$

$$X, Y \in CL \implies (XY) \in CL \quad (c)$$

Como se puede apreciar en la definición de CL , los términos en la lógica combinatoria son parecidos a los términos en el cálculo λ , sólo que en CL no hay abstracciones y en λ no hay constantes atómicas.

Definición 2.2.5 (Teoría $CL\omega$). El conjunto de fórmulas \mathcal{F} en $CL\omega$ tiene como elementos ecuaciones de la forma:

$$X = Y \quad \forall X, Y \in CL \text{ (de la definición 2.2.4)}$$

Los axiomas \mathcal{A} de $CL\omega$ son:

$$I X = X \quad (I)$$

$$K X Y = X \quad (K)$$

$$S X Y Z = X Z(Y Z) \quad (S)$$

$$X = X \quad (\rho)$$

Las reglas \mathcal{R} de $CL\omega$ son:

$$\frac{X = X'}{Z X = Z X'} \quad (\mu)$$

$$\frac{X = X'}{X Z = X' Z} \quad (\nu)$$

$$\frac{X = Y \quad Y = Z}{X = Z} \quad (\tau)$$

$$\frac{X = Y}{Y = X} \quad (\sigma)$$

Existen equivalencias entre modificaciones de teorías del cálculo λ y la lógica combinatoria [14, pp. 92–106], pero en este trabajo se muestra una introducción a las similitudes entre λ y $CL\omega$, en particular, la correspondencia entre las clases de términos de $CL\omega$ que se pueden representar en λ y viceversa.

Es sencillo representar las constantes atómicas de CL utilizando abstracciones del cálculo λ . Los términos que representen a I , K y S en Λ deben satisfacer las reglas **(I)**, **(K)** y **(S)** de la definición 2.2.5.

Definición 2.2.6 (Constantes I , K y S en Λ).

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

Se corroboran estas definiciones considerando a M , N y Z en Λ utilizando el axioma (β) de λ .

$$\begin{aligned} IM &=_{\lambda} (\lambda x.x)M \\ &=_{\lambda} M \\ KMN &=_{\lambda} (\lambda x y.x)MN \\ &=_{\lambda} (\lambda y.M)N \\ &=_{\lambda} M \\ SMNZ &=_{\lambda} (\lambda x y z.x z(y z))MNZ \\ &=_{\lambda} (\lambda y z.M z(y z)) \\ &=_{\lambda} (\lambda z.M z(N z)) \\ &=_{\lambda} MZ(NZ) \end{aligned}$$

Las variables en CL pueden representarse en Λ como átomos, las aplicaciones en CL como aplicaciones en Λ y el resto de los axiomas y reglas en $CL\omega$ tienen sus contrapartes en Λ . Por lo tanto, todo término en CL puede ser representado con términos en Λ y toda deducción realizada en $CL\omega$ puede también realizarse en λ considerando la definición 2.2.6.

Por otra parte, la lógica combinatoria puede representar grán parte del cálculo λ , pero no todo. En particular la regla (ξ) en λ no es admisible en la teoría $CL\omega$ [14, p. 92].

Con la teoría $CL\omega$ se puede generar al conjunto Λ^0 con combinaciones de aplicaciones. Debido a que $SKK =_{\lambda} I$, sólo es necesario combinar con aplicaciones a K y a S para generar cualquier término cerrado.

Definición 2.2.7 (Bases). El concepto de *base* es de importancia para plantear formalmente la proposición anterior.

1. Sea $\mathcal{X} \subset \Lambda$. El conjunto de términos *generado* por \mathcal{X} , denotado \mathcal{X}^+ , es el conjunto mas pequeño tal que
 - (a) $\mathcal{X} \subseteq \mathcal{X}^+$,
 - (b) $M, N \in \mathcal{X}^+ \implies (MN) \in \mathcal{X}^+$.
2. Sea $\mathcal{P}, \mathcal{X} \subset \Lambda$. \mathcal{X} es una *base* para \mathcal{P} si para toda $M \in \mathcal{P}$ existe $N \in \mathcal{X}^+$ tal que $N = M$.
3. \mathcal{X} es llamada una *base* si \mathcal{X} es una base para Λ^0 .

Proposición 2.2.2. $\{S, K, I\}$ es una base, es decir, para todo término $M \in \Lambda^0$, existe un término M' compuesto de aplicaciones de S , K e I tal que $M =_{\lambda} M'$.

La demostración de la proposición 2.2.2 consiste en la construcción de un algoritmo para transformar M a M' . El siguiente lema es utilizado en esa demostración:

Lema 2.2.3. Sea $\lambda x.M$ una abstracción tal que $\text{Sub}(M)$ no contiene abstracciones

1. Si $M =_{\lambda} x$, entonces $\lambda x.M =_{\lambda} I$;
2. Si $x \notin \text{FV}(M)$, entonces $\lambda x.M =_{\lambda} (KM)$;
3. Si $M =_{\lambda} P Q$, entonces $\lambda x.M =_{\lambda} S(\lambda x.P)(\lambda x.Q)$.

Demostración. Utilizando la definición 2.2.6

$$I N =_{\lambda} N \tag{1}$$

$$=_{\lambda} (\lambda x.x)N$$

$$=_{\lambda} (\lambda x.M)N$$

$$K M N =_{\lambda} (\lambda x y.x)M N \tag{2}$$

$$=_{\lambda} (\lambda y.M)N$$

$$=_{\lambda} (\lambda x.M)N$$

$$S(\lambda x.P)(\lambda x.Q) =_{\lambda} (\lambda abc.(a c)(b c))(\lambda x.P)(\lambda x.Q) \tag{3}$$

$$=_{\lambda} \lambda c.((\lambda x.P)c)((\lambda x.Q)c)$$

$$=_{\lambda} \lambda c.P[x := c]Q[x := c]$$

$$=_{\lambda} \lambda c.(P Q)[x := c]$$

$$=_{\lambda} \lambda x.P Q$$

$$=_{\lambda} \lambda x.M$$

□

Demostración de la proposición 2.2.2. Sea $M \in \Lambda^0$, se construye un término $M' \in \{S, K, I\}^+$ tal que $M' =_\lambda M$ con el algoritmo 2.1.

Algoritmo 2.1 SKI

Entrada: $M \in \Lambda^0$

Salida: $M' \in \{S, K, I\}$

$M' \leftarrow M$

$\mathcal{L} \leftarrow \{A \in \text{Sub}(M') \mid A \equiv \lambda x.N\}$

mientras $\mathcal{L} \neq \emptyset$ **hacer**

$A \leftarrow \lambda x.N \in \mathcal{L} \mid \|\lambda x.N\| \leq A', \forall A' \in \mathcal{L}$

si $A \equiv \lambda x.a$ **entonces**

si $a \equiv x$ **entonces**

$M'[A] \leftarrow I$

de lo contrario y si $a \not\equiv x$ **entonces**

$M'[A] \leftarrow K a$

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**

$M'[A] \leftarrow S(KP)(KQ)$

de lo contrario y si $x \notin \text{FV}(P) \wedge x \in \text{FV}(Q)$ **entonces**

$M'[A] \leftarrow S(KP)(\lambda x.Q)$

de lo contrario y si $x \in \text{FV}(P) \wedge x \notin \text{FV}(Q)$ **entonces**

$M'[A] \leftarrow S(\lambda x.P)(KQ)$

de lo contrario y si $x \in \text{FV}(P) \wedge x \in \text{FV}(Q)$ **entonces**

$M'[A] \leftarrow S(\lambda x.P)(\lambda x.Q)$

fin si

$M'[A] \leftarrow \text{SKI}(M'[A])$

fin si

$\mathcal{L} \leftarrow \mathcal{L} \setminus \{A\}$

fin mientras

regresar M'

El algoritmo enumera 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 I en M .

(b) Si $a \neq x$ se transforma $\lambda x.N$ a $(K a)$ 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 $S(KP)(KQ)$ en M .

(b) Si $x \notin \text{FV}(P)$ y $x \in \text{FV}(Q)$ se transforma $\lambda x.P Q$ a $S(KP)(\lambda x.Q)$ en M .

(c) Si $x \in \text{FV}(P)$ y $x \notin \text{FV}(Q)$ se transforma $\lambda x.P Q$ a $S(\lambda x.P)(KQ)$ en M .

(d) Si $x \in \text{FV}(P)$ y $x \in \text{FV}(Q)$ se transforma $\lambda x.P Q$ a $S(\lambda x.P)(\lambda x.Q)$ en M .

En los casos **2a**, **2b**, **2c**, **2d** 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 **1a** y **1b** de la transformación no introducen abstracciones, en una cantidad finita de pasos el término M transformado no tendrá abstracciones de la forma $\lambda x.N$.

Si el término M tiene dos o más abstracciones con la misma longitud, el orden en el que se procesen por el algoritmo no cambia el resultado, esto es debido a que los términos que transforma son subtérminos de las abstracciones.

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 **1b** (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 **2d** (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 $a': \mathbb{N} \rightarrow \mathbb{N}$:

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

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 $p: \mathbb{N} \rightarrow \mathbb{N}$:

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

□

Con esta proposición demostrada, tenemos que cualquier término λ cerrado puede ser

traducido a términos en CL sin variables y estos dos términos son convertibles en la teoría λ .

2.3 Teoría de reducción

El concepto de reducción fue introducido en la sección 1.3.3 como el análogo de la evaluación de funciones para el cálculo λ . Sin embargo, la formalización del cálculo λ como una teoría de reducción va más allá de complementar la analogía de este sistema con las funciones de la matemática clásica.

Cuando se estudia el cálculo λ desde una perspectiva computacional, la convertibilidad de términos λ vista como la demostrabilidad de una fórmula en una teoría formal presenta cierta asimetría

$$(\lambda x.x x)y =_{\lambda} (y y)$$

puede ser interpretada como “ $(y y)$ es el resultado de computar $(\lambda x.x x)y$ ”, pero no viceversa. Esto es debido a que toda la información necesaria para obtener $(y y)$ se encuentra en $(\lambda x.x x)y$ mientras que en el otro sentido, $(y y)$ no nos dice como obtener un resultado. Conociendo las reglas de transformación de términos $(\lambda x.x x)y$ nos provee, de cierta manera, un algoritmo y una entrada.

En este capítulo se desarrolla el complemento computacional al estudio del cálculo λ de la sección 2.2. Primero se aborda la reducción en general, independiente de la regla (β) como fué presentada en la introducción a este trabajo. Después se aborda la propiedad de confluencia, la cuál contempla ciertas características deseables de los sistemas de reducción. Finalmente se presentarán algunas estrategias para la reducción de términos λ .

Esta sección se basa principalmente en el capítulo 3, 11 y 13 de [5].

2.3.1 Nociones de reducción

El concepto de reducción que se ha considerado hasta el momento corresponde a la β -reducción, sin embargo, nociones de reducción pueden ser definidas de manera general.

Las reducciones sobre Λ se formalizan como relaciones binarias R representadas como conjuntos de tuplas (M, M') donde $M, M' \in \Lambda$. Las propiedades que cumplan las tuplas de una relación nos permiten caracterizar las relaciones de varias maneras.

Consideremos las reglas de inferencia en la definición 2.2.2 de la teoría λ . Asignarle una de estas reglas a una relación nos permite analizar el tipo de transformaciones que se pueden realizar en un término para obtener otro.

Cuando se le asignan a una relación R sobre Λ las reglas (μ) , (ν) y (ξ) de la teoría λ se dice

que R es *compatible*. Formalmente, una relación es compatible cuando

$$\begin{aligned} (M, M') \in R &\implies (Z M, Z M') \in R \\ (M, M') \in R &\implies (M Z, M' Z) \in R \\ (M, M') \in R &\implies (\lambda x.M, \lambda x.M') \in R \end{aligned}$$

para toda $M, M', Z \in \Lambda$. La compatibilidad de una regla nos permite asegurar que las transformaciones de una regla se pueden realizar sobre subtérminos de un término.

Una *relación de reducción* sobre Λ es aquella que es compatible, reflexiva y transitiva. Esto complementa las propiedades de compatibilidad con las reglas (ρ) y (τ) de la teoría λ , es decir

$$\begin{aligned} (M, M) &\in R \\ (M, M') \in R, (M', Z) &\implies (M Z) \in R \end{aligned}$$

para toda $M, M', Z \in \Lambda$. Una relación de reducción establece transformaciones de más de un paso y también la capacidad de no transformar un término.

Finalmente, si a una relación de reducción sobre Λ se le añade la regla (σ) de la teoría λ , se dice que es una *relación de igualdad* o *congruencia*, debido a que además de ser compatible, es simétrica, reflexiva y transitiva. Formalmente una relación R sobre Λ es de congruencia cuando es una relación de reducción y además

$$(M, M') \in R \implies (M', M) \in R$$

para toda $M, M' \in \Lambda$. Una relación de igualdad establece que las transformaciones se pueden realizar de forma inversa.

Para definir reducciones en el cálculo λ se define primero una *noción de reducción*, esta es simplemente una relación binaria sobre Λ , por ejemplo, la noción de reducción β se define como

$$\beta = \{((\lambda x.M)N, M[x := N]) \mid M, N \in \Lambda\} \quad (2.11)$$

A partir de una noción de reducción nos podemos referir a sus *clausuras*. Sea R una noción de reducción sobre Λ , entonces

- La clausura reflexiva de R es la relación más pequeña que extiende a R para que sea reflexiva;
- La clausura transitiva de R es la relación más pequeña que extiende a R para que sea transitiva;
- La clausura compatible de R es la relación más pequeña que extiende a R para que sea compatible.

Al considerar las nociones de reducción como reglas de transformación de términos λ , es natural referirse a las transformaciones realizadas en un paso, en cero o más pasos y en ambas direcciones.

Definición 2.3.1. Sea R una noción de reducción sobre Λ . R introduce las siguientes relaciones binarias

- R -contracción, denotada \rightarrow_R , es la clausura compatible de R :

$$\begin{aligned} (M, N) \in R &\implies M \rightarrow_R N \\ M \rightarrow_R N &\implies ZM \rightarrow_R ZN \\ M \rightarrow_R N &\implies MZ \rightarrow_R NZ \\ M \rightarrow_R N &\implies \lambda x.M \rightarrow_R \lambda x.N \end{aligned}$$

- R -reducción, denotada \twoheadrightarrow_R , es la clausura reflexiva y transitiva de \rightarrow_R :

$$\begin{aligned} M \rightarrow_R N &\implies M \twoheadrightarrow_R N \\ M &\twoheadrightarrow_R M \\ M \twoheadrightarrow_R N, N \twoheadrightarrow_R L &\implies M \twoheadrightarrow_R L \end{aligned}$$

- R -convertibilidad, denotada $=_R$, es la relación de equivalencia generada por \twoheadrightarrow_R :

$$\begin{aligned} M \twoheadrightarrow_R N &\implies M =_R N \\ M =_R N &\implies N =_R M \\ M =_R N, N =_R L &\implies M =_R L \end{aligned}$$

Observación (Notación). Las nociones de reducción serán denotadas con letras en negritas, por ejemplo β y sus relaciones derivadas omitirán esta convención, por ejemplo \rightarrow_β , \twoheadrightarrow_β , $=_\beta$.

Ejemplo 2.3.1. Considerando la noción de reducción β

$$\begin{aligned} (\lambda x.x x)(\lambda y.y)z &\rightarrow_\beta (\lambda y.y)(\lambda y.y)z \\ &\rightarrow_\beta (\lambda y.y)z \\ &\rightarrow_\beta z \\ (\lambda x.x x)(\lambda y.y)z &\twoheadrightarrow_\beta z \\ z &=_\beta (\lambda x.x x)(\lambda y.y)z \end{aligned}$$

Una manera alternativa de definir nociones de reducción en el cálculo λ es establecer la transformación denotada como una contracción, por ejemplo, la noción de reducción α es

$$\alpha : (\lambda x.M) \rightarrow \lambda y.M[x := y] \text{ dado que } y \notin \text{FV}(M)$$

Ya que las nociones de reducción son conjuntos es posible tomar la unión de dos nociones de reducción para formar otra. Por ejemplo, la noción de reducción β no considera cambios de variables ligadas entre términos, sin embargo, la noción de reducción $\alpha\beta = \beta \cup \alpha$ si. De hecho, $=_{\alpha\beta}$ tiene las mismas propiedades que la convertibilidad $=_{\lambda}$ de la teoría formal λ . También se puede definir una noción de reducción para el axioma de extensionalidad de la teoría $\lambda\eta$ y estudiar las transformaciones en $\alpha \cup \beta \cup \eta$ es equivalente a estudiar la demostrabilidad en la teoría $\lambda\eta$:

$$\eta : \lambda x.M x \rightarrow M \text{ dado que } x \notin \text{FV}(M)$$

Proposición 2.3.1 (Equivalencia $\alpha\beta$ y λ). Para todo $M, N \in \Lambda$

$$M =_{\alpha\beta} N \iff \lambda \vdash M = N$$

Demostración. Se demuestran las implicaciones por separado

(\Leftarrow) Ya que $\lambda \vdash M = N$, existe un árbol de deducción que a partir de los axiomas y reglas de λ hasta la fórmula $M = N$. Las fórmulas del árbol se etiquetan con (α) , (β) , (ρ) , (μ) , (ν) , (ξ) , (τ) y (σ) de acuerdo al axioma o regla aplicada a la fórmula. En cada paso de la deducción se aplica la noción o clausura correspondiente de $\alpha\beta$ para pasar al siguiente paso:

- La noción α para el axioma (α) ;
- La noción β para el axioma (β) ;
- La clausura compatible de $\alpha\beta$ para las reglas (μ) , (ν) y (ξ) ;
- La clausura transitiva y reflexiva de $\alpha\beta$ para el axioma (ρ) y la regla (τ) ;
- La relación $=_{\alpha\beta}$ para la regla (σ) .

(\Rightarrow) Por las definiciones de las relaciones que genera $\alpha\beta$

- $M \rightarrow_{\alpha\beta} N \implies \lambda \vdash M = N$;
- $M \twoheadrightarrow_{\alpha\beta} N \implies \lambda \vdash M = N$;
- $M =_{\alpha\beta} N \implies \lambda \vdash M = N$.

□

Observación (Sobre notación). Dada esta equivalencia, se abrevia la noción $\alpha\beta$ simplemente como β ya que las propiedades interesantes de $\alpha\beta$ son las de la β -reducción.

Ya que las nociones de reducción son estudiadas como cálculos sobre un término, las siguientes definiciones nos permiten identificar las partes de un término λ que pueden o no ser transformadas por una noción de reducción R :

Definición 2.3.2 (Clasificación de términos). Dada una noción R , los términos λ pueden ser clasificados con respecto a esta noción de la siguiente manera:

- Un R -redex es un término M tal que $(M, N) \in R$ para algún término N ;
- Si $(M, N) \in R$, a N se le llama R -contractum de M ;
- Un término M es llamado una R -forma normal si M no contiene algún subtérmino que sea R -redex;
- Se dice que M tiene la R -forma normal N si $M =_R N$ y N es una R -forma normal.

La idea de que un término que sea una R -forma normal puede ser vista desde una perspectiva computacional como si una máquina realizara reducciones sobre un término M hasta llegar a un punto en donde no hay más posibles transformaciones que realizar, es entonces cuando el término resultante es el estado final de la máquina. Es por ello que en la literatura es común encontrar que se refieren a un término como “ M está en R -forma normal” en lugar de “ M es una R -forma normal”.

Ejemplo 2.3.2. Considerando las reducciones del ejemplo 2.3.1

- $(\lambda x.x x)(\lambda y.y)$ es un β -redex;
- $(\lambda y.y)z$ no está en β -forma normal;
- Estos términos tienen la β -forma normal z .

Las nociones de reducción pueden visualizarse y estudiarse como pseudográficas dirigidas. Sea R una noción de reducción, la R -gráfica de un término M , denotada $G_R(M)$ es la pseudográfica con el conjunto de vértices $\{N \in \Lambda \mid M \rightarrow_R N\}$ y conjunto de aristas \rightarrow_R .

Ejemplo 2.3.3 (β -Gráficas). Los siguientes son ejemplos de gráficas de reducción con la noción β :

1. $G_\beta(I x)$



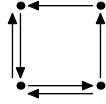
2. $G_\beta(I(I x))$



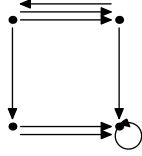
3. $G_\beta((\lambda x.x x)(\lambda x.x x))$



4. $G_\beta(W W W)$ con $W \equiv \lambda x y.x y y$



5. $G_\beta(M M)$ con $M \equiv \lambda x.(\lambda y.y y)x$



6. $G_\beta((\lambda x.x x x)(\lambda x.x x x))$



Un R -camino de reducción es una secuencia finita o infinita de R -contracciones

$$M_0 \xrightarrow{\Delta_0}_R M_1 \xrightarrow{\Delta_1}_R M_2 \rightarrow_R \dots$$

Donde Δ_i es el R -redex de M_i que es contraído en cada paso de las R -contracciones. Sea M un término λ , se puede categorizar su R -reducción a partir de $G_R(M)$ y sus caminos con las siguientes definiciones

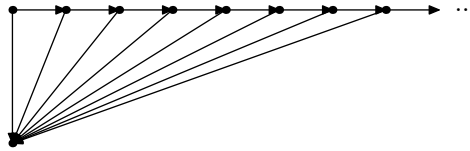
1. M R -normaliza fuertemente si no hay un R -camino de reducción infinito a partir de M ;
2. M es R -infinito si M no R -normaliza fuertemente;
3. R normaliza fuertemente, si para toda M , M R -normaliza fuertemente.

Ejemplo 2.3.4. Sea $G_\beta(M)$ una β -gráfica de un término M :

Es posible que M tenga una β -forma normal pero que $G_\beta(M)$ sea infinita. Si consideramos el término

$$M \equiv (\lambda x.I)((\lambda x.x x x)(\lambda x.x x x))$$

el cual tiene β -forma normal I , al dibujar su β -gráfica se obtiene



Si $G_\beta(M)$ es finita y M tiene una β -forma normal, es posible que M no β -normalice fuertemente. Si consideramos el término

$$M \equiv (\lambda x.I)((\lambda x.x x)(\lambda x.x x))$$

su β -forma normal es I y su β -gráfica es



Sin embargo, tiene el β -camino de reducción infinito

$$(\lambda x. I)((\lambda x. x x)(\lambda x. x x)) \xrightarrow{\beta^{(\lambda x. x x)(\lambda x. x x)}} (\lambda x. I)((\lambda x. x x)(\lambda x. x x)) \xrightarrow{\beta^{(\lambda x. x x)(\lambda x. x x)}} \dots$$

2.3.2 Confluencia

En la sección anterior se presenta una formalización del cálculo λ como un sistema de reducción, a partir de la noción β sobre los términos λ se puede estudiar el comportamiento de los términos bajo transformaciones ya sea en un paso, cero o muchos pasos y admitiendo transformaciones inversas. En general un sistema de reducción es una relación binaria sobre algún conjunto de objetos.

Consideremos por ejemplo un sistema de reducción para expresiones aritméticas con números naturales y operaciones $+$ y \times . Las transformaciones que realiza este sistema de reducción son muy similares a las que realiza una calculadora simplificada, por ejemplo:

$$\begin{aligned} (5 \times 3) + (9 \times 2) &\rightarrow 15 + (9 \times 2) \\ &\rightarrow 15 + 18 \\ &\rightarrow 33 \end{aligned}$$

Estas reducciones pudieron haberse realizado de una manera diferente, por ejemplo, de derecha a izquierda:

$$\begin{aligned} (5 \times 3) + (9 \times 2) &\rightarrow (5 \times 3) + 18 \\ &\rightarrow 15 + 18 \\ &\rightarrow 33 \end{aligned}$$

El resultado en ambos casos es el mismo y es natural pensar que, mientras se respeten las reglas de precedencia, las expresiones aritméticas podrán ser reducidas en cualquier orden y el resultado final será siempre el mismo.

Sin embargo, esta característica no es una propiedad general de todos los sistemas de

reducción. Consideremos un sistema con las siguientes reducciones:

$$\begin{aligned} f(0) &\rightarrow f(1) \\ f(1) &\rightarrow f(0) \\ f(g(x)) &\rightarrow g(x) \\ g(x) &\rightarrow x \end{aligned}$$

Al reducir $f(g(0))$ se tienen dos caminos

$$\begin{array}{ll} f(g(0)) \rightarrow g(0) & f(g(0)) \rightarrow f(0) \\ \rightarrow 0 & \rightarrow f(1) \\ & \rightarrow \dots \end{array}$$

Por un lado se obtiene el resultado final o el cual no puede ser reducido por ninguna regla, por otro lado se llega al término $f(1)$ el cual entra en un ciclo de reducciones del que no puede salir:

$$f(1) \rightarrow f(0) \rightarrow f(1) \rightarrow \dots$$

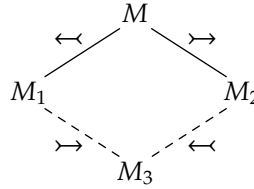
Al estudiar los sistemas de reducción desde un punto de vista computacional, independientemente de su utilidad, el segundo ejemplo puede nunca llegar a una forma normal (no tener un resultado final) y además el orden en el que se apliquen las reducciones es importante para determinar el resultado. En el caso del primer ejemplo, ya que la suma y la multiplicación son conmutativas y el orden de sus operandos no altera el resultado, por lo tanto el primer ejemplo no tendrá estas características indeseables.

La propiedad de los sistemas de reducción que se refiere a estas características es la *confluencia*. A continuación se aborda su definición considerando relaciones sobre términos λ .

Definición 2.3.3 (Propiedad de diamante). Sea \rightarrow una relación binaria sobre Λ . Se dice que \rightarrow satisface la *propiedad de diamante*, si para toda M, M_1 y M_2

$$M \rightarrow M_1 \text{ y } M \rightarrow M_2 \implies \text{existe } M_3 \text{ tal que } M_1 \rightarrow M_3 \text{ y } M_2 \rightarrow M_3$$

Gráficamente, esta propiedad se visualiza



Definición 2.3.4 (Confluencia global). Una noción de reducción R se dice ser *Church-Rosser*, abreviado CR, si \rightarrow_R tiene la propiedad de diamante. Cuando una relación de reducción es CR, se dice que tiene la propiedad de *confluencia global*.

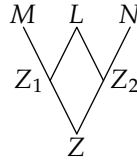
El siguiente teorema y corolario relacionan algunos conceptos abordados en la anterior sección con la propiedad de confluencia.

Teorema 2.3.2. Sea R una noción CR. Entonces

$$M =_R N \implies \text{existe } Z \text{ tal que } M \twoheadrightarrow_R Z \text{ y } N \twoheadrightarrow_R Z$$

Demostración. Por inducción de la definición de $=_R$:

- Si $M =_R N$ es una consecuencia directa de que $M \twoheadrightarrow_R N$, entonces $Z \equiv N$.
- Si $M =_R N$ es una consecuencia directa de que $N =_R M$, Z puede ser encontrada por la hipótesis de inducción.
- Si $M =_R N$ es una consecuencia directa de que $M =_R L$ y $L =_R N$, Z puede ser encontrada usando la hipótesis de inducción y la suposición de que \twoheadrightarrow_R es CR, gráficamente



□

Corolario. Sea R una noción CR. Entonces

1. Si N es una R -forma normal de M , entonces $M \twoheadrightarrow_R N$.
2. Un término M puede tener a lo más una R -forma normal.

Demostración. 1. Sea $M =_R N$ y N un término en R -forma normal. Para alguna Z se tiene que $M \twoheadrightarrow_R Z$ y $N \twoheadrightarrow_R Z$, pero ya que N está en forma normal $Z \equiv N$. Por lo tanto $M \twoheadrightarrow_R N$.

2. Supongamos que N_1 y N_2 son dos R -formas normales de M . Ya que $M \twoheadrightarrow_{N_1}$ y $M \twoheadrightarrow_{N_2}$, por la propiedad de confluencia, $N_1 \twoheadrightarrow_R Z$ y $N_2 \twoheadrightarrow_R Z$, pero ya que N_1 y N_2 no tienen subtérminos R -redex, $N_1 \equiv N_2 \equiv Z$.

□

Ya que la sustitución es una operación fundamental para definir las nociones de reducción en el cálculo λ , la siguiente definición establece una característica de las relaciones que se comportan bien bajo la sustitución.

Definición 2.3.5. Una noción de reducción R es *sustitutiva* si para todo término $\lambda M, N$ y L y toda variable x

$$(M, N) \in R \implies (M[x := L], N[x := L]) \in R$$

Es importante observar que si una noción R es sustitutiva, entonces las relaciones generadas \rightarrow_R , \twoheadrightarrow_R y $=_R$ también lo son.

Consideremos la noción β , considerando dos términos M y N tal que $(M, N) \in \beta$, se muestra que β es sustitutiva ya que

$$\begin{aligned} M &\equiv (\lambda y.P)Q \\ N &\equiv P[y:=Q] \\ M[x:=L] &\equiv (\lambda y.P[x:=L])Q[x:=L] \\ N[x:=L] &\equiv P[y:=Q][x:=L] \\ &\equiv P[x:=L][y:=Q[x:=L]] \end{aligned}$$

La propiedad de diamante está detrás de la definición de las relaciones Church-Rosser, otra propiedad similar que es importante considerar al estudiar nociones de reducción en el cálculo λ es la propiedad débil de diamante.

Definición 2.3.6 (Propiedad débil de diamante). Una relación binaria \succrightarrow satisface la propiedad *débil* de diamante si para todo término M, M_1 y M_2

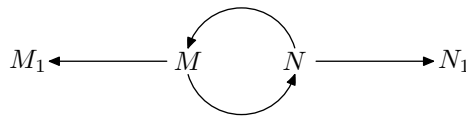
$$M \succrightarrow M_1 \text{ y } M \succrightarrow M_2 \implies \text{existe } M_3 \text{ tal que } M_1 \xrightarrow{*}_{=} M_3 \text{ y } M_2 \xrightarrow{*}_{=} M_3$$

donde $\xrightarrow{*}_{=}$ es la clausura transitiva y reflexiva de \succrightarrow .

Definición 2.3.7 (Confluencia local). Una noción de reducción R se dice ser Church-Rosser débil, abreviado WCR, si \rightarrow_R satisface la propiedad débil de diamante. Cuando una noción de reducción es WCR se dice que tiene la propiedad de *confluencia local*.

Se pudiera pensar que una noción R que sea Church-Rosser débil debe ser Church-Rosser, ya que la relación \rightarrow_R tiene como clausura transitiva y reflexiva a \twoheadrightarrow_R y todo lo que se puede reducir con \rightarrow_R se puede reducir con \twoheadrightarrow_R , sin embargo, esto no siempre se cumple.

Existen algunas nociones de reducción que son WCR pero no CR, consideremos la siguiente gráfica de reducción para una noción R a partir de un término M :



En este ejemplo se puede apreciar que la noción R es WCR ya que $M \rightarrow_R M_1$ y $M \rightarrow_R N$ y que también $M_1 \twoheadrightarrow_R M_1$ y $N \twoheadrightarrow_R M_1$, de manera similar comenzando desde M_1, N y N_1 . Sin embargo, la noción no es CR ya que $M \twoheadrightarrow_R M_1$ y $M \twoheadrightarrow_R N_1$, pero no existe un término Z en donde $M_1 \twoheadrightarrow_R Z$ y $N_1 \twoheadrightarrow_R Z$.

Alonzo Church y Barkley Rosser en un artículo publicado en 1936 [10] demostraron que la noción de reducción β es Church-Rosser y que por lo tanto, si un término λ podía ser reducido a dos términos diferentes, estos podían ser reducidos a un término en común [14, p. 14]. Este

resultado es de grán importancia para poder considerar al cálculo λ como un sistema que fundamente la computación.

A diferencia del cálculo λ , los lenguajes de programación suelen asignar un orden particular a la manera en la que sus expresiones son reducidas. Consideremos el fragmento de pseudocódigo 2.2.

Algoritmo 2.2 Potencial ciclo infinito

```

 $x \leftarrow 4$ 
si  $x \neq 4$  entonces
  repetir
    ...
  fin repetir
de lo contrario
  regresar  $x$ 
fin si

```

Si en la expresión condicional se reduce primero su consecuente el programa nunca terminará de ejecutarse, sin embargo, si se reduce de manera usual, es decir, primero el predicado $x \neq 4$ y dependiendo de su resultado se reducir o el consecuente o la alternativa, este programa terminará.

Estos problemas ocurren ya que, aunque el programa tenga una forma normal, automatizar el proceso de reducción puede causar que nunca se encuentre. En la siguiente sección se estudian diferentes estrategias para reducir términos en el cálculo λ .

2.3.3 Estrategias de reducción

Cuando un término λ tiene varios redex puede ser reducido de diferentes maneras. Por la propiedad de Church-Rosser no se obtendrán resultados esencialmente diferentes, sin embargo se pueden plantear estrategias de reducción para reducir un término de manera sistemática. Las estrategias de reducción son útiles, por ejemplo, para demostrar que un término no tiene forma normal [5, p. 324].

Definición 2.3.8 (Estrategia de reducción). Una estrategia de R -reducción es una función $F: \Lambda \rightarrow \Lambda$ tal que para toda M

$$M \rightarrow_R F(M)$$

Una estrategia de R -contracción es una estrategia de R -reducción, tal que para toda M que no esté en R -forma normal

$$M \rightarrow_R F(M)$$

En este trabajo consideraremos tres estrategias de reducción para el cálculo λ muy utilizadas en la especificación de lenguajes de programación [22]:

- Estrategia de *orden normal*;

- Estrategia de *llamada por nombre*;
- Estrategia de *llamada por valor*.

Para referirse a un redex en particular, es usual referirse a los redex más al exterior, esto se refiere a los redex que no son subtérminos de otro redex excepto ellos mismos. Por ejemplo, el término

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

tiene los siguientes redex

$$\begin{aligned} & \underline{(\lambda x.(\lambda y.I y)x)z} \\ & (\lambda x.(\lambda y.I y)x)z \\ & (\lambda x.(\lambda y.I y)x)z \end{aligned}$$

El primer redex es el que está más al exterior y el último redex es el que está más al interior. También es usual referirse a redex a la derecha o a la izquierda, por ejemplo, el término

$$(((I x)(I y))(I z))(I w))$$

tiene los redex

$$\begin{aligned} & (((\underline{I x})(I y))(I z))(I w)) \\ & (((I x)(\underline{I y}))(I z))(I w)) \\ & (((I x)(I y))(\underline{I z}))(I w)) \\ & (((I x)(I y))(I z))(\underline{I w))} \end{aligned}$$

El redex más a la izquierda es $(I x)$ y el más a la derecha es $(I w)$.

Orden normal

Esta estrategia establece que el redex que esté más a la izquierda y más al exterior es el que se reduce. Utilizar esta estrategia garantiza que si un término M tiene una forma normal N , entonces $F(M) = N$.

Considerando el término $\lambda M \equiv I(I(\lambda z.I z))$, su reducción bajo esta estrategia es

$$\begin{aligned} & \underline{I(I(\lambda z.I z))} \rightarrow_{\beta} I(\lambda z.I z) \\ & \underline{I(\lambda z.I z)} \rightarrow_{\beta} \lambda z.I z \\ & \lambda z.\underline{I z} \rightarrow_{\beta} \lambda z.z \end{aligned}$$

Una desventaja de utilizar esta estrategia es que los términos obtenidos al reducir pueden ser muy largos de escribir. Consideremos el término

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

Utilizando esta estrategia, en el primer paso se obtiene el término

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

Por lo tanto, las reducciones realizadas sobre $((\lambda y.I\ I\ I\ I\ y)z)$ se repetirán dos veces para obtener el término final.

Llamada por nombre

Esta estrategia es similar a la de orden normal, sin embargo, no se reducen apariciones de redex dentro de abstracciones.

Variantes de esta estrategia han sido utilizados en algunos lenguajes de programación populares como Algol-60 y Haskell.

Considerando el término $\lambda M \equiv I(I(\lambda z.I\ z))$, su reducción bajo esta estrategia es

$$\begin{aligned} \frac{I(I(\lambda z.I\ z))}{I(\lambda z.I\ z)} &\rightarrow_{\beta} I(\lambda z.I\ z) \\ \frac{I(\lambda z.I\ z)}{\lambda z.I\ z} &\rightarrow_{\beta} \lambda z.I\ z \end{aligned}$$

Con la estrategia llamada por nombre, algunos términos que no son formas normales no serán reducidos, por ejemplo el término

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

No es reducido siguiendo esta estrategia. De cierta manera, una abstracción $(\lambda x.M)$ “suspende” la reducción de M hasta que sea aplicada a algún término que no esté dentro de otra abstracción. Por este motivo, hay términos con forma normal que no pueden ser encontrados con llamada por nombre.

Llamada por valor

Esta estrategia solo reduce los redex más al exterior del término λ y un redex es reducido únicamente cuando los redex a su derecha estén en forma normal.

Variantes de esta estrategia han sido utilizadas en algunos lenguajes de programación populares como C y Java. Si consideramos que un redex es una llamada a función, esta estrategia establece que antes de reducir la función, se reducen todos sus argumentos, de derecha a izquierda, de tal manera que al “entrar” al cuerpo de la función, todos los valores son conocidos.

Considerando el término $\lambda M \equiv I(I(\lambda z.I z))$, su reducción bajo esta estrategia es

$$\begin{aligned} \underline{I(I(\lambda z.I z))} &\rightarrow_{\beta} I(\lambda z.I z) \\ \underline{I(\lambda z.I z)} &\rightarrow_{\beta} \lambda x.I z \end{aligned}$$

Con esta estrategia se pueden obtener términos bastante cortos, considerando el término

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

Primero se reduce $(\lambda y.I I I I y)z$ a z y posteriormente se reduce $(\lambda x.x x x)z$, por lo que en este ejemplo no se presenta la duplicación de reducciones como en el caso de orden normal.

Una grán desventaja de esta estrategia es que hay términos con formas normales a las que no se puede reducir. Consideremos el término

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

Es fácil notar que este término tiene forma normal y , sin embargo, siguiendo la estrategia de llamada por nombre, el término $(\lambda x.x x)(\lambda x.x x)$ será reducido hasta llegar a su forma normal, pero este redex no tiene forma normal y por consiguiente nunca se obtendrá el resultado.

Esta desventaja es resuelta por los lenguajes de programación basados en llamada por nombre forzando el orden de evaluación de ciertas expresiones.

Capítulo 3

Codificación de objetos

Alonzo Church y Alan Turing hipotetizaron en 1936 de manera independiente que el cálculo λ y las máquinas de Turing formalizaban el concepto de cómputo [?, ?]. Esto puede parecer extraño ya que claramente los programas de computadora pueden realizar cálculos complejos que involucran números, texto, árboles, gráficas o conjuntos mientras que en el cálculo λ se está limitado a átomos, abstracciones y aplicaciones.

En este capítulo se presenta un tratamiento computacional del cálculo λ con el objetivo de explorar el tipo de conceptos matemáticos y algorítmicos que se pueden representar y corroborar informalmente que todo lo que se puede hacer con un lenguaje de programación se puede hacer con el cálculo λ .

En la primer sección se plantean codificaciones para valores de verdad y operaciones booleanas, con esto se construye en el cálculo una operación similar a la sentencia condicional de los lenguajes de programación; en la segunda sección se exploran los números naturales y las operaciones aritméticas elementales, con esto se plantea un mecanismo de iteración; en la tercera sección se deriva un mecanismo de recursividad que permitirá representar algoritmos recursivos de manera sencilla; en la cuarta sección se presentan codificaciones de las listas, árboles y gráficas, finalmente se presenta la codificación de términos λ .

Este capítulo está fuertemente influenciado por la serie de reportes técnicos llamados en la comunidad de lenguajes de programación como “The Lambda Papers” por Sussman y Steele [29, ?, ?, 28, ?].

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 λ 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 λ en sí, si no a propiedades de estos,

por ejemplo, es falso que $\|\lambda x.x\| = 5$ y es verdadero que $(K\ x) \rightarrow_{\beta} (\lambda x.y)$. 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 λ 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 decisión 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 λ , 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.

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 decisión (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 λ 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 λM y N se β -reduce a M si la oración O es verdadera y a N si es falsa:

$$P O \rightarrow_{\beta} D,$$

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

Para fines prácticos no es necesario saber cómo es P ni O , lo importante es que cuando O es cierta, D elegirá M y si O es falsa, elegirá N . Por lo tanto, $(P O) = D$ es un término λ 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 λ 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 λ como los términos T y F respectivamente.

$$T \equiv \lambda x y.x$$

$$F \equiv \lambda x y.y$$

Utilizar T y F en términos λ 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

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

y la negación es una operación unaria definida en

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

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 λ , por ejemplo, suponiendo que \wedge , \vee , \neg son términos λ , las expresiones mencionadas escribirían con notación de prefijo:

$$\vee(\wedge T F)F$$

$$\neg(\forall F F)$$

$$\wedge T(\forall F F)$$

Al igual que los valores de verdad, las operaciones básicas son codificadas como abstracciones del cálculo λ . Hay varias metodologías para derivar términos λ para las operaciones booleanas a partir de T y 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 λ en $\{T, F\}^+$.

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 λ a partir de F y 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.

FF	$(\lambda x y.y)F \rightarrow_{\beta} \lambda y.y \equiv I$
FT	$(\lambda x y.y)T \rightarrow_{\beta} \lambda y.y \equiv I$
TF	$(\lambda x y.x)F \rightarrow_{\beta} \lambda y.F \equiv KF$
TT	$(\lambda x y.x)T \rightarrow_{\beta} \lambda y.T \equiv KT$

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:

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

$$\lambda \vdash (FFM) = M$$

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

$$\lambda \vdash (FNM) = M$$

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

$$\lambda \vdash (TFM) = F$$

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

$$\lambda \vdash (T N M) = N$$

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

$$F F T \rightarrow_{\beta} T$$

$$T F T \rightarrow_{\beta} F$$

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

$$P F T \rightarrow_{\beta} \neg P$$

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

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

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

$$\neg F \equiv (\lambda p.p F T)F$$

$$\rightarrow_{\beta} F F T$$

$$\rightarrow_{\beta} T$$

$$\neg T \equiv (\lambda p.p F T)T$$

$$\rightarrow_{\beta} T F T$$

$$\rightarrow_{\beta} F$$

Las reducciones del cuadro 3.2 se pueden aplicar a F y 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 algunos patrones en los resultados de las reduccio-

FFF	$IF \rightarrow_{\beta} F$
FFT	$IT \rightarrow_{\beta} T$
FTF	$IF \rightarrow_{\beta} F$
FTT	$IT \rightarrow_{\beta} T$
TFF	$KFF \rightarrow_{\beta} F$
TFT	$KFT \rightarrow_{\beta} F$
TTF	$KTF \rightarrow_{\beta} T$
TTT	$KTT \rightarrow_{\beta} T$

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

nes, por ejemplo, si P es un valor de verdad cualquiera, $(P F F)$ se reduce a F y $(P T T)$ se reduce a 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 $P, Q \in \{F, T\}$. La cantidad de combinaciones de estos valores es $2 \times \binom{3}{2} = 2 \times \frac{3!}{2!} = 6$ y son $(P Q F)$, $(P Q T)$, $(P F Q)$, $(P T Q)$, $(F P Q)$, $(T P Q)$. 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	$P Q F$	$P Q T$	$P F Q$	$P T Q$	$F P Q$	$T P Q$
F	F	F	T	F	F	F	F
F	T	F	T	T	T	T	F
T	F	F	F	F	T	F	T
T	T	T	T	F	T	T	T

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

Las columnas de la combinación $(P Q F)$ y $(P T Q)$ 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: $(P Q T)$ es la implicación material; $(P F Q)$ es la no implicación inversa; $(F P Q)$ es la proyección de Q ; y $(T P Q)$ es la proyección de P .

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

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

$$\wedge \equiv \lambda p q. p q F$$

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

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

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

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

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

Definición 3.1.4 (Operación de disyunción). El término \vee que representa la disyunción es

$$\vee \equiv \lambda p q.p T q$$

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

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

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

$$\begin{aligned}
\forall T F &\equiv (\lambda p q.p T q) T F \\
&\rightarrow_{\beta} T T F \\
&\rightarrow_{\beta} T
\end{aligned}$$

$$\begin{aligned}
\forall T T &\equiv (\lambda p q.p T q) T T \\
&\rightarrow_{\beta} T T T \\
&\rightarrow_{\beta} T
\end{aligned}$$

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 construidas. 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 construir las operaciones booleanas como términos λ es la de partir de un algoritmo 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 $(T M N) \rightarrow_{\beta} M$ y $(F M N) \rightarrow_{\beta} N$, si $M \rightarrow_{\beta} M'$ y $N \rightarrow_{\beta} N'$, entonces

$$T M N \rightarrow_{\beta} M'$$

$$F M N \rightarrow_{\beta} N'$$

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

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

Esta aplicación de un valor de verdad a dos términos λ 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 λ 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

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

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 λ como

$$\neg \equiv \lambda p. \prec p F T$$

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

3.1.2

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

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

Traducido al cálculo λ como

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

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 falso
fin si
de lo contrario
  regresar falso
fin si

```

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

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

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 λ como

$$\vee \equiv \lambda p_1 p_2. \prec p_1 T (\prec p_2 T F)$$

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

$$\begin{aligned}
\lambda p_1 p_2. < p_1 T(< p_2 T F) &\equiv \lambda p_1 p_2. (\lambda p m n. p m n) p_1 T((\lambda p m n. p m n) p_2 T F) \\
&\rightarrow_\beta \lambda p_1 p_2. p_1 T((\lambda p m n. p m n) p_2 T F) \\
&\rightarrow_\beta \lambda p_1 p_2. p_1 T(p_2 T F) \\
&=_{\lambda} \lambda p_1 p_2. p_1 T p_2
\end{aligned}$$

Utilizando esta técnica, se puede obtener el término λ 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 construir un término λ 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	$P \odot Q$
F	F	x_1
F	T	x_2
T	F	x_3
T	T	x_4

El procedimiento generalizado es

Algoritmo 3.4 Operación booleana \odot dado x_1, x_2, x_3, x_4

Entrada: $p_1, p_2 \in \{\text{falso}, \text{verdadero}\}$

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 λ es

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

Ejemplo 3.1.1 (Operaciones *NAND* y *NOR*). Estas operaciones booleanas binarias conforman los conjuntos unitarios $\{\text{NAND}\}$ y $\{\text{NOR}\}$ los cuales son conjuntos funcionalmente 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	$P \uparrow Q$
F	F	T
F	T	T
T	F	T
T	T	F

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

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

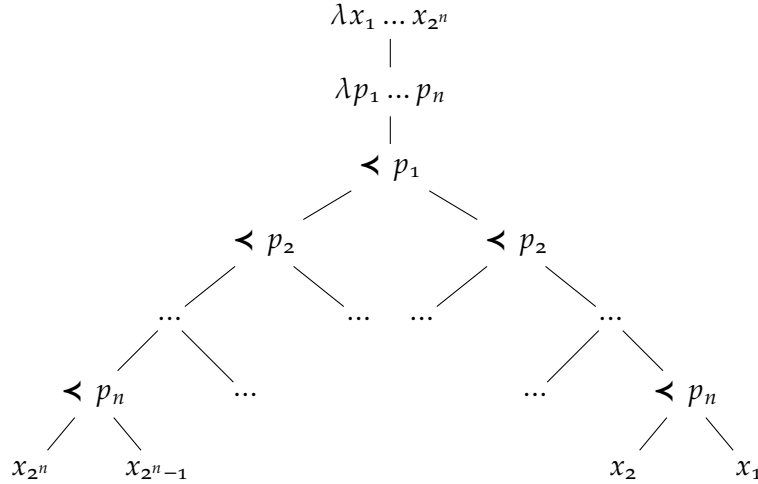
La operación *NOR* se denota $P \downarrow Q$ y tiene la siguiente tabla de verdad

P	Q	$P \downarrow Q$
F	F	T
F	T	F
T	F	F
T	T	F

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

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

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 λ , 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 λ de los valores de ésta álgebra trivalente son

$$T \equiv \lambda x y z. x$$

$$F \equiv \lambda x y z. y$$

$$U \equiv \lambda x y z. z$$

Al igual que en la codificación bivalente, se puede codificar un término \leftarrow_3 , similar a \leftarrow de la definición 3.1.5 pero con tres ramificaciones

Definición 3.1.8 (Condicional trivalente).

$$\leftarrow_3 \equiv \lambda p m n o. p m n o$$

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

$$(\leftarrow_3 P M N O) \rightarrow_\beta \begin{cases} M & P \equiv T; \\ N & P \equiv F; \\ O & P \equiv U. \end{cases}$$

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

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

El procedimiento en pseudocódigo que la describe es:

Algoritmo 3.5 Operación booleana \odot dado x_1, x_2, \dots, x_9

Entrada: $p_1, p_2 \in \{\text{verdadero, falso, desconocido}\}$

Salida: Valor x_i de la tabla de verdad

```

si  $p_1 = \text{verdadero}$  entonces
  si  $p_2 = \text{verdadero}$  entonces
    regresar  $x_1$ 
  de lo contrario y si  $p_2 = \text{falso}$  entonces
    regresar  $x_2$ 
  de lo contrario y si  $p_2 = \text{desconocido}$  entonces
    regresar  $x_3$ 
  fin si
de lo contrario y si  $p_1 = \text{falso}$  entonces
  si  $p_2 = \text{verdadero}$  entonces
    regresar  $x_4$ 
  de lo contrario y si  $p_2 = \text{falso}$  entonces
    regresar  $x_5$ 
  de lo contrario y si  $p_2 = \text{desconocido}$  entonces
    regresar  $x_6$ 
  fin si
de lo contrario y si  $p_1 = \text{desconocido}$  entonces
  si  $p_2 = \text{verdadero}$  entonces
    regresar  $x_7$ 
  de lo contrario y si  $p_2 = \text{falso}$  entonces
    regresar  $x_8$ 
  de lo contrario y si  $p_2 = \text{desconocido}$  entonces
    regresar  $x_9$ 
  fin si
fin si

```

Traducido al cálculo λ 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

$$R_1 \equiv (\prec_3 p_2 x_1 x_2 x_3)$$

$$R_2 \equiv (\prec_3 p_2 x_4 x_5 x_6)$$

$$R_3 \equiv (\prec_3 p_2 x_7 x_8 x_9)$$

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 λ , los números naturales no son términos λ , sin embargo, desde el metalenguaje se pueden manejar números naturales y expresiones aritméticas como por ejemplo en la definición de longitud 2.1.3. Al igual que el álgebra booleana, las expresiones aritméticas pueden ser codificadas como términos λ .

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 λ de manera similar a como se abordó en 3.1, también se aborda la representación de la noción de iteración y algunos mecanismos que nos permiten abstraer 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

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

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 extiende 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 \hat{n} , asociado al número natural n es

$$\hat{n} \equiv (\lambda x y. x^n y) \quad (3.1)$$

Ejemplo 3.2.1. Los primeros 5 numerales son:

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

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 T y F se combinan y corroborar manualmente que una combinación se reduce a otra. Sin embargo, al aplicar un numeral de Church \hat{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 \hat{n} y términos cualesquiera P y Q .

$$\hat{n} P \rightarrow_{\beta} \lambda x. P^n x \quad (3.2)$$

$$\hat{n} P Q \rightarrow_{\beta} (\lambda x. P^n x) Q \rightarrow_{\beta} P^n Q \quad (3.3)$$

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).

$$(\widehat{n} P Q)^m R \rightarrow_{\beta} (P^n Q)^m R \quad \text{Por (3.3)} \quad (3.4)$$

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.

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

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

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

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

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

Consideremos la reducción de una aplicación de numerales $(\widehat{n} \widehat{m})$. Cuando $\widehat{n} \equiv \widehat{o}$ se tiene que para todo natural m , $(\widehat{o} \widehat{m}) \rightarrow_{\beta} I$ debido a que $\widehat{o} \equiv F$ y $(F M) \rightarrow_{\beta} I$. 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:

$$\begin{aligned}
 \widehat{1}\widehat{0} &\rightarrow_{\beta} (\lambda x. \widehat{0}^1 x) && \text{Por (3.2)} \\
 &\rightarrow_{\beta} (\lambda x. \widehat{0}^1 x) && \text{Por (3.6)} \\
 &\rightarrow_{\beta} (\lambda x y. x^0 y) && \text{Por (3.2)} \\
 &\equiv \widehat{0} \\
 \widehat{1}\widehat{1} &\rightarrow_{\beta} (\lambda x. \widehat{1}^1 x) && \text{Por (3.2)} \\
 &\rightarrow_{\beta} (\lambda x. \widehat{1}^1 x) && \text{Por (3.6)} \\
 &\rightarrow_{\beta} (\lambda x y. x^1 y) && \text{Por (3.2)} \\
 &\equiv \widehat{1} \\
 \widehat{1}\widehat{2} &\rightarrow_{\beta} (\lambda x. \widehat{2}^1 x) && \text{Por (3.2)} \\
 &\rightarrow_{\beta} (\lambda x. \widehat{2}^1 x) && \text{Por (3.6)} \\
 &\rightarrow_{\beta} (\lambda x y. x^2 y) && \text{Por (3.2)} \\
 &\equiv \widehat{2}
 \end{aligned}$$

Para cualquier numeral \widehat{m} :

$$\widehat{1}\widehat{m} \rightarrow_{\beta} \lambda x. \widehat{m}^1 x \rightarrow_{\beta} \lambda x. \widehat{m}^1 x \rightarrow_{\beta} \lambda x y. x^m y \equiv \widehat{m}$$

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

$$\widehat{2}\widehat{m} \rightarrow_{\beta} \lambda x. \widehat{m}^2 x \rightarrow_{\beta} \lambda x. \widehat{m}^2 x \rightarrow_{\beta} \lambda x y. x^{m^2} y \equiv \widehat{m}^2$$

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

$$\widehat{n}\widehat{m} \rightarrow_{\beta} \lambda x. \widehat{m}^n x \rightarrow_{\beta} \lambda x. \widehat{m}^n x \rightarrow_{\beta} \lambda x y. x^{m^n} y \equiv \widehat{m}^n \quad (3.7)$$

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 $\widehat{\odot} \equiv (\lambda x y. x^{a \odot b} y)$, 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 \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

$$\widehat{\uparrow} \equiv \lambda m n. \lambda x y. n m x y \quad (3.8)$$

En el caso de la codificación de la multiplicación, denotada \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

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

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

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

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.

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

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 decisión, 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 expresiones 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 λ 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:

$$P Q \rightarrow_{\beta} R,$$

$$R A O \rightarrow_{\beta} \underbrace{A(A(\dots (A(A O)) \dots))}_{n \text{ veces}}$$

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 λ de la forma

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

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.

$$m \times n = \underbrace{m + m + \dots + m}_{n \text{ veces}} = \sum_{i=1}^n m \quad (3.11)$$

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.

$$m^n = \underbrace{m \times m \times \dots \times m}_{n \text{ veces}} = \prod_{i=1}^n m \quad (3.12)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 construir 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

$$\widehat{+} \widehat{m} \equiv (\lambda m n. n \widehat{+}_1 m) \widehat{m} \rightarrow_{\beta} \lambda n. n \widehat{+}_1 \widehat{m}$$

El cual al ser aplicado a algún 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

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

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

$$\begin{aligned}
\widehat{n}(\widehat{+}\widehat{m})\widehat{o} &\rightarrow_{\beta} (\widehat{+}\widehat{m})^n\widehat{o} && \text{Por (3.3)} && (3.17) \\
&\equiv (\widehat{+}\widehat{m})^{n-1}(\widehat{+}\widehat{m}\widehat{o}) && \text{Por (2.3)} \\
&\rightarrow_{\beta} (\widehat{+}\widehat{m})^{n-1}\widehat{m+o} && \text{Por (3.15)} \\
&\equiv (\widehat{+}\widehat{m})^{n-2}(\widehat{+}\widehat{m}\widehat{m+o}) && \text{Por (2.3)} \\
&\rightarrow_{\beta} (\widehat{+}\widehat{m})^{n-2}\widehat{m \times 2 + o} && \text{Por (3.15)} \\
&\dots \\
&\rightarrow_{\beta} (\widehat{+}\widehat{m})^{n-n}\widehat{m \times n + o} \\
&\equiv \widehat{m \times n} && \text{Por (2.3)}
\end{aligned}$$

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

$$\widehat{\times} \equiv \lambda m n. n(\widehat{+} m)\widehat{o} \quad (3.18)$$

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

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

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

$$\begin{aligned}
\widehat{n}(\widehat{\times} \widehat{m})\widehat{1} &\rightarrow_{\beta} (\widehat{\times} \widehat{m})^n \widehat{1} && \text{Por (3.3)} && (3.19) \\
&\equiv (\widehat{\times} \widehat{m})^{n-1} (\widehat{\times} \widehat{m} \widehat{1}) && \text{Por (2.3)} \\
&\rightarrow_{\beta} (\widehat{\times} \widehat{m})^{n-1} \widehat{\overline{m \times 1}} && \text{Por (3.17)} \\
&\equiv (\widehat{\times} \widehat{m})^{n-2} (\widehat{\times} \widehat{m} \widehat{\overline{m \times 1}}) && \text{Por (2.3)} \\
&\rightarrow_{\beta} (\widehat{\times} \widehat{m})^{n-2} \widehat{\overline{m^2 \times 1}} && \text{Por (3.17)} \\
&\dots \\
&\rightarrow_{\beta} (\widehat{\times} \widehat{m})^{n-n} \widehat{\overline{m^n \times 1}} \\
&\equiv \widehat{m^n}
\end{aligned}$$

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

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

Comparando esta segunda aproximación de las codificaciones 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 estructural entre las codificaciones definidas, las similitudes menos importantes se encuentran coloreadas en gris.

$\widehat{+}$	\equiv	$\lambda m n. n \widehat{\overline{+_1 m}}$	\equiv	$(\lambda m. (\lambda n. ((n \widehat{\overline{+_1}}) m)))$
$\widehat{\times}$	\equiv	$\lambda m n. n(\widehat{\overline{\times m}})\widehat{0}$	\equiv	$(\lambda m. (\lambda n. ((n(\widehat{\overline{\times m}}))\widehat{0})))$
$\widehat{\uparrow}$	\equiv	$\lambda m n. n(\widehat{\overline{\times m}})\widehat{1}$	\equiv	$(\lambda m. (\lambda n. ((n(\widehat{\overline{\times m}}))\widehat{1})))$

Figura 3.2: Codificaciones de adición, multiplicación y exponenciación

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 azul se encuentran los valores iniciales a los que se aplica el procedimiento, estos corresponden al caso trivial de la operación, es decir, 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 exponenciación 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 \hat{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 Barendregt, en el artículo titulado “The Impact of Lambda Calculus in Logic and Computer Science” [4], menciona que la codificación del predecesor en el cálculo λ 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 λ era un sistema

demasiado débil para representar el predecesor; es entonces que Church, habiendo aprendido que el predecesor era definible en el cálculo λ , se convenció de que todas las funciones que eran intuitivamente computables, eran definibles en el cálculo λ .

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

$$\widehat{0}' \equiv \lambda x y z. y$$

$$\widehat{1}' \equiv \lambda x y z. z y$$

$$\widehat{2}' \equiv \lambda x y z. x(z y)$$

$$\widehat{3}' \equiv \lambda x y z. x(x(z y))$$

$$\widehat{4}' \equiv \lambda x y z. x(x(x(z y)))$$

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 $(\widehat{n}' x y I)$ resulta en $(x^{n-1} y)$.

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

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

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

Donde $\mathcal{T}[n \mapsto n']$ es una transformación que a partir de n obtiene el mismo número pero

con la codificación modificada. Con la cual es sencillo encontrar $n - 1$.

Para construir el término $\mathcal{T}[n \mapsto n']$ se debe encontrar una manera de contar desde o 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 $\widehat{n+1}'$:

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

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 sustituida por $(z y)$ en el sucesor. Cuando se computa el sucesor de $\widehat{1}'$, se tienen variables enlazadas z y y , si y es sustituido por $(z y)$, la z deberá ser sustituida 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 sustituida por x en el sucesor. Para corroborar que estas sustituciones son correctas, se considera la aplicación $(\widehat{3}' x (z y) x)$:

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

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

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

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

$$\widehat{-}_1 \equiv \lambda n. \lambda x y. (n \widehat{+}'_1 \widehat{o}') x y I \quad (3.21)$$

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

$$\begin{aligned} \widehat{-}_1 \widehat{o} &\rightarrow_{\beta} \lambda x y. (\widehat{o} \widehat{+}'_1 \widehat{o}') x y I \\ &\rightarrow_{\beta} \lambda x y. (I \widehat{o}') x y I \\ &\rightarrow_{\beta} \lambda x y. \widehat{o}' x y I \\ &\equiv \lambda x y. (\lambda x y z. y) x y I \\ &\rightarrow_{\beta} \lambda x y. y \\ &\equiv \widehat{o} \end{aligned}$$

$$\begin{aligned} \widehat{-}_1 \widehat{k} &\rightarrow_{\beta} \lambda x y. (\widehat{k} \widehat{+}'_1 \widehat{o}') x y I \\ &\rightarrow_{\beta} \lambda x y. \widehat{+}'_1{}^k \widehat{o}' x y I \\ &\rightarrow_{\beta} \lambda x y. \widehat{k}' x y I \\ &\equiv \lambda x y. (\lambda x y z. x^{k-1}(z y)) x y I \\ &\rightarrow_{\beta} \lambda x y. x^{k-1}(I y) \\ &\rightarrow_{\beta} \lambda x y. x^{k-1} y \\ &\equiv \widehat{k-1} \end{aligned}$$

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 .

$$\widehat{-} \equiv \lambda m n. n \widehat{-}_1 m \quad (3.22)$$

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 $m - n$.

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

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 $(\hat{n}(\hat{-} \hat{m})\hat{o})$ 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, $(\hat{n}(\hat{-} \hat{m})\hat{o})$ se reduce a \hat{m} cuando \hat{n} es impar y a \hat{o} cuando \hat{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 \hat{o} , 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 λ . En lugar de codificar el *mientras*, se puede utilizar \hat{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 \hat{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 $(\hat{n} P B)$ 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 T y F para *decidir* el valor del estado que se desea obtener.

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

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

$$S \equiv \lambda n_1 n_2. \lambda p.p n_1 n_2 \quad (3.23)$$

Para seleccionar de un determinado valor a partir de un estado, se plantean abstracciones λ 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

$$S_r \equiv \lambda s.s T \quad (3.24)$$

$$S_a \equiv \lambda s.s F \quad (3.25)$$

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, conociendo un numeral \hat{n} , el término B se codifica como

$$B \equiv S \hat{n} \hat{1} \quad (3.26)$$

El término P debe ser una abstracción que sea aplicada a un estado y sea reducida a un estado, ya que en el cálculo λ 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 \hat{r} y \hat{a} son los valores del estado previo, el término P debe reducirse a un estado en donde el primer elemento sea $(\hat{\neg}_1 \hat{r})$ y el segundo sea $(\hat{\times} \hat{a} \hat{r})$. Utilizando los selectores S_r y S_a , el término P se codifica como

$$P \equiv \lambda s.S(\hat{\neg}_1(S_r s))(\hat{\times}(S_a s)(S_r s)) \quad (3.27)$$

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

$$\hat{!} \equiv \lambda n.n(\lambda s.S(\hat{\neg}_1(S_r s))(\hat{\times}(S_a s)(S_r s)))(S n \hat{1}) \quad (3.28)$$

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

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

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 ${}_n v_1, {}_n v_2, \dots, {}_n v_n$, términos de la forma

$${}_n v_i \equiv \lambda x_1 \dots x_n. x_i$$

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

$${}_n S \equiv \lambda x_1 \dots x_n. \lambda p. p x_1 \dots x_n \quad (3.30)$$

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

$${}_n S_i \equiv \lambda s. s v_i \quad (3.31)$$

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 T si $n = 0$ y a F si $n > 0$. El $\widehat{0}$ no tiene aplicaciones internas y simplemente es reducido al segundo término al que fue aplicado, por lo tanto $(\widehat{n} P T)$ debe reducirse a T cuando $n = 0$. Cuando el numeral es mayor a cero, la primera aplicación de $(P T)$ debe reducirse a F y las siguientes aplicaciones deben ser $(P F)$ y reducirse también a F . Un término que al ser aplicado a cualquier término es reducido a F es $(K F)$. Por lo tanto, la codificación de este predicado se define

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

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

$$\begin{aligned}
\widehat{o}_? \widehat{o} &\equiv (\lambda n. n(KF)T)\widehat{o} \\
&\rightarrow_{\beta} \widehat{o}(KF)T \\
&\rightarrow_{\beta} (KF)^0 T \\
&\equiv T \\
\widehat{o}_? \widehat{n} &\equiv (\lambda n. n(KF)T)\widehat{n} \\
&\rightarrow_{\beta} \widehat{n}(KF)T \\
&\rightarrow_{\beta} (KF)^n T \\
&\equiv (KF)^{n-1}(KF)T \\
&\rightarrow_{\beta} (KF)^{n-1}F \\
&\equiv (KF)^{n-2}(KF)F \\
&\rightarrow_{\beta} (KF)^{n-2}F \\
&\dots \\
&\rightarrow_{\beta} (KF)^{n-n}F \\
&\equiv F
\end{aligned}$$

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{o} si \widehat{m} es menor o igual a \widehat{n} . Por lo tanto, la codificación de este predicado se define

$$\widehat{\leq}_? \equiv \lambda m n. \widehat{o}_? (\widehat{-} m n) \quad (3.33)$$

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 haciendo uso de propiedades numéricas.

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

$$\widehat{\geq}_? \equiv \lambda m n. \widehat{\leq}_? n m \quad (3.34)$$

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

$$\widehat{=}_? \equiv \lambda m n. \Lambda (\widehat{\leq}_? m n)(\widehat{\geq}_? m n) \quad (3.35)$$

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

$$\widehat{>}_? \equiv \lambda m n. \neg (\widehat{\leq}_? m n) \quad (3.36)$$

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

$$\widehat{<}_? \equiv \lambda m n. \neg(\widehat{\leq}_? m n) \quad (3.37)$$

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 $\min(m, n) = n$:

$$\widehat{\min} \equiv \lambda m n. \prec(\widehat{<}_? m n) m n \quad (3.38)$$

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

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

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

$$m - n = x \iff m = x + n$$

$$m/n = x \iff m = x \times n$$

$$\log_n m = x \iff m = n^x$$

$$\sqrt[n]{m} = x \iff x^n = m$$

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.

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

Entrada: $m, n, t \in \mathbb{N}$, $\odot: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $err: \mathbb{N} \times \mathbb{N} \rightarrow \{\text{verdadero, falso}\}$

Salida: $m \odot^{-1} n$

si $err(m, n)$ **entonces**

regresar t

de lo contrario

$x \leftarrow 0$

$a \leftarrow x \odot n$

mientras $a < m$ **hacer**

$a \leftarrow x \odot n$

$x \leftarrow x + 1$

fin mientras

regresar a

fin si

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 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.

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

$$: \equiv {}_2S$$

$$A \equiv {}_2S_1$$

$$X \equiv {}_2S_2$$

Algoritmo 3.8 Cálculo de $m \odot^{-1} n$ **Entrada:** $m, n, t \in \mathbb{N}$, $\odot: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $err: \mathbb{N} \times \mathbb{N} \rightarrow \{\text{verdadero, falso}\}$ **Salida:** $m \odot^{-1} n$

```

si  $err(m, n)$  entonces
  regresar  $t$ 
de lo contrario
   $x \leftarrow 0$ 
   $a \leftarrow x \odot n$ 
  para iteraciones hasta  $m$  hacer
    si  $a < m$  entonces
       $a \leftarrow x \odot n$ 
       $x \leftarrow x + 1$ 
    de lo contrario
       $a \leftarrow a$ 
    fin si
  fin para
  regresar  $a$ 
fin si

```

La codificación del algoritmo 3.8 es

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

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

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

$$\widehat{\text{f}} \equiv \widehat{\odot}^{-1} \widehat{\times} \widehat{\text{z}}_? \widehat{\text{1}} \quad (3.42)$$

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

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

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* [17], 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:

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

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

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

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

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

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

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 $\widehat{\uparrow}_i$

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

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 construir 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 $(\widehat{\mathcal{H}} \widehat{n}) \rightarrow_{\beta} \widehat{\mathcal{H}}(\widehat{n})$.

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

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

$$\begin{aligned} \widehat{\mathcal{H}} \widehat{1} &\equiv (\lambda n. n(\lambda f. \lambda m n. n(f m) \widehat{1}) \widehat{\times}) \widehat{1} \\ &\rightarrow_{\beta} \widehat{1}(\lambda f. \lambda m n. n(f m) \widehat{1}) \widehat{\times} \\ &\rightarrow_{\beta} (\lambda f. \lambda m n. n(f m) \widehat{1}) \widehat{\times} \\ &\rightarrow_{\beta} \lambda m n. n(\widehat{\times} m) \widehat{1} \\ &\equiv \widehat{\uparrow} \\ &\equiv \widehat{\uparrow}_1 \end{aligned}$$

$$\begin{aligned}
\widehat{\mathcal{H}}\widehat{i} &\equiv (\lambda n.n(\lambda f.\lambda m n.n(f m)\widehat{1})\widehat{\times})\widehat{i} \\
&\rightarrow_{\beta} \widehat{i}(\lambda f.\lambda m n.n(f m)\widehat{1})\widehat{\times} \\
&\rightarrow_{\beta} (\lambda f.\lambda m n.n(f m)\widehat{1})^i\widehat{\times} \\
&\equiv (\lambda f.\lambda m n.n(f m)\widehat{1})^{i-1}((\lambda f.\lambda m n.n(f m)\widehat{1})\widehat{\times}) \\
&\rightarrow_{\beta} (\lambda f.\lambda m n.n(f m)\widehat{1})^{i-1}(\lambda m n.n(\widehat{\times} m)\widehat{1}) \\
&\equiv (\lambda f.\lambda m n.n(f m)\widehat{1})^{i-1}\widehat{\uparrow}_1 \\
&\dots \\
&\rightarrow_{\beta} (\lambda f.\lambda m n.n(f m)\widehat{1})^{i-i}\widehat{\uparrow}_i \\
&\equiv \widehat{\uparrow}_i
\end{aligned}$$

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 [12].

Un ejemplo de definición recursiva es el de términos λ . 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* [12]. 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 λ 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] [28].

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:

$$\begin{aligned} \hat{!} &\equiv \lambda n. (\prec (\hat{0}_? n) \\ &\quad \hat{1} \\ &\quad (\hat{\times} n (\hat{!} (\hat{-}_1 n)))) \end{aligned} \tag{3.47}$$

El problema con esta definición es que antes de definir $\hat{!}$, se hace referencia a ella en $(\hat{!}(\hat{-}_1 n))$, 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 $\hat{!}$ 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 sustituida por el término $\hat{!}$ dentro de la definición.

$$\hat{!} \equiv \lambda f n. (\hat{!} (\hat{-}_1 n)) \quad (3.48)$$

El problema con (3.48) es que si se reduce $(\hat{!}\hat{!}\hat{n})$ y $n \neq 0$, ocurre lo siguiente:

$$\hat{!}\hat{!}\hat{n} \rightarrow_{\beta} (\hat{\times} \hat{n} (\hat{!}(\hat{-}_1 \hat{n})))$$

La multiplicación de \hat{n} debe realizarse con otro numeral, sin embargo, $\hat{!}$ espera ser aplicado a $\hat{!}$ y a $(\hat{-}_1 \hat{n})$, sin embargo, en la reducción el término $\hat{!}$ no es aplicado a sí mismo.

$$(\hat{\times} \hat{n} (\hat{!} \underbrace{(\hat{-}_1 \hat{n})}_{\text{Debe de ir } \hat{!}}))$$

Para escapar de este problema, se debe de aplicar $(f f(\hat{-}_1 \hat{n}))$ en el cuerpo de la definición. El factorial modificado es

$$\hat{!} \equiv \lambda f n. (\hat{!} (\hat{-}_1 n)) \quad (3.49)$$

Esta definición se escapa del problema de la definición recursiva, puede reducirse a la codificación de la función factorial aplicando

$$\hat{!}\hat{!} \rightarrow_{\beta} \lambda n. (\hat{!} (\hat{-}_1 n) \hat{!} (\hat{\times} \hat{n} (\hat{!}\hat{!}(\hat{-}_1 n))))$$

En efecto, no se necesita el nombrar la abstracción con un símbolo como $\hat{!}$ para expresar la codificación del factorial. La reducción anterior es exactamente la misma a

En el ejemplo 1.2.3 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 $(\omega \omega) \rightarrow_{\beta} (\omega \omega)$, que es justo lo que deseamos en nuestro mecanismo recursivo, el combinador que se necesita es uno similar a ω , llamado ω' tal que $(\omega' \omega') \rightarrow_{\beta} (\widehat{!}(\omega' \omega'))$.

Suponiendo que $\widehat{!}$ ya es una variable enlazada, el combinador ω' debe ser una abstracción que espera ser aplicada a sí 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 Y .

$$Y \equiv \lambda f.(\lambda x.f(x x))(\lambda x.f(x x)) \quad (3.50)$$

Al aplicar el combinador Y a la definición (3.48) del factorial, se obtiene que

$$\begin{aligned} Y\widehat{!} &=_{\beta} \widehat{!}(Y\widehat{!}) \\ &\equiv (\lambda f n.(\prec (\widehat{0} n) \widehat{1} (\widehat{\times} n (f(\widehat{-}_1 n)))))(Y\widehat{!}) \\ &\rightarrow_{\beta} \lambda n.(\prec (\widehat{0} n) \widehat{1} (\widehat{\times} n (Y\widehat{!}(\widehat{-}_1 n)))) \end{aligned}$$

Al reducir la aplicación de este término en un numeral mayor a cero se obtiene

$$\begin{aligned} (\lambda n.(\prec (\widehat{0} n) \widehat{1} (\widehat{\times} n (Y\widehat{!}(\widehat{-}_1 n)))) \widehat{n} &\rightarrow_{\beta} (\widehat{\times} \widehat{n} (Y\widehat{!}(\widehat{-}_1 \widehat{n}))) \\ (\widehat{\times} n (Y\widehat{!}(\widehat{-}_1 n)))) \widehat{n} \end{aligned}$$

Es entonces que $(Y\widehat{!})$ puede ser β -convertido nuevamente a $(\widehat{!}(Y\widehat{!}))$ 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 λ .

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

$$X \equiv (\lambda x.F(x x))W \rightarrow_{\beta} F(W W) \equiv F X$$

□

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 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 [14, p. 34], es decir, si M es un combinador de punto fijo, entonces, para toda F

$$F(M F) =_{\beta} M F \quad (3.51)$$

A partir de la ecuación (3.50), se puede corroborar que Y es un combinador de punto fijo de la siguiente manera

$$\begin{aligned} Y F &\equiv (\lambda f.(\lambda x.f(x x))(\lambda x.f(x x)))F \\ &\rightarrow_{\beta} (\lambda x.F(x x))(\lambda x.F(x x)) \\ &\rightarrow_{\beta} F((\lambda x.F(x x))(\lambda x.F(x x))) \\ &=_{\beta} F(Y F) \end{aligned} \quad (3.52)$$

Existen combinadores con una propiedad más fuerte que (3.51). Alan Turing descubrió el combinador Θ en 1937, su definición es

$$\Theta \equiv U U \quad \text{donde} \quad U \equiv \lambda u x.x(u u x) \quad (3.53)$$

La propiedad interesante de Θ es que puede computar puntos fijos únicamente con la β -reducción. Se corrobora esto de la siguiente manera

$$\begin{aligned}
\Theta F &\equiv U U F \\
&\rightarrow_{\beta} (\lambda x. x(U U x))F \\
&\rightarrow_{\beta} F(U U F) \\
&\equiv F(\Theta F)
\end{aligned} \tag{3.54}$$

Esta propiedad no la tiene 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 λ , con variables libres f y \vec{x} , encuentra el término F tal que

$$F \vec{M} =_{\beta} Z [f := F] [\vec{x} := \vec{M}]$$

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 $Z \equiv (\prec (\hat{o}_? n) \hat{1} (\hat{\times} n (f (\hat{\prec}_1 n))))$, encuentra el término $\hat{!}$ tal que

$$\hat{!} \hat{n} =_{\beta} Z [f := \hat{!}] [n := \hat{n}]$$

Solución. Por la regla (β) se tiene que

$$\hat{!} \hat{n} =_{\beta} (\lambda f n. \prec (\hat{o}_? n) \hat{1} (\hat{\times} n (\hat{!} (\hat{\prec}_1 n)))) \hat{!} \hat{n}$$

Por la regla (ν) se tiene que

$$\hat{!} =_{\beta} (\lambda f n. \prec (\hat{o}_? n) \hat{1} (\hat{\times} n (\hat{!} (\hat{\prec}_1 n)))) \hat{!}$$

Esta ecuación tiene la forma $A =_{\beta} (B A)$, al considerar a A de la forma $(Y B)$, se cumple la propiedad de (3.51) y por lo tanto

$$\hat{!} \equiv Y (\lambda f n. \prec (\hat{o}_? n) \hat{1} (\hat{\times} n (f (\hat{\prec}_1 n))))$$

□

En general, la solución para este tipo de problemas es

$$F \equiv Y (\lambda f \vec{x}. Z) \tag{3.55}$$

3.4 Estructuras recursivas

Utilizando combinadores de punto fijo como Y y Θ 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 λ .

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 “construir 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 λ , se deben plantear los mecanismos para construir 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)).

$$p \equiv \lambda a d. \lambda q. q a d \quad (3.56)$$

$$a \equiv \lambda c. c(\lambda a d. a) \quad (3.57)$$

$$d \equiv \lambda c. c(\lambda a d. d) \quad (3.58)$$

Estas tres ecuaciones cumplen con las reducciones

$$(a(p M N)) \rightarrow_{\beta} M$$

$$(d(p M N)) \rightarrow_{\beta} N$$

para cualesquiera términos λM y N . Para corroborar esto, se desarrollan las reducciones:

$$\begin{aligned} a(p M N) &\equiv (\lambda c. c(\lambda a d. a))(p M N) & (3.59) \\ &\rightarrow_{\beta} (p M N)(\lambda a d. a) \\ &\equiv ((\lambda a d. \lambda q. q a d) M N)(\lambda a d. a) \\ &\rightarrow_{\beta} (\lambda q. q M N)(\lambda a d. a) \\ &\rightarrow_{\beta} (\lambda a d. a) M N \\ &\rightarrow_{\beta} M \end{aligned}$$

$$\begin{aligned}
d(p\ M\ N) &\equiv (\lambda c.c(\lambda a\ d.d))(p\ M\ N) \\
&\rightarrow_{\beta} (p\ M\ N)(\lambda a\ d.d) \\
&\equiv ((\lambda a\ d.\lambda q.q\ a\ d)\ M\ N)(\lambda a\ d.d) \\
&\rightarrow_{\beta} (\lambda q.q\ M\ N)(\lambda a\ d.d) \\
&\rightarrow_{\beta} (\lambda a\ d.d)\ M\ N \\
&\rightarrow_{\beta} N
\end{aligned} \tag{3.60}$$

En el resto de esta sección se abordan diferentes maneras en las que se puede emplear la estructura del par para construir 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 λ 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 .

$$\langle \widehat{n}_1 : \langle \widehat{n}_2 : \langle \widehat{n}_3 : \langle \dots \langle \widehat{n}_k : \emptyset \rangle \dots \rangle \rangle \rangle \rangle$$

Denotada de manera abreviada como

$$\langle \widehat{n}_1, \widehat{n}_2, \widehat{n}_3, \dots, \widehat{n}_k \rangle$$

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 $\widehat{o}?$ fue construido 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).

$$\mathcal{L} \longrightarrow \langle \widehat{n} : \mathcal{L} \rangle \mid \emptyset \tag{3.61}$$

Por lo tanto, se espera que el predicado $\emptyset?$ sea aplicado a un término de la forma $(\lambda q.q\ \widehat{n}\ \mathcal{L})$ o al término \emptyset , de tal manera que

$$\begin{aligned}\emptyset_?(\lambda q.q \widehat{n} \mathcal{L}) &\rightarrow_\beta F \\ \emptyset_? \emptyset &\rightarrow_\beta T\end{aligned}$$

Una manera de convertir un término par a F es reduciendo la aplicación $(KF(\lambda q.q \widehat{n} \mathcal{L}))$, sin embargo, este resultado es el mismo para cualquier valor al que se aplique (KF) y el objetivo es poder discriminar entre un par y \emptyset . Se puede considerar un término similar a (KF) pero que cancele los siguientes dos términos a los que sea aplicado:

$$(\lambda x y.F)MN \rightarrow_\beta F$$

De esta manera, aplicar un par a este término resulta en la reducción

$$\begin{aligned}(\lambda q.q \widehat{n} \mathcal{L})(\lambda x y.F) &\rightarrow_\beta (\lambda x y.F)\widehat{n} \mathcal{L} \\ &\rightarrow_\beta F\end{aligned}$$

Por lo tanto, la codificación de $\emptyset_?$ que es reducida a falso al ser aplicada a un par es

$$\emptyset_? \equiv (\lambda l.l(\lambda x y.F)) \tag{3.62}$$

Con esta definición, la propiedad que debe cumplir la codificación de \emptyset es $(\emptyset_? \emptyset) \rightarrow_\beta T$, lo cuál resulta ser el término (KT) ya que

$$\begin{aligned}\emptyset_?(KT) &\equiv (\lambda l.l(\lambda x y.F))(KT) \\ &\rightarrow_\beta KT(\lambda x y.F) \\ &\rightarrow_\beta T\end{aligned}$$

Por lo tanto

$$\emptyset \equiv KT \tag{3.63}$$

Observación (Sobre predicados). La aplicación de un predicado como $\widehat{o}_?$ o $\emptyset_?$ a un término λM 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 construir 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 \mathcal{L} no sea el valor nulo, en cuyo caso, será un par cuyo primer elemento es n y segundo elemento es otra lista \mathcal{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 \mathcal{L}' .

Algoritmo 3.11 Procedimiento recursivo dobles(\mathcal{L})

Entrada: $\mathcal{L} \in \{\langle n_1, \dots, n_k \rangle \mid n \in \mathbb{N}\} \cup \emptyset$

Salida: $\langle 2 \times n_1, \dots, 2 \times n_k \rangle$ ó \emptyset

si $\mathcal{L} = \emptyset$ **entonces**
 regresar \emptyset
de lo contrario
 $\langle n : \mathcal{L}' \rangle \leftarrow \mathcal{L}$
 regresar $\langle 2 \times n : \text{dobles}(\mathcal{L}') \rangle$
fin si

El predicado $\emptyset?$ es utilizado con la condicional booleana \prec sobre la lista \mathcal{L} . Cuando la lista sea la codificación de nulo, el resultado es nulo; de lo contrario, se asume que \mathcal{L} es un par y se construye el par correspondiente utilizando los términos p , a y d . La codificación resultante es (3.64).

$$\begin{aligned} Y(\lambda f \mathcal{L}. \prec (\emptyset? \mathcal{L}) & \quad (3.64) \\ \emptyset & \\ (p (\widehat{\times} \widehat{2} (a \mathcal{L})) & \\ (f (d \mathcal{L})))) & \end{aligned}$$

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.

$$\begin{aligned} Y(\lambda f g \mathcal{L}. \prec (\emptyset? \mathcal{L}) & \quad (3.65) \\ \emptyset & \\ (p (g(a \mathcal{L})) & \\ (f g(d \mathcal{L})))) & \end{aligned}$$

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.

$$\begin{aligned}
Y(\lambda f p \emptyset g \mathcal{L}. < (\emptyset, \mathcal{L}) & \quad (3.66) \\
\emptyset & \\
(p (g(a \mathcal{L})) & \\
(f p \emptyset g(d \mathcal{L}))) &
\end{aligned}$$

Si \mathcal{F} es la abstracción (3.66), \mathcal{M} es la abstracción (3.65), \mathcal{D} es la abstracción (3.64) y \mathcal{L} es el término λ que codifica la lista $\langle n_1, \dots, n_k \rangle$, entonces

$$\mathcal{F} p \emptyset (\widehat{\times} \widehat{2}) \mathcal{L} =_{\beta} \mathcal{M} (\widehat{\times} \widehat{2}) \mathcal{L} =_{\beta} \mathcal{D} \mathcal{L}$$

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 \mathcal{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(\mathcal{L})

Entrada: $\mathcal{L} \in \{\langle n_1, \dots, n_k \rangle \mid n \in \mathbb{N}\} \cup \emptyset$

Salida: $\langle n'_1, \dots, n'_{k'} \rangle$ ó $\emptyset \mid n'_i \in \mathcal{L}, n'_i$ impar

si $\mathcal{L} = \emptyset$ entonces

regresar \emptyset

de lo contrario

$\langle n : \mathcal{L}' \rangle \leftarrow \mathcal{L}$

si n impar entonces

regresar $\langle n : \text{impares}(\mathcal{L}') \rangle$

de lo contrario

regresar $\text{impares}(\mathcal{L}')$

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 $\widehat{\text{imp}}_?$ la codificación del predicado impar, se deben de satisfacer las siguientes reducciones:

$$\begin{aligned}
\widehat{imp}_?(\lambda x y. y) &\rightarrow_\beta F \\
\widehat{imp}_?(\lambda x y. x y) &\rightarrow_\beta T \\
\widehat{imp}_?(\lambda x y. x(x y)) &\rightarrow_\beta F \\
\widehat{imp}_?(\lambda x y. x(x(x y))) &\rightarrow_\beta T \\
&\dots
\end{aligned}$$

Una manera sencilla de codificar este término es

$$\widehat{imp}_? \equiv \lambda n. n \neg F$$

De esta manera

$$\begin{aligned}
\widehat{imp}_?(\lambda x y. y) &\rightarrow_\beta F \\
\widehat{imp}_?(\lambda x y. x y) &\rightarrow_\beta \neg F \\
\widehat{imp}_?(\lambda x y. x(x y)) &\rightarrow_\beta \neg(\neg F) \\
\widehat{imp}_?(\lambda x y. x(x(x y))) &\rightarrow_\beta \neg(\neg(\neg F)) \\
&\dots
\end{aligned}$$

Con este nuevo predicado, la codificación del algoritmo 3.12 es (3.67).

$$\begin{aligned}
Y(\lambda f \mathcal{L}. &\prec (\emptyset? \mathcal{L}) \\
&\emptyset \\
&(\prec (\widehat{imp}_?(a \mathcal{L})) \\
&\quad (p(a \mathcal{L})(f(d \mathcal{L}))) \\
&\quad (f(d \mathcal{L}))))
\end{aligned} \tag{3.67}$$

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.

$$\begin{aligned}
Y(\lambda f g? \mathcal{L}. < (\emptyset? \mathcal{L}) \\
& \emptyset \\
& (< (g?(a \mathcal{L})) \\
& \quad (p(a \mathcal{L})(f g?(d \mathcal{L}))) \\
& \quad (f g?(d \mathcal{L}))))
\end{aligned} \tag{3.68}$$

Si \mathcal{F} es la abstracción (3.68), \mathcal{I} es la abstracción (3.67) y \mathcal{L} es el término λ que codifica la lista $\langle n_1, \dots, n_k \rangle$, entonces

$$\widehat{\mathcal{F} \text{ imp}_? \mathcal{L}} =_{\beta} \mathcal{I} \mathcal{L}$$

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 \mathcal{L} puede tener términos λ que no sean numerales, al utilizar los algoritmos genéricos sólo se debe tener cuidado con que la forma de los términos en \mathcal{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 construir árboles. Consideremos el árbol mostrado en la figura 3.3.

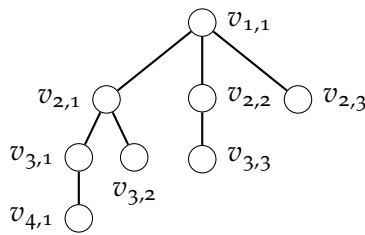


Figura 3.3: Ejemplo de árbol

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 \mathcal{T} puede ser o un vértice v acompañado de una lista de árboles $\mathcal{L}_{\mathcal{T}}$ o un valor nulo \emptyset , es decir

$$\mathcal{T} \longrightarrow (v, \langle \mathcal{T}_1, \dots, \mathcal{T}_k \rangle) \mid \emptyset$$

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 puede ser representado por cualquier término λ , en ocasiones es útil asociar un término a cada vértice del árbol para construir, por ejemplo, árboles de numerales.

El árbol de la figura 3.3 se representa de la siguiente manera:

$$\begin{aligned} &\langle v_{1,1} : \langle \langle v_{2,1} : \langle \langle v_{3,1} : \langle \langle v_{4,1} : \emptyset \rangle \rangle \rangle \rangle \rangle \\ &\quad \langle v_{3,2} : \emptyset \rangle \rangle \rangle \\ &\quad \langle v_{2,2} : \langle \langle v_{3,3} : \emptyset \rangle \rangle \rangle \\ &\quad \langle v_{2,3} : \emptyset \rangle \rangle \rangle \end{aligned}$$

La agrupación entre un vértice y sus subárboles se puede realizar con la estructura par, de tal manera que

$$\mathcal{T} \equiv \langle v : \langle \mathcal{T}_1, \dots, \mathcal{T}_k \rangle \rangle =_{\beta} \langle v, \mathcal{T}_1, \dots, \mathcal{T}_k \rangle$$

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: $\mathcal{T} \in \{\langle v : \langle \mathcal{T}_1, \dots, \mathcal{T}_k \rangle \rangle\} \cup \emptyset$

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 λ como

$$\begin{aligned}
& Y(\lambda f \mathcal{T}. \prec(\emptyset, \mathcal{T}) \\
& \quad \widehat{0} \\
& \quad (\widehat{+}_1(\mathcal{F} \widehat{+} f(d \mathcal{T}))))
\end{aligned}$$

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.

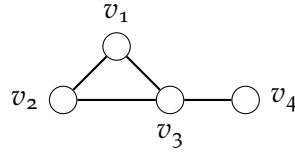


Figura 3.4: Ejemplo de gráfica

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:

$$\begin{aligned}
& \langle \langle v_1 : \langle v_2, v_3 \rangle \rangle \\
& \quad \langle v_2 : \langle v_1, v_3 \rangle \rangle \\
& \quad \langle v_3 : \langle v_1, v_2, v_4 \rangle \rangle \\
& \quad \langle v_4 : \langle v_3 \rangle \rangle \rangle
\end{aligned}$$

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 λ conocidos, los algoritmos que manipulan gráficas pueden ser codificados de manera similar a los presentados en las secciones 3.4.2 y 3.4.1.

3.4.4 Términos λ

En esta sección se aborda el mecanismo mediante el cual es posible codificar términos λ dentro del cálculo λ y de esta manera, tener las herramientas para codificar algoritmos que manipulen

términos λ como la sustitución, la α -contracción, la β -contracción, encontrar los subtérminos de un término, etc.

El hecho de poder codificar términos λ en el cálculo λ no introduce problema alguno. Esta 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 λ , se modifica el conjunto V para que en lugar de ser $\{v_0, v_{00}, \dots\}$, sea $\{0, 1, \dots\}$, es decir, que los átomos sean números naturales; además en lugar de construir las abstracciones y aplicaciones a partir de símbolos, se construyen a partir de pares.

Definición 3.4.1 (Términos λ). El conjunto Λ' tiene elementos que son pares y números naturales. Λ' es el conjunto más pequeño que satisface:

$$n \in \mathbb{N} \implies n \in \Lambda' \quad (a)$$

$$M \in \Lambda', n \in \mathbb{N} \implies \langle n : M \rangle \in \Lambda' \quad (b)$$

$$M, N \in \Lambda' \implies \langle M : N \rangle \in \Lambda' \quad (c)$$

El problema con esta definición de términos λ es que no es posible distinguir al término $(\lambda x.x)$ de $(x x)$. Además, la mayoría de los algoritmos que manipulan términos λ 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 λ). El conjunto Λ' tiene elementos que son pares. Λ' es el conjunto más pequeño que satisface:

$$n \in \mathbb{N} \implies \langle 1 : n \rangle \in \Lambda' \quad (a)$$

$$M \in \Lambda', n \in \mathbb{N} \implies \langle 2 : \langle n : M \rangle \rangle \in \Lambda' \quad (b)$$

$$M, N \in \Lambda' \implies \langle 3 : \langle M : N \rangle \rangle \in \Lambda' \quad (c)$$

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 λ se basan en las técnicas abordadas a lo largo de este capítulo.

Sea \mathcal{T} la codificación de un término λ , los predicados *atomo?*, *abstraccion?* y *aplicacion?* permiten determinar la clase de término que es \mathcal{T} :

$$\text{atomo}_? \equiv \lambda \mathcal{T}. \widehat{=}_? \widehat{1}(a \mathcal{T}) \quad (3.71)$$

$$\text{abstraccion}_? \equiv \lambda \mathcal{T}. \widehat{=}_? \widehat{2}(a \mathcal{T}) \quad (3.72)$$

$$\text{aplicacion}_? \equiv \lambda \mathcal{T}. \widehat{=}_? \widehat{3}(a \mathcal{T}) \quad (3.73)$$

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:

$$\text{ramifica} \equiv \lambda \mathcal{T} x y z e. \prec (\text{atomo}_? \mathcal{T}) x (\prec (\text{abstraccion}_? \mathcal{T}) y (\prec (\text{aplicacion}_? \mathcal{T}) z e)) \quad (3.74)$$

Los constructores para cada clase de término λ siguen la definición 3.4.2:

$$\text{atomo} \equiv \lambda n. p \widehat{1} n \quad (3.75)$$

$$\text{abstraccion} \equiv \lambda n x. p \widehat{2}(p n x) \quad (3.76)$$

$$\text{aplicacion} \equiv \lambda x y. p \widehat{3}(p x y) \quad (3.77)$$

Los constructores en (3.75) deben satisfacer los predicados en 3.71, se corrobora esto con los siguientes desarrollos.

$$\begin{aligned} \text{atomo}_?(\text{atomo } \widehat{n}) &\equiv (\lambda \mathcal{T}. \widehat{=}_? \widehat{1}(a \mathcal{T}))(\text{atomo } \widehat{n}) \\ &\rightarrow_\beta \widehat{=}_? \widehat{1}(a(\text{atomo } \widehat{n})) \\ &\equiv \widehat{=}_? \widehat{1}(a((\lambda n. p \widehat{1} n)\widehat{n})) \\ &\rightarrow_\beta \widehat{=}_? \widehat{1}(a(p \widehat{1} \widehat{n})) \\ &\rightarrow_\beta \widehat{=}_? \widehat{1} \widehat{1} \\ &\rightarrow_\beta T \end{aligned}$$

$$\begin{aligned} \text{atomo}_?(\text{aplicacion } M N) &\equiv (\lambda \mathcal{T}. \widehat{=}_? \widehat{1}(a \mathcal{T}))(\text{aplicacion } M N) \\ &\rightarrow_\beta \widehat{=}_? \widehat{1}(a(\text{aplicacion } M N)) \\ &\equiv \widehat{=}_? \widehat{1}(a((\lambda x y. p \widehat{3}(p x y))M N)) \\ &\rightarrow_\beta \widehat{=}_? \widehat{1}(a(p \widehat{3}(p M N))) \\ &\rightarrow_\beta \widehat{=}_? \widehat{1} \widehat{3} \\ &\rightarrow_\beta F \end{aligned}$$

Corroborar que las otras combinaciones de aplicaciones se reducen de manera correcta implica un desarrollo similar.

Los selectores para cada clase de término λ 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.

$$\text{atomo}_n \equiv \lambda x. d \ x \quad (3.78)$$

$$\text{abstraccion}_v \equiv \lambda x. a(d \ x) \quad (3.79)$$

$$\text{abstraccion}_t \equiv \lambda x. d(d \ x) \quad (3.80)$$

$$\text{aplicacion}_o \equiv \lambda x. a(d \ x) \quad (3.81)$$

$$\text{aplicacion}_a \equiv \lambda x. d(d \ x) \quad (3.82)$$

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 λ puede ser más robusta, evitando errores que puedan surgir al reducir los términos.

Conclusión

En este trabajo se presentaron las ideas generales del cálculo λ , se formalizaron utilizando como herramientas matemáticas las teorías formales y los sistemas de reducción, también se desarrollaron varias maneras para representar información compleja y algoritmos tanto iterativos como recursivos utilizando únicamente funciones y aplicación de funciones. Sin embargo, este trabajo comprende sólo una pequeña parte del estudio del cálculo λ .

Este trabajo puede ser ameno de leer para algunas personas y para otras no. Intenté mantener un balance entre los aspectos conceptuales y formales, pero el contenido del trabajo terminó siendo más técnico de lo que esperaba. Los aspectos pragmáticos y las aplicaciones del cálculo λ están ausentes del trabajo y esto puede resultar contraproducente para interesar a un lector que nunca ha sido expuesto al cálculo λ .

Algunos aspectos que se pueden estudiar después de leer este trabajo son las extensiones del cálculo con lógica ilativa, teoría de tipos, semántica denotacional y programación funcional.

Apéndice A

Programación de codificaciones

Eduardo Acuña A.1

La versión final de este apéndice contiene una descripción de la implementación y el resto de las codificaciones (como se puede observar la traducción es bastante directa).

El siguiente código es una implementación literal de los combinadores *S*, *K* e *I* descritos en 2.2.6; 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 ( $\lambda$  (x) x))  
(define K ( $\lambda$  (x) ( $\lambda$  (y) x)))  
(define S ( $\lambda$  (x) ( $\lambda$  (y) ( $\lambda$  (z) ((x z) (y z))))))  
  
;;  
;; Álgebra booleana  
;;  
(define T K)  
(define F ( $\lambda$  (x) ( $\lambda$  (y) y)))  
(define IF ( $\lambda$  (p) ( $\lambda$  (m) ( $\lambda$  (n) ((p m) n)))))  
(define NOT ( $\lambda$  (p) (((IF p) F) T)))  
(define OR ( $\lambda$  (p1) ( $\lambda$  (p2) (((IF p1) T) (((IF p2) T) F)))))  
(define AND ( $\lambda$  (p1) ( $\lambda$  (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 λ como λ y la secuencia \rightarrow 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 =  $\lambda x\ y\ z \rightarrow n\ x\ (z\ y)\ x$ 
N_Pred n =  $\lambda x\ y \rightarrow n\ N\_Succ2\ N\_02\ x\ y\ C\_I$ 
N_Sub m n = n N_Pred m

```


Apéndice B

Intérprete del cálculo λ

Eduardo Acuña B.1

La versión final de este apéndice es un programa literario en español, está pendiente actualizar la implementación. En el repositorio de Github están los avances del programa literario <https://github.com/eduardoacye/tesis/blob/master/racket/docs/lambda.pdf>

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- $\lambda$  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: $\lambda$  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* '(#\{ #\[ #\ ( #\  $\lambda$  #\ \ ))

(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))])])
          [else
           (loop (cons arg lis) (read-expr in))])
    (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- $\lambda$  [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 #\lambda))))
    'read- $\lambda$ 
    "expecting ' $\lambda$ ' 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: $\lambda$  (make-parameter " $\lambda$ "))
(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~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~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)

```

```

=> ( $\lambda$  (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  $\lambda$  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  $\lambda$  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~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~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 T_EX

```

#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: $\lambda$  " \\lambda ")
(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)
    sub))

```

```

      (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)]
              [else #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 "tex $\lambda$ " #:args (entry) entry))  
  
(tex $\lambda$  console-args)
```

Apéndice C

Intérprete Lambda

Este apéndice consiste de un programa literario¹ conformado por un intérprete y un editor estructural para el cálculo λ . Se utilizó la herramienta Noweb, escrita por Norman Ramsey², para extraer del documento el código en Racket del programa ejecutable y el código en \LaTeX para la composición tipográfica del presente apéndice.

La versión de Racket para la que este programa fue programado y probado es la 6.6. La versión del programa Noweb es la 2.11 adaptada al Español.

Eduardo Acuña C.1

Este tercer apéndice es la versión actualizada del apéndice B, cuando el contenido de este apéndice haya sido redactado por completo, aquel apéndice será eliminado. Los cambios son principalmente un código más limpio, compartimentado y en español. Otra adición al código es el editor estructural el cual complementa al REPL. El código ejecutable se encuentra en <https://github.com/eduardoacye/tesis/blob/master/racket/progs/editor2.rkt> pero aún no ha sido escrito por completo como un programa literario.

¹La programación literaria es un paradigma propuesto por Donald E. Knuth en donde los programas son escritos en un orden que favorece la manera de pensar del programador más que el orden de ejecución del código por la computadora.

²Para más información, visitar el sitio <http://www.cs.tufts.edu/~nr/noweb/>

LAMBDA

Un programa para la exploración del cálculo λ puro

Introducción El programa `lambda`³ tiene la finalidad de proveer un entorno computacional ameno para estudiar el contenido de este trabajo de manera interactiva. La presente implementación está escrita en el lenguaje de programación Racket⁴ y consta de un intérprete de expresiones del cálculo λ el cuál puede ser utilizado de tres maneras diferentes:

1. A través de un ciclo de lectura, evaluación y escritura (conocido como *REPL*) por consola;
2. Por medio de un editor estructural gráfico;
3. Como una biblioteca para facilitar la programación de algoritmos que involucren términos del cálculo λ .

El corazón del intérprete es un *evaluador*, una función que transforma expresiones de un lenguaje a valores, los cuales a su vez pueden ser expresiones de este lenguaje. El lenguaje con el que trabaja este procedimiento es una extensión a Λ que incluye “huecos” y “metainstrucciones”, estas últimas permiten manipular términos λ de manera arbitraria, por ejemplo calcular la longitud de un término M involucraría invocar una metainstrucción similar a la definición de $\|M\|$. Las expresiones de este lenguaje son almacenadas en memoria como un árbol de sintaxis. Para construir estos árboles se utilizan dos métodos: usando un *lector* (o parser) que transforma cadenas de caracteres a árboles o utilizando el *editor estructural* que permite únicamente editar expresiones válidas del lenguaje.

Estructuras del lenguaje

Árbol de sintaxis Un árbol de sintaxis para el lenguaje de `lambda` consiste de vértices que pueden ser *variables*, *aplicaciones*, *abstracciones*, *huecos* o *metainstrucciones*. Estos tipos de vértices son implementados haciendo uso del macro `struct` el cuál a partir de un identificador de la estructura y los identificadores de sus componentes produce un constructor, un predicado y selectores para cada componente.

La estructura que representa a las variables de un árbol es llamada *variable* y se compone de una cadena de caracteres llamada *nombre*.

³Nombrado así en ausencia de originalidad.

⁴Para más información, visitar el sitio oficial <http://racket-lang.org/>.

- 1 $\langle \text{estructuras del lenguaje } 1 \rangle \equiv$
 (struct variable (nombre) #:transparent)

Define:

variable, usado en fragmentos 1, 4, 7, 1, 8, 9, 26, 29, 33, 1, 34-36, 40, 1, 41, y 42.

Esta definición se continua en fragmentos 2, 3, 5, y 6.

Este código es usado en fragmento 10.

La estructura que representa a las abstracciones de un árbol es llamada abstracción, se compone de una variable llamada argumento y de otro vértice del árbol llamado cuerpo, este último puede ser de cualquiera de los cuatro tipos de vértices mencionados.

- 2 $\langle \text{estructuras del lenguaje } 1 \rangle + \equiv$
 (struct abstracción (argumento cuerpo) #:transparent)

Define:

abstracción, usado en fragmentos 2, 4, 7, 2, 8, 9, 22, 2, 25, 2, 26, 33, 36, 40, 42, y 43.

La estructura que representa a las aplicaciones de un árbol es llamada aplicación, se compone de un vértice llamado operador y de otro vértice llamado operando.

- 3 $\langle \text{estructuras del lenguaje } 1 \rangle + \equiv$
 (struct aplicación (operador operando) #:transparent)

Define:

aplicación, usado en fragmentos 3, 4, 7, 3, 8, 9, 20, 3, 21-23, 25, 3, 30, 33, 37, 40, 3, y 43.

Se define el predicado término? para determinar si un objeto dado es un término λ , es decir, si es una estructura variable, abstracción o aplicación. Se debe verificar que los campos de cada subtérmino corresponden a un término λ adecuado, por ejemplo, las abstracciones deben tener como argumento una variable y como cuerpo un término.

- 4 $\langle \text{predicados del lenguaje } 4 \rangle \equiv$
 (define (término? x)
 (cond
 [(variable? x) (string? (variable-nombre x))]
 [(abstracción? x) (and (variable? (abstracción-argumento x))
 (término? (abstracción-cuerpo x)))]
 [(aplicación? x) (and (término? (aplicación-operador x))
 (término? (aplicación-operando x)))]
 [else #f]))

Define:

término?, usado en fragmentos 4 y 9.

Hace referencia a abstracción 2, aplicación 3, y variable 1.

Esta definición se continua en fragmento 7.

Este código es usado en fragmento 10.

La estructura que representa a las metainstrucciones de un árbol es llamada metainstrucción, se compone de una cadena de caracteres llamada nombre y de una lista de vértices llamada argumentos.

5 $\langle \text{estructuras del lenguaje } 1 \rangle + \equiv$
 (struct metainstrucción (nombre argumentos) #:transparent)

Define:

metainstrucción, usado en fragmentos 5, 7, 9, 11, 29, 33, 39, 40, y 45.

Para representar un vértice sin un término asignado se utiliza la estructura hueco, esta funciona como una especie de comodín o valor nulo.

6 $\langle \text{estructuras del lenguaje } 1 \rangle + \equiv$
 (struct hueco () #:transparent)

Define:

hueco, usado en fragmentos 6, 7, 9, 23, 24, 33, 40, y 44.

De manera análoga a la identificación de términos λ mediante el procedimiento término? se define el predicado expresión? el cuál permite distinguir estructuras que sean expresiones de este lenguaje, incluyendo a los términos λ .

7 $\langle \text{predicados del lenguaje } 4 \rangle + \equiv$
 (define (expresión? x)
 (cond
 [(variable? x) (string? (variable-nombre x))]
 [(abstracción? x) (and (variable? (abstracción-argumento x))
 (expresión? (abstracción-cuerpo x)))]
 [(aplicación? x) (and (expresión? (aplicación-operador x))
 (expresión? (aplicación-operando x)))]
 [(metainstrucción? x) (and (string? (metainstrucción-nombre x))
 (every expresión? (metainstrucción-argumentos x)))]
 [(hueco? x) #t]
 [else #f]))

Define:

expresión?, usado en fragmentos 7, 9, y 26.

Hace referencia a abstracción 2, aplicación 3, hueco 6, metainstrucción 5, y variable 1.

Utilizando los constructores variable, abstracción y aplicación, un término λ como

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

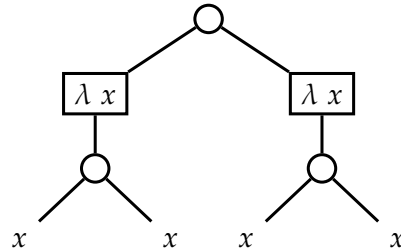
puede ser representado en memoria como

- 8 *<construcción de un término 8>*≡
 (aplicación (abstracción (variable "x") (aplicación (variable "x") (variable "x")))
 (abstracción (variable "x") (aplicación (variable "x") (variable "x"))))

Hace referencia a abstracción 2, aplicación 3, y variable 1.

Fragmento raíz (no usado en este documento).

el cuál corresponde a un árbol de sintaxis visualizado graficamente como



Módulo de estructuras del lenguaje Las estructuras y procedimientos del árbol de sintaxis son agrupados en un módulo de Racket el cuál podrá ser accedido desde otros módulos. Para “exportar” los identificadores definidos se utiliza el macro provide.

- 9 *<provee las definiciones del árbol 9>*≡
 (provide variable variable? variable-nombre
 abstracción abstracción? abstracción-argumento abstracción-cuerpo
 aplicación aplicación? aplicación-operador aplicación-operando
 metainstrucción metainstrucción? metainstrucción-nombre metainstrucción-argumentos
 hueco hueco?
 término? expresión?)

Hace referencia a abstracción 2, aplicación 3, expresión? 7, hueco 6, metainstrucción 5, término? 4, y variable 1.

Este código es usado en fragmento 10.

El archivo estructuras.rkt contiene este módulo y tiene la siguiente forma:

- 10 *<estructuras.rkt 10>*≡
 ;;; -*- mode: racket; coding: utf-8 -*-
 ;;; Copyright (C) 2016 Eduardo Acuña Yeomans <eduardo.acye@gmail.com>
 ;;; Procura no editar este archivo, mejor modifica lambda.nw

 #lang racket/base

 (require srfi/1)

<provee las definiciones del árbol 9>

 ;;; Estructuras del árbol de sintaxis
<estructuras del lenguaje 1>

 ;;; Predicados especiales

⟨predicados del lenguaje 4⟩

Fragmento raíz (no usado en este documento).

Lectura de expresiones

Representación textual El lenguaje de lambda es descrito conceptualmente por las estructuras de los árboles de sintaxis. Sin embargo es tedioso definir expresiones del lenguaje escribiendo manualmente los constructores de las expresiones. Trabajando en un contexto meramente textual como en el código de un lenguaje de programación es conveniente tener una manera de transformar una cadena de caracteres como $((\lambda x.(xx))(\lambda x.(xx)))$ (o mejor aún, con abuso de notación $(\lambda x.xx)\lambda x.xx$) a un árbol de sintaxis de este término. Este es el trabajo del lector.

La representación textual del lenguaje contempla la escritura de términos descrita en la definición formal de Λ y los abusos de notación contemplados en el trabajo. Adicionalmente se representan las metainstrucciones de tal manera que

11 *⟨pseudocódigo de una metainstrucción 11⟩*≡
 (metainstrucción nombre (list argumento-0 argumento-1 ... argumento-n))

Hace referencia a metainstrucción 5.

Fragmento raíz (no usado en este documento).

tiene como representación textual

‘nombre[argumento-0, argumento-1, ... , argumento-n]’

Parser El lector de expresiones de este lenguaje es un *parser* descendente recursivo que a partir de un puerto de entrada de Racket construye un árbol de sintaxis.

El procedimiento leer es el punto de entrada del parser, recibe opcionalmente un puerto llamado p que por defecto es (current-input-port).

12 *⟨lector de expresiones 12⟩*≡
 (define (leer [p (current-input-port)])
 (let ([e (parsear-expresión p)])
 (cond
 [(and (char? e) (char=? #\; e))
 (leer p)]
 [(not (char? e))
 e]
 [else
 (error 'leer "Se esperaba una expresión completa pero se leyó '~a' " e)])))

Define:

leer, usado en fragmentos 12, 31, 12, y 32.

Hace referencia a parsear-expresión 13.

Esta definición se continua en fragmento 13.

Este código es usado en fragmento 32.

El procedimiento `parsear-expresión` se encarga de regresar una expresión `e`, en caso que la estructura sintáctica analizada no haya sido una expresión válida se regresa un caracter que indica el inicio de lo que no se pudo leer. En esta implementación del lector el caracter punto y coma delimita expresiones, por lo tanto, obtener como resultado `#\;` provoca que se analice la siguiente expresión del puerto.

La implementación de `parsear-expresión` considera un puerto de entrada `p` de donde se consumen caracteres, un caracter `c` que es el primer caracter del puerto y un acumulador de expresiones `a` que representa el árbol leído. A partir de `c` se puede determinar el tipo de expresión que se está leyendo.

```
13  <lector de expresiones 12>+≡
    (define (parsear-expresión [p (current-input-port)])
      [c (peek-char p)]
      [a null])

    (cond
      [(eof-object? c)
       <lectura de fin de archivo 14>]
      [(delimitador-final? c)
       <lectura de delimitador final 16>]
      [(char-whitespace? c)
       <lectura de espacio en blanco 19>]
      [(char=? #\ ( c)
       <lectura de paréntesis abierto 20>]
      [(char=? #\[ c)
       <lectura de hueco 23>]
      [(or (char=? (integer->char 955) c)
           (char=? #\\ c))
       <lectura de lambda 25>]
      [else
       <lectura de variable o metainstrucción 30>]]))
```

Define:

`parsear-expresión`, usado en fragmentos 12, 13, 19, 20, 13, 22, 23, 25, 26, 30, y 31.

Hace referencia a `delimitador-final?` 15.

El análisis de casos tiene tres cláusulas “triviales”: cuando no hay más que leer del puerto, cuando se lee un delimitador final y cuando se lee un espacio en blanco. Cuando el siguiente objeto del puerto de entrada es el “fin de archivo”, el lector debe regresar este objeto si el acumulador `a` es nulo, de lo contrario se regresa la expresión acumulada.

```
14  <lectura de fin de archivo 14>≡
    (if (null? a) c a)
```

Este código es usado en fragmento 13.

El siguiente caso es la lectura de un delimitador final, el procedimiento `delimitador-final?` se encarga de determinar si un caracter termina una expresión.

```
15 <procedimientos auxiliares de lectura 15>≡
  (define (delimitador-final? c)
    (or (eof-object? c)
        (char=? #\] c)
        (char=? #\ ) c)
        (char=? #\ . c)
        (char=? #\ , c)
        (char=? #\ ; c)))
```

Define:

`delimitador-final?`, usado en fragmentos 13, 15, y 18.

Esta definición se continua en fragmentos 17, 18, 21, 22, 24, 26–28, y 31.

Este código es usado en fragmento 32.

Cuando un objeto leído de un puerto de entrada satisface este predicado, el lector consume y regresa el delimitador si el acumulador `a` es nulo, de lo contrario regresa la expresión acumulada.

```
16 <lectura de delimitador final 16>≡
  (if (null? a) (read-char p) a)
```

Este código es usado en fragmento 13.

El caracter delimitador no es consumido en caso de que `a` no sea nulo ya que usualmente los delimitadores finales tienen asociado un delimitador inicial con el que deben concordar, por ejemplo, el predicado `delimitador-inicial?` se define⁵ como

```
17 <procedimientos auxiliares de lectura 15>+≡
  (define (delimitador-inicial? c)
    (or (char=? #\ ( c)
        (char=? #\[ c)
        (char=? (integer->char 955) c)
        (char=? #\\ c)))
```

Define:

`delimitador-inicial?`, usado en fragmentos 17 y 18.

En el lenguaje, el delimitador `#\ (` debe estar asociado al delimitador `#\)` y de igual manera con `#\[` y `#\]`. Esta verificación se realiza en otra parte del código, pero se debe preservar la información del delimitador leído.

Técnicamente, los caracteres delimitadores del lenguaje son los que satisfacen los predicados `delimitador-inicial?`, `delimitador-final?` y `char-whitespace?`, sin embargo es conveniente definirlos por separado ya que son tratados de manera diferente en la lectura de una expresión.

⁵Se realiza una comparación con `(integer->char 955)` ya que 955 es el valor numérico del caracter `λ` en Unicode.

Para determinar si algún carácter es delimitador (sin importar el tipo de delimitador que sea) se utiliza el predicado `delimitador?`.

```
18 <procedimientos auxiliares de lectura 15>+≡
    (define (delimitador? c)
      (or (delimitador-final? c)
          (delimitador-inicial? c)
          (char-whitespace? c)))
```

Define:

`delimitador?`, usado en fragmentos 18, 26–28, 18, y 32.

Hace referencia a `delimitador-final?` 15 y `delimitador-inicial?` 17.

El tercer y último caso “trivial” en la lectura es el de espacios en blanco. Al encontrarse un carácter de este tipo se consume del puerto y se llama de manera recursiva `parsear-expresión` de tal manera que el carácter es ignorado y se preserva el valor de la expresión acumulada `a`.

```
19 <lectura de espacio en blanco 19>≡
    (read-char p)
    (parsear-expresión p (peek-char p) a)
```

Hace referencia a `parsear-expresión` 13.

Este código es usado en fragmento 13.

El resto de los casos son significativamente más complejos que los primeros tres. Primero se aborda el leer un paréntesis abierto, esto puede ocurrir cuando se desea agrupar una aplicación o una abstracción. El procedimiento `parsear-paréntesis` se encarga de leer una expresión después de un paréntesis abierto y verificar que dicha expresión termina con un paréntesis cerrado.

```
20 <lectura de paréntesis abierto 20>≡
    (read-char p)
    (let ([e (parsear-paréntesis p)])
      (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```

Hace referencia a `aplicación` 3, `aplicación/identidad` 21, `parsear-expresión` 13, y `parsear-paréntesis` 22.

Este código es usado en fragmento 13.

Primero se consume el carácter `#\` (del puerto de entrada, después se llama el procedimiento `parsear-paréntesis` el cuál regresará la expresión `e` delimitada por `#\` (y `#\`). El procedimiento `parsear-expresión` es llamado de manera recursiva ya que es posible que esta expresión entre paréntesis sea parte de una aplicación escrita con abuso de notación, por ejemplo `(M N)P`. El resultado es acumulado con el procedimiento `aplicación/identidad`, este contempla la posibilidad que la expresión leída sea parte de una expresión de la forma `M(N P)` y se define como

```
21 <procedimientos auxiliares de lectura 15>+≡
    (define (aplicación/identidad a e)
      (if (null? a) e (aplicación a e)))
```

Define:

aplicación/identidad, usado en fragmentos 20, 21, 23, 25, 21, y 30.

Hace referencia a aplicación 3.

Cuando el acumulador *a* es nulo, el nuevo acumulador es la expresión *e*, de lo contrario, el nuevo acumulador es la aplicación de *a* con *e*.

El procedimiento *parsear-paréntesis* lee una expresión *e*, verifica que el resultado es una aplicación o una abstracción y verifica que el caracter después la representación textual de *e* es un paréntesis cerrado.

```
22 <procedimientos auxiliares de lectura 15>+≡
  (define (parsear-paréntesis [p (current-input-port)]
    [c (peek-char p)])
    (let ([e (parsear-expresión p c)])
      (unless (or (abstracción? e) (aplicación? e))
        (error 'parsear-paréntesis
          "Se esperaba una abstracción o una aplicación pero se leyó ~a" e))
      (let ([c (read-char p)])
        (unless (char=? #\) c)
          (error 'parsear-paréntesis
            "Se esperaba ')' pero se leyó '~a'" c))
        e))))
```

Define:

parsear-paréntesis, usado en fragmentos 22, 20, y 22.

Hace referencia a abstracción 2, aplicación 3, y *parsear-expresión* 13.

Ahora se aborda el caso de leer el caracter `#\[` este caracter es utilizado después de leer el nombre de una metainstrucción, sin embargo, cuando se encuentra este caracter por sí solo denota el inicio de un hueco, los cuales se conforman de un corchete abierto y uno cerrado con espacios en blanco opcionales entre los corchetes.

```
23 <lectura de hueco 23>≡
  (read-char p)
  (let ([e (parsear-hueco p)])
    (parsear-expresión p (peek-char p) (aplicación/identidad a e))))
```

Hace referencia a aplicación 3, aplicación/identidad 21, hueco 6, *parsear-expresión* 13, y *parsear-hueco* 24.

Este código es usado en fragmento 13.

```
24 <procedimientos auxiliares de lectura 15>+≡
  (define (parsear-hueco [p (current-input-port)]
    [c (peek-char p)])
    (cond [(char-whitespace? c)
      (read-char p)
      (parsear-hueco p (peek-char p))])
```



```

[(char=? #\ ] c)
  (read-char p)
  (hueco)]
[else
  (error 'parsear-hueco
    "Se esperaba ']' pero se leyó '~a'" c)))]))

```

Define:

parsear-hueco, usado en fragmento 23.

Hace referencia a hueco 6.

El siguiente caso es leer un caracter que representa a λ , en esta implementación estos caracteres son `#\\` y el resultado de `(integer->char 955)` que es el caracter Unicode de la letra lambda minúscula. De manera similar a la lectura de paréntesis abierto, se consume del puerto el caracter leído, se obtiene la abstracción con el procedimiento `parsear-abstracción` y se manejan los abusos de notación de la aplicación.

25 *<lectura de lambda 25>* \equiv

```

(read-char p)
(let ([e (parsear-abstracción p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))

```

Hace referencia a `abstracción` 2, `aplicación` 3, `aplicación/identidad` 21, `parsear-abstracción` 26, y `parsear-expresión` 13.

Este código es usado en fragmento 13.

La implementación del procedimiento `parsear-abstracción` es un poco más compleja a la de `parsear-paréntesis`, la representación textual de las abstracciones con el abuso de notación es de la forma `'"\ x-1 x-2 ... x-n . M'`, ya que el procedimiento inicia después de consumir el caracter de λ , primero se deben identificar todas las variables antes del punto y regresar una abstracción “descurrificada”.

26 *<procedimientos auxiliares de lectura 15>* $+\equiv$

```

(define (parsear-abstracción [p (current-input-port)]
  [c (peek-char p)])
  (let itera ([c c]
    [a null])
    (cond [(or (eof-object? c)
      (and (delimitador? c) (not (char-whitespace? c)) (not (char=? #\. c))))
      (error 'parsear-abstracción
        "Se esperaba una variable pero se leyó '~a'" c)]
      [(char-whitespace? c)
        (read-char p)
        (itera (peek-char p) a)]
      [(char=? #\. c)
        (read-char p)
        (if (null? a)

```

```

(error 'parsear-abstracción
  "Se esperaba una variable pero se leyó '~a'" c)
(let ([e (parsear-expresión p)])
  (if (expresión? e)
      (foldr abstracción e (reverse a))
      (error 'parsear-abstracción
        "Se esperaba una expresión pero se leyó ~a" e))))]
[else
  (let ([v (variable (parsear-constituyente p))])
    (itera (peek-char p) (cons v a))))))

```

Define:

parsear-abstracción, usado en fragmentos 26, 25, y 26.

Hace referencia a abstracción 2, delimitador? 18, expresión? 7, parsear-constituyente 27, parsear-expresión 13, y variable 1.

El procedimiento parsear-constituyente se utiliza para consumir del puerto de entrada una cadena de caracteres constituida por no-delimitadores.

27 \langle procedimientos auxiliares de lectura 15 $\rangle + \equiv$

```

(define (parsear-constituyente [p (current-input-port)]
  [c (peek-char p)])

  (let itera ([a null]
    [c c])
    (if (delimitador? c)
        (if (null? a)
            (error 'parsear-constituyente
              "Se esperaba un identificador pero se leyó '~a'" c)
            (list->string (reverse a)))
        (itera (cons (read-char p) a)
          (peek-char p)))))

```

Define:

parsear-constituyente, usado en fragmentos 26, 27, y 29.

Hace referencia a delimitador? 18.

Las cadenas conformadas por caracteres constituyentes son usadas en los nombres de las variables y metainstrucciones, para determinar si una cadena es un “nombre” se puede utilizar el predicado nombre?.

28 \langle procedimientos auxiliares de lectura 15 $\rangle + \equiv$

```

(define (nombre? x)
  (and (string? x)
    (not (string=? x ""))
    (not (any delimitador? (string->list x)))))

```

Define:

nombre?, usado en fragmentos 28 y 32.
 Hace referencia a delimitador? 18.

La última cláusula del análisis de casos se encarga de leer variables o metainstrucciones, esto es debido a que ambas inician con la lectura de caracteres constituyentes. Para obtener la variable o la expresión se lee una cadena de caracteres constituyentes del puerto de entrada, si el delimitador de esta cadena es #\[entonces la expresión debe ser una metainstrucción, de lo contrario debe ser una variable.

29 $\langle \text{obtener variable o metainstrucción } 29 \rangle \equiv$

```
(let* ([x (parsear-constituyente p c)]
      [c (peek-char p)])
  (cond [(and (char? c) (char=? #\[ c))
        (read-char p)
        (metainstrucción x (parsear-corchetes p))]
        [else
         (variable x)]))
```

Hace referencia a metainstrucción 5, parsear-constituyente 27, parsear-corchetes 31, y variable 1.
 Este código es usado en fragmento 30.

Al igual que en el caso del paréntesis abierto y que la λ , se debe acumular este valor utilizando aplicación/identidad.

30 $\langle \text{lectura de variable o metainstrucción } 30 \rangle \equiv$

```
((lambda (e)
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
   $\langle \text{obtener variable o metainstrucción } 29 \rangle$ )
```

Hace referencia a aplicación 3, aplicación/identidad 21, y parsear-expresión 13.
 Este código es usado en fragmento 13.

En el caso que la expresión termine con el corchete abierto, se utiliza el procedimiento parsear-corchetes para obtener una lista de expresiones separadas por coma y delimitadas por #\[. Es importante tener en cuenta que no se admiten espacios entre el nombre de una metainstrucción y el corchete que denota el inicio de su lista de argumentos.

31 $\langle \text{procedimientos auxiliares de lectura } 15 \rangle + \equiv$

```
(define (parsear-corchetes [p (current-input-port)]
  [c (peek-char p)])
  (let itera ([a null]
    [e (parsear-expresión p c)])
    (cond [(eof-object? e)
          (error 'parsear-corchetes
            "Se esperaba leer '[' o ',' pero se leyó ~a" e)]
          [(char? e)
           (case e
            [(#\[) (reverse a)]
            [(#\\,) (itera a (parsear-expresión p))])])
```

```

      [else
        (error 'parsear-corchetes
          "Se esperaba leer '[' o ',' pero se leyó ~a" e)]]]
    [else
      (itera (cons e a) (parsear-expresión p))]]))

```

Define:

parsear-corchetes, usado en fragmentos 29 y 31.

Hace referencia a leer 12 y parsear-expresión 13.

Módulo de lector El módulo del lector utiliza el módulo de estructuras.rkt y provee el procedimiento leer, evitando que los usuarios del parser malutilicen los procedimientos auxiliares de lectura. Adicionalmente se proveen los predicados delimitador? y nombre? que pueden ser reutilizados en otras partes del programa. El archivo lector.rkt contiene este módulo y tiene la siguiente forma:

```

32 <lector.rkt 32>≡
    ;;; -*- mode: racket; coding: utf-8 -*-
    ;;; Copyright (C) 2016 Eduardo Acuña Yeomans <eduardo.acye@gmail.com>
    ;;; Procura no editar este archivo, mejor modifica lambda.nw

    #lang racket/base

    (require srfi/1
      "estructuras.rkt")

    ;;; Exportar punto de entrada del parser
    (provide leer
      delimitador? nombre?)

    ;;; Procedimientos principales
    <lector de expresiones 12>

    ;;; Procedimientos auxiliares
    <procedimientos auxiliares de lectura 15>

```

Hace referencia a delimitador? 18, leer 12, y nombre? 28.

Fragmento raíz (no usado en este documento).

Pruebas del lector El archivo de pruebas lector-pruebas.rkt es utilizado para verificar si la funcionalidad del lector es la esperada. Se omite su definición en este documento pero es distribuido con el código del trabajo.

Escritura de expresiones

Así como se tiene un mecanismo para pasar de la representación externa textual de una expresión a una estructura en memoria, también se tiene un mecanismo para realizar el procedimiento inverso, es decir, escribir la representación externa textual de una estructura en memoria.

Diversidad en la escritura En el caso del lenguaje de expresiones esto es algo complicado ya que hay varias representaciones textuales para una sola expresión, por ejemplo, formalmente correcta o con algún abuso de notación. Por lo tanto, la escritura de expresiones se maneja utilizando diferentes procedimientos que resultan en estilos diversos de escritura.

Una cuestión importante que se debe tener en cuenta es que una misma representación externa textual puede escribirse para ser interpretada de manera diferente, por ejemplo, se pudieran escribir diferentes “backends” de escritura: texto plano, L^AT_EX, postscript, o incluso escribir la estructura en memoria de Racket a otro lenguaje de programación en la que pueda ser representada.

Los escritores que son implementados en el programa son:

- A texto plano sin abuso de notación;
- A texto plano con abuso de notación;
- A L^AT_EX sin abuso de notación;
- A L^AT_EX con abuso de notación;
- A una figura funcional de Racket sin abuso de notación;
- A una figura funcional de Racket con abuso de notación.

Se abordan estos estilos en orden

De expresiones a texto plano Para traducir una expresión a texto plano se procesa el árbol de manera recursiva, concatenando cadenas de caracteres con la representación externa de las subexpresiones.

Primero define el procedimiento expresión->texto-plano el cuál se encarga de identificar la clase de expresión que se desea traducir.

```

33 <A texto plano sin abuso 33>≡
  (define (expresión->texto-plano e)
    (cond
      [(variable? e)
       <Traducir variable a texto plano sin abuso 35>]
      [(abstracción? e)
       <Traducir abstracción a texto plano sin abuso 36>]
```

```

[(aplicación? e)
  <Traducir aplicación a texto plano sin abuso 37>]
[(hueco? e)
  <Traducir hueco a texto plano sin abuso 38>]
[(metainstrucción? e)
  <Traducir metainstrucción a texto plano sin abuso 39>]
[else
  (error 'expresión->texto-plano
    "La expresión ~a no es válida" e)]]))

```

Define:

expresión->texto-plano, usado en fragmentos 33, 36, 37, 39, 33, y 47.

Hace referencia a abstracción 2, aplicación 3, hueco 6, metainstrucción 5, y variable 1.

Esta definición se continua en fragmento 34.

Este código es usado en fragmento 47.

Para traducir una variable a texto plano sin abuso de notación, basta con obtener el nombre de la variable ya que los nombres de las variables ya son cadenas de caracteres. Sin embargo, la traducción de variables es utilizada en la traducción de abstracciones, por lo tanto es conveniente definir el procedimiento variable->texto-plano el cuál corrobora que la expresión es una variable y regresa su nombre.

```

34 <A texto plano sin abuso 33>+≡
  (define (variable->texto-plano e)
    (if (variable? e)
        (variable-nombre e)
        (error 'variable->texto-plano
          "La expresión ~a no es una variable" e)))

```

Define:

variable->texto-plano, usado en fragmentos 34-36 y 41.

Hace referencia a variable 1.

En expresión->texto-plano solo se invoca este procedimiento

```

35 <Traducir variable a texto plano sin abuso 35>≡
  (variable->texto-plano e)

```

Hace referencia a variable 1 y variable->texto-plano 34.

Este código es usado en fragmento 33.

Para traducir una abstracción a texto plano sin abuso de notación, primero se debe traducir su argumento y su cuerpo y después concatenarlos de la forma (λ argumento.cuerpo).

```

36 <Traducir abstracción a texto plano sin abuso 36>≡
  (format "(~a~a.~a)"
    (integer->char 955)
    (variable->texto-plano (abstracción-argumento e)))

```

```
(expresión->texto-plano (abstracción-cuerpo e)))
```

Hace referencia a abstracción 2, expresión->texto-plano 33, variable 1, y variable->texto-plano 34.
Este código es usado en fragmento 33.

Para traducir una aplicación a texto plano sin abuso de notación, se traduce su operador y su operando y después se concatenan de la forma (operador operando).

```
37 <Traducir aplicación a texto plano sin abuso 37>≡
  (format "(~a ~a)"
    (expresión->texto-plano (aplicación-operador e))
    (expresión->texto-plano (aplicación-operando e)))
```

Hace referencia a aplicación 3 y expresión->texto-plano 33.
Este código es usado en fragmento 33.

La traducción de un hueco a texto plano sin abuso de notación es simplemente la cadena compuesta por el corchete abierto y el corchete cerrado.

```
38 <Traducir hueco a texto plano sin abuso 38>≡
  "[ ]"
```

Este código es usado en fragmento 33.

La traducción de una metainstrucción a texto plano sin abuso de notación es un poco más complicada que en los otros casos. Primero se verifica que el nombre de la metainstrucción es una cadena de caracteres y que los argumentos de la metainstrucción sean una lista, después se obtiene una lista de las traducciones de cada expresión en la lista de argumentos de la metainstrucción, teniendo esta lista se concatenan comas entre cada argumento de la lista.

```
39 <Traducir metainstrucción a texto plano sin abuso 39>≡
  (let ([nombre (metainstrucción-nombre e)]
        [argumentos (metainstrucción-argumentos e)])
    (if (and (string? nombre) (list? argumentos))
        (format "~a[~a]"
          nombre
          (string-join (map expresión->texto-plano argumentos) ", "))
        (error 'metainstrucción->texto-plano
          "La metainstrucción ~a está mal formada" e)))
```

Hace referencia a expresión->texto-plano 33 y metainstrucción 5.
Este código es usado en fragmento 33.

En ocasiones será de utilidad obtener la representación externa de una expresión con las convenciones de abuso de notación utilizadas en este trabajo. La lectura de expresiones admite el abuso de notación en cualquier parte de la expresión, sin embargo, para la escritura de expresiones con abuso, habrá una sola representación externa con abuso de notación y será aquella que aproxime la máxima cantidad de abuso de notación posible.

La implementación del escritor para texto plano con abuso de notación es el procedimiento expresión->abuso-texto-plano el cuál tiene una forma similar a expresión->texto-plano

sólo que en el análisis de casos se invocan procedimientos especializados para cada clase de expresión.

```

40  <A texto plano con abuso 40>≡
    (define (expresión->abuso-texto-plano e)
      (cond
        [(variable? e)
         (variable->abuso-texto-plano e)]
        [(abstracción? e)
         (abstracción->abuso-texto-plano e)]
        [(aplicación? e)
         (aplicación->abuso-texto-plano e)]
        [(hueco? e)
         (hueco->abuso-texto-plano e)]
        [(metainstrucción? e)
         (metainstrucción->abuso-texto-plano e)]
        [else
         (error 'expresión->abuso-texto-plano
                  "La expresión ~a no es válida" e)]))

```

Define:

expresión->abuso-texto-plano, usado en fragmentos 40, 42, 43, 40, 45, y 47.

Hace referencia a abstracción 2, abstracción->abuso-texto-plano 42, aplicación 3, aplicación->abuso-texto-plano 43, hueco 6, hueco->abuso-texto-plano 44, metainstrucción 5, metainstrucción->abuso-texto-plano 45, variable 1, y variable->abuso-texto-plano 41.

Esta definición se continua en fragmentos 41-45.

Este código es usado en fragmento 47.

Para traducir una variable a texto plano con abuso de notación, se hace uso del procedimiento variable->texto-plano ya que las variables tienen una sola representación externa en texto plano.

```

41  <A texto plano con abuso 40>+≡
    (define (variable->abuso-texto-plano e)
      (variable->texto-plano e))

```

Define:

variable->abuso-texto-plano, usado en fragmentos 40 y 42.

Hace referencia a variable 1 y variable->texto-plano 34.

La traducción de abstracciones a texto plano con abuso de notación consiste en revisar la clase de expresión que es el cuerpo, en caso que sea también una abstracción se debe iterar sobre los cuerpos de las abstracciones acumulando la representación externa de sus argumentos hasta llegar a un cuerpo que no sea abstracción. La forma de la cadena resultante debe ser $\lambda \text{argumento}_1 \text{ argumento}_2 \dots \text{argumento}_n . \text{cuerpo}_n$.

42 $\langle A \text{ texto plano con abuso } 40 \rangle + \equiv$

```

(define (abstracción->abuso-texto-plano e)
  (let iter ([argumentos (list (abstracción-argumento e))]
            [cuerpo      (abstracción-cuerpo e)])
    (if (abstracción? cuerpo)
        (itera (cons (abstracción-argumento cuerpo) argumentos)
                (abstracción-cuerpo cuerpo))
        (format "~a~a.~a"
                  (integer->char 955)
                  (string-join (map variable->abuso-texto-plano
                                     (reverse argumentos))
                               " ")
                  (expresión->abuso-texto-plano cuerpo))))))

```

Define:

abstracción->abuso-texto-plano, usado en fragmentos 40 y 43.

Hace referencia a abstracción 2, expresión->abuso-texto-plano 40, variable 1, y variable->abuso-texto-plano 41.

Para traducir una aplicación a texto plano con abuso de notación primero se computa una lista de términos conformada por los operandos anidados de las aplicaciones de la forma $((M N)P)$ donde el primer operador que no sea aplicación es el primer elemento de la lista, es decir, se obtiene la lista de expresiones removiendo la asociación a la izquierda de la aplicación original, por ejemplo si consideramos una aplicación de la forma

$$(\dots (((((M_1 M_2) M_3) M_4) M_5) \dots M_n))$$

la lista de expresiones sería

$$(\text{list } M_1 M_2 M_3 M_4 M_5 \dots M_n)$$

donde M_1 no es una aplicación.

Después de remover la asociación a la izquierda se determinan los elementos de la lista que deben estar encerrados entre paréntesis. El criterio utilizado en general es: si el elemento de la lista es una aplicación o una abstracción, va encerrada entre paréntesis, de lo contrario, se escribe tal cuál. Además se aplica un criterio diferente para un caso en particular: cuando el último elemento de la lista es una abstracción, no se encierra entre paréntesis, esto es para que una expresión como

$$(((M N) P) (\lambda x.Q))$$

pueda ser escrita como

$$M N P \lambda x.Q$$

la cuál es una expresión sintácticamente equivalente. Esto no se puede realizar para las abstrac-

ciones en general ya que una expresión como

$$((M (\lambda x.N)) P)$$

sería considerada equivalente a

$$M \lambda x.N P$$

sin embargo, esta última expresión corresponde a

$$(M (\lambda x.(N P)))$$

al tener los paréntesis escritos de manera adecuada se concatenan las expresiones con un espacio en blanco.

El procedimiento aplicación->abuso-texto-plano implementa este algoritmo para el abuso de notación.

```
43 <A texto plano con abuso 40>+≡
(define (aplicación->abuso-texto-plano e)
  (let itera ([operador (aplicación-operador e)]
             [expresiones (list (aplicación-operando e))])
    (if (aplicación? operador)
        (itera (aplicación-operador operador)
                (cons (aplicación-operando operador) expresiones))
        (let itera ([expresiones (cons operador expresiones)]
                    [traducciones null])
          (if (null? expresiones)
              (string-join (reverse traducciones) " ")
              (let ([e (first expresiones)]
                    [es (rest expresiones)])
                (cond
                 [(aplicación? e)
                  (itera es (cons (format "~a" (aplicación->abuso-texto-plano e))
                                  traducciones))]
                 [(and (abstracción? e) (not (null? es)))
                  (itera es (cons (format "~a" (abstracción->abuso-texto-plano e))
                                  traducciones))]
                 [else
                  (itera es (cons (expresión->abuso-texto-plano e)
                                  traducciones))]))))))))
```

Define:

aplicación->abuso-texto-plano, usado en fragmentos 40 y 43.

Hace referencia a abstracción 2, abstracción->abuso-texto-plano 42, aplicación 3, y expresión->abuso-texto-plano 40.

La traducción a texto plano con abuso de notación para huecos es exactamente la misma que cuando no se tiene abuso de notación.

```
44 <A texto plano con abuso 40>+≡
    (define (hueco->abuso-texto-plano e)
      "[ ]")
```

Define:

hueco->abuso-texto-plano, usado en fragmento 40.

Hace referencia a hueco 6.

En el caso de la traducción de metainstrucciones a texto plano con abuso de notación se emplea el mismo procedimiento que en el caso donde no hay abuso de notación, con la excepción de que las llamadas recursivas a la traducción de argumentos se realizan invocando el procedimiento expresión->abuso-texto-plano en lugar de expresión->texto-plano.

```
45 <A texto plano con abuso 40>+≡
    (define (metainstrucción->abuso-texto-plano e)
      (let ([nombre (metainstrucción-nombre e)]
            [argumentos (metainstrucción-argumentos e)])
        (if (and (string? nombre) (list? argumentos))
            (format "~a[~a]"
                    nombre
                    (string-join (map expresión->abuso-texto-plano argumentos) ", "))
            (error 'metainstrucción->abuso-text-plano
                    "La metainstrucción ~a está mal formada" e))))
```

Define:

metainstrucción->abuso-texto-plano, usado en fragmento 40.

Hace referencia a expresión->abuso-texto-plano 40 y metainstrucción 5.

De expresiones a L^AT_EX Traducir expresiones a código interpretable por L^AT_EX es relativamente fácil, lo complicado es la diversidad de estilos con los que se puede escribir una expresión. Por ejemplo, el espaciado entre dos términos en una aplicación, el color de algún subtérmino, la tipografía de los símbolos o la manera en como se escriben los paréntesis. Para solventar estos problemas se reduce el problema de generar un archivo válido de L^AT_EX a simplemente producir una cadena de caracteres que al ser insertada en un documento válido y en un lugar donde se aplique el modo de matemáticas, el archivo seguirá siendo válido y producirá una expresión sintácticamente correcta.

Se define el parámetro `plantilla-latex` el cuál debe contener una cadena de caracteres que podrá ser utilizado de tal manera que `(format (plantilla-latex) expresión-traducida)` sea una cadena de caracteres con el contenido del documento de L^AT_EX válido. Para evitar complicar la implementación de este parámetro, la única condición que se verifica es que sea una cadena de caracteres y que contenga únicamente una secuencia `~a`.

```

46 <A LaTeX sin abuso 46>≡
  (define +plantilla-latex-inicial+
    "\
    \documentclass[preview]{standalone}\n\
    \usepackage{amsmath}\n\
    \begin{document}\n\
    \(\ ~a \)\n\
    \end{document}")

  (define plantilla-latex
    (make-parameter
      +plantilla-latex-inicial+
      (lambda (x)
        (if (and (string? x) (= (length (string-split x "~a" #:trim? #f)) 2))
            x
            +plantilla-latex-inicial+))))

```

Define:

plantilla-latex, usado en fragmentos 46 y 47.

Este código es usado en fragmento 47.

De expresiones a figuras funcionales de Racket

Módulo de escritor El módulo de los escritores utiliza el módulo `estructuras.rkt` y provee los procedimientos de escritura a texto plano, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ y a figuras de Racket con abuso y sin abuso de notación. El archivo `escritor.rkt` contiene este módulo y tiene la siguiente forma.

```

47 <escritor.rkt 47>≡
  ;;; -*- mode: racket; coding: utf-8 -*-
  ;;; Copyright (C) 2016 Eduardo Acuña Yeomans <eduardo.acye@gmail.com>
  ;;; Procura no editar este archivo, mejor modifica lambda.nw

  #lang racket/base

  (require "estructuras.rkt"
    racket/list
    racket/string)

  (provide expresión->texto-plano
    expresión->abuso-texto-plano
    plantilla-latex)

  ;;; Traducir a texto plano sin abuso de notación
  <A texto plano sin abuso 33>

```

;;; Traducir a texto plano con abuso de notación
 $\langle A \text{ texto plano con abuso } 40 \rangle$

;;; Traducir a LaTeX sin abuso de notación
 $\langle A \text{ LaTeX sin abuso } 46 \rangle$

Hace referencia a expresión->abuso-texto-plano 40, expresión->texto-plano 33, y plantilla-latex 46.
 Fragmento raíz (no usado en este documento).

Pruebas de escritores El archivo de pruebas `escritor-pruebas.rkt` es utilizado para verificar si la funcionalidad de los escritores es la esperada. Se omite su definición en este documento pero es distribuido con el código del trabajo.

Evaluador de expresiones

Pendiente

Ciclo de Lectura Evaluación y Escritura

Pendiente

Editor estructural

Pendiente

¿Cómo hackear la λ ?

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 York, 1941.
- [10] CHURCH, A., AND ROSSER, J. B. Some properties of conversion. *Transactions of the American Mathematical Society* 39 (1936), 472–482. <http://www.jstor.org/stable/2268573>Electronic Edition.
- [11] CROSSLEY, J. N. *Reminiscences of logicians*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975, pp. 1–62.
- [12] 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.

- [13] 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.
- [14] HINDLEY, J. R., AND SELDIN, J. P. *Lambda-Calculus and Combinators*, 2 ed. Cambridge University Press, 2008.
- [15] HOFSTADTER, D. R. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979.
- [16] 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.
- [17] KNUTH, D. E. Mathematics and computer science: Coping with finiteness. *Science* 194, 4271 (1976), 1235–1242.
- [18] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM* 3, 4 (Apr. 1960), 184–195.
- [19] MENDELSON, E. *Introduction to Mathematical Logic*, 5 ed. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2010.
- [20] 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.
- [21] PEYTON JONES, S., ET AL. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming* 13, 1 (Jan 2003).
- [22] PIERCE, B. C. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- [23] QUINE, W. V. *The Ways of Paradox, and Other Essays*. Harvard University Press, 1976.
- [24] S. C. KLEENE, J. B. R. The inconsistency of certain formal logics. *Annals of Mathematics* 36, 3 (1935), 630–636.
- [25] SCHÖNFINKEL, M. Über die bausteine der mathematischen logik. *Mathematische Annalen* 92, 3 (1924), 305–316.
- [26] SCOTT, D. S. Lambda calculus then and now. In *ACM Turing Centenary Celebration (2012)*, ACM-TURING '12, ACM.
- [27] 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.
- [28] 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.

- [29] SUSSMAN, G. J., AND STEELE JR, G. L. SCHEME : an interpreter for extended lambda calculus. Tech. Rep. AI 349, Massachusetts Institute of Technology (MIT). Cambridge (MA US), 1975.