El cálculo λ y los fundamentos de la computación

Eduardo Acuña Yeomans

2016

Dedicado a los hackers de LCC

Agradecimientos

Agradezco a los profes que revisaron este trabajo: Olivia Gutú, Julio Waissman, Martín Frías y Jesús Espinoza. En particular al Frías por acompañarme en la exploración inicial de los temas y a Gutú por las sesiones de discusión y por su revisión matona (tanto en contenido como en forma) del presente trabajo.

Agradezco a los profesores Irene Rodríguez, Olivia Gutú, Julio Waissman y Donald Rodríguez por impartir las clases de computación que más disfruté y a los profesores Fernando Luque, Gabriela Robles, Martín Frías y Jacobo Núñez por impartir las clases de matemáticas que más disfruté.

Agradezco a mis compañeros de LCC y matemáticas por su amistad y compañía, en particular a Manuel Valle Ruiz y Diana Rivera Segundo a quienes admiro y fueron generadores de benevolencia e inspiración.

Agradezco a María José Flores por las tardes de hackeo en las banquitas de mate, por las pláticas en los cactus y por erradicar gnomos.

Agradezco a mis hermanos Esteban y Emilio y a mis progenitores Heriberto y Laura, por existir y ser como son.

Agradezco al Software Libre, a The Pirate Bay, a Library Genesis, a Sci-Hub y a OpenCourseWare por su razón de ser.

Índice general

Introducción

Descripción del trabajo

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.

El objetivo de este trabajo es explorar y plantear los aspectos básicos del cálculo λ relacionados con la computación a un nivel adecuado para estudiantes de la LCC de la Universidad de Sonora con la finalidad de promover el estudio de las ramas teóricas de las Ciencias de la Computación.

Aportaciones principales

Este trabajo se basa fuertemente en varios libros sobre el cálculo λ y la aportación principal es la adaptación y redacción de los temas teniendo en cuenta los conocimientos de un estudiante de la LCC de últimos semestres. Otras aportaciones realizadas son:

- El Algoritmo 2.1 para compilar cualquier término cerrado a los combinadores $\boldsymbol{SKI}$;
- Las codificaciones de operaciones booleanas basadas en la codificación del término condicional en la Subsección 3.1.2 y Subsección 3.1.3;
- Las codificaciones de operaciones aritméticas basadas en numerales de Church en la Subsección 3.2.2 y Subsección 3.2.4;
- El mecanismo para codificar algoritmos iterativos en la Subsección 3.2.3;
- La codificación de términos λ en el cálculo λ en la Subsección 3.4.4;
- La implementación de las codificaciones del Capítulo 3 en Haskell y Scheme en el Apéndice A;
- El programa Lambda en el Apéndice B.

Estructura del trabajo

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 la implementación de un intérprete y editores para un lenguaje diseñado con la finalidad de explorar los temas de este trabajo de manera interactiva.

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]. 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 funcional usada en el cálculo en comparación a la notación matemática usual; 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 [15]; y la monografía titulada "The Calculi of Lambda-Conversion" por Alonzo Church [10].

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 $f : \mathbb{Z} \to \mathbb{Z}$ tal que

$$f = \{(x,\ 2x + y) \mid \text{ para todo } x \in \mathbb{Z}\}$$

se escribe con la notación convencional como

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

En la notación funcional usada en el cálculo λ, f se escribe como

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

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 usual mencionada 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 capacidad de denotar simbólicamente funciones anónimas resulta ser de utilidad ya que nos permite 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 en $\lambda y.3y$ se tiene que

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

En varias áreas de las matemáticas se utilizan otras notaciones funcionales para expresar funciones anónimas. Por ejemplo, la notación $\mapsto$ es similar a la notación λ y puede ser usada para denotar los ejemplos anteriores. Con esta otra notación, la función $f = \lambda g.\lambda x.2(g\ x)$ es escrita

$$f = g \mapsto (x \mapsto 2g(x))$$

mientras que el desarrollo de la evaluación de f en $y \mapsto 3y$ es

$$\begin{aligned}f(y \mapsto 3y) &= [g \mapsto (x \mapsto 2g(x))](y \mapsto 3y)\\ &= x \mapsto 2[y \mapsto 3y](x)\\ &= x \mapsto 2(3x)\\ &= x \mapsto 6x\end{aligned}$$

A pesar de poder utilizar notaciones funcionales equivalentes pero sintácticamente diferentes a la del cálculo λ, en este trabajo se opta por utilizar la notación λ debido a su ubicuidad en los textos que tratan temas relacionados con este cálculo.

Poniendo a un lado las cuestiones superficiales, la diferencia conceptual más importante entre el cálculo λ y el enfoque clásico del estudio de funciones es que este cálculo permite expresar únicamente funciones, evaluaciones y variables. Otros objetos matemáticas como los números y los conjuntos no son permitidos en el cálculo λ. Las implicaciones de esto son fuertes ya que en la matemática clásica una función es representada como un conjunto y la notación utilizada al tratar funciones es únicamente "azúcar sintáctico" sobre la notación usada en teoría de conjuntos, mientras que en el cálculo λ todo es representado como cadenas de símbolos de un lenguaje formal.

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 y 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 en forma de expresiones.

Es posible utilizar la definición de función basada en conjuntos para describir operaciones o transformaciones de expresiones y utilizar lógica de primer orden o aritmética para aseverar o cuantificar propiedades, sin embargo estos objetos matemáticos son parte del metalenguaje y no del lenguaje del cálculo λ.

1.2. Expresiones

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

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.

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.

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 el cálculo λ el cuantificador de variables es el símbolo λ de las 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 *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 $(\lambda x.x)$, esta se conforma por dos variables representadas con el mismo átomo, la expresión x por sí sola no es de mucha utilidad, sin embargo, al ser parte de esta abstracción toma importancia ya que si pensamos en las abstracciones como funciones, la x podrá ser cambiada por alguna otra expresión.

Las aplicaciones, al igual que las abstracciones, son expresiones con estructura. Se con-

forman por otras dos expresiones y se escriben de la forma

$$(M\,N)$$

donde M y N son cualesquiera átomos, abstracciones o aplicaciones.

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 estructura sintáctica para denotar una evaluación como $f(x)$ y el acto de encontrar el valor asociado a x en f. En el cálculo λ el primero se refiere a las aplicaciones y el segundo se refiere a las *reducciones* las cuales serán abordadas más a detalle en la siguiente sección.

A pesar de estar asociado a la evaluación de funciones, una aplicación válida pudiera ser

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

Esto puede resultar extraño ya que la parte izquierda de una evaluación es una función, las funciones están asociadas a las abstracciones y en este ejemplo y es una variable. Sin embargo, la y en la expresión pudiera estar cuantificada en la expresión

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

y ahora al reducir esta expresión con otra abstracción como por ejemplo $(\lambda w.w)$, la y en el cuerpo se referirá a $(\lambda w.w)$ la cuál pudiera ser a su vez reducida con $(\lambda x.x)$. Es por esto que en el cálculo λ si una expresión es válida, sus partes también lo son.

Todas las combinaciones válidas de expresiones nos permiten formar estructuras sintácticas diversas y asignarles significados interesantes. Por ejemplo, considerando la expresión $(\lambda x.(x\,x))$ y teniendo en cuenta la analogía entre abstracciones, funciones, aplicaciones, reducciones y evaluaciones se puede considerar que esta cadena de símbolos representan el concepto de duplicación de expresiones, ya que reducir

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

resulta en $(y\,y)$ y reducir

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

resulta en $((y\,y)\,(y\,y))$. Por otro lado, reducir

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

resulta en sí misma. A este tipo de expresiones se les llaman "quines" [16, pp. 431–437] término originalmente asociado a una paradoja sobre sistemas lógicos [24]. En la actualidad,

el término "quine" hace referencia a un programa cuya *salida* es el programa mismo.

Otro ejemplo de una expresión interesante es

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

Al reducirla con una expresión cualquiera M se obtiene $(\lambda x.(M\,(M\,x)))$ si esta expresión resultante es reducida con otra expresión cualquiera N el resultado es $(M\,(M\,N))$. Este proceso puede pensarse como si una función de dos argumentos fuera evaluada en dos valores, bajo esta analogía una abstracción cuyo cuerpo sea otra abstracción se pudiera considerar como una abstracción con dos argumentos.

Asignarle significados a expresiones del cálculo λ es una tarea que debe realizarse con cuidado. Debido a que usualmente se trabaja con este cálculo en contextos rigurosos es de extrema importancia ser consistentes con la interpretación que se le está dando a las expresiones. 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 $f : \mathbb{N} \to \mathbb{N}$ 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.

1.3. 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 mencionó que las *reducciones* son análogas al acto de evaluar una función en un valor. Considerando la función $f(x) = 2x+3$ la evaluación de $f(4)$ resulta en 11 el cuál puede ser obtenido sustituyendo en $2x+3$ la x por 4. Las operaciones más importantes

del cálculo λ se basan en este proceso de sustitución y serán abordadas en esta sección.

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 fácilmente, consideremos por ejemplo la operación

$$x[x:=y]$$

ya que x es la expresión en donde x es cambiada por y, el resultado es y. De manera similar, se pueden involucrar expresiones más complejas en donde es sencillo encontrar el resultado de la sustitución. Por ejemplo

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

resulta en $(z(z(\lambda y.y)))$ y

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

resulta en $((\lambda w.w)(\lambda w.w))$

Sin embargo, 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 análoga a la función constante $f(x) = y$:

$$(\lambda x.y)$$

Reducir esta expresión 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*. Por ejemplo, cuando se considera la misma abstracción pero se desea sustituir la y por una x:

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

el resultado de simplemente cambiar el átomo y por el átomo x resultaría en

$$(\lambda x.x)$$

la cuál no tiene la interpretación de la función constante, esta expresión se pudiera considerar análoga a la función identidad.

La operación de sustitución en el cálculo λ así como en otras áreas de la matemática y la lógica no admite 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 una variable *libre*: No está en el *alcance* de alguna λz y puede ser libremente sustituida por alguna otra expresión.

En el caso de $(\lambda x.y)[y := x]$ se pretende sustituir la variable libre y por una expresión x, un cambio de átomos 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 definir el comportamiento de la operación de sustitución cuando se presentan los conflictos mencionados 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. En el cálculo λ, cambiar el símbolo que representa el átomo x en la expresión $(\lambda x.y)$ por otro símbolo no utilizado nos permite realizar la sustitución sin abordar los conflictos mencionados.

Para realizar la sustitución $(\lambda x.y)[y := x]$ primero se realiza un cambio de variable ligada en $(\lambda x.y)$ para obtener, por ejemplo, $(\lambda z.y)$, con la cuál se puede replantear la operación como

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

la cuál resulta en $(\lambda z.x)$ y mantiene la interpretación de una función constante.

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 *vinculada* como todas las variables ligadas a λx 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 $(\lambda x.M)$.

La operación de cambiar una variable ligada x a y en una abstracción $(\lambda x.M)$ se denota $\overset{y}{\rightarrow}_\alpha$ y se define en base a la operación de sustitución como

$$(\lambda x.M) \overset{y}{\rightarrow}_\alpha (\lambda y.M[x:=y])$$

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 se define la sustitución de acuerdo a los siguientes casos:

- $x[x:=M] = M$;
- $y[x:=M] = y$;
- $(M\,N)[x:=P] = (M[x:=P]\,N[x:=P])$;
- $(\lambda x.M)[x:=N] = (\lambda x.M)$ 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 manera similar a como se pudiera realizar la evaluación de funciones, el concretar una función consiste en sustituir todas las apariciones del argumento por el valor en el que la función es aplicada.

La reducción de aplicaciones se puede realizar sobre aplicaciones en donde la expresión izquierda es una abstracción, se denota $\rightarrow_\beta$ y al igual que el cambio de variable ligada, también se define en base a la operación de sustitución. Sea $(\lambda x.M)$ una abstracción cualquiera y N una expresión cualquiera, la reducción de $((\lambda x.M)N)$ se define como

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

Cuando la aplicación que se reduce es parte de otra expresión, se suele denotar la aplicación sobre la $\rightarrow_\beta$. Por ejemplo el procedimiento para reducir la expresión $(\lambda x.((\lambda w.w)\,x))$ a $(\lambda x.x)$ es

$$(\lambda x.((\lambda w.w)\,x)) \overset{((\lambda \mathrm{w.w})\,\mathrm{x})}{\rightarrow_\beta} (\lambda x.w\,[w:=x]) = (\lambda x.x)$$

En la mayoría de los casos, la aplicación reducida se puede inferir observando la transformación de expresiones, por lo tanto esta especificidad será omitida en este trabajo.

Las operaciones de sustitución, cambio de variable ligada y reducción se pueden aplicar en varios pasos de un proceso de transformación sin especificar qué operación en particular se realiza, denotando el paso de una expresión a otra con el símbolo $\rightarrow$. Por ejemplo la expresión $((\lambda y.(\lambda x.y))\,x)$ se puede transformar a $(\lambda z.x)$ siguendo los siguientes pasos:

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

El cálculo λ es un sistema maleable y se permite definir operaciones arbitrarias sobre expresiones para estudiar cómo 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 uso principalmente de las operaciones de *sustitución*, *cambio de variable ligada* y *reducción de aplicaciones*.

1.4. Equivalencias

El cálculo λ se considera formalmente como una *teoría ecuacional*, esto significa que los axiomas de su teoría 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 aso-

ciar 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 átomo usado para representar una variable vinculada y las variables enlazadas a ella.

Considerando la expresión análoga a la función identidad $(\lambda x.x)$ se observa que tiene la misma estructura que $(\lambda y.y)$ 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.

El criterio de equivalencia relacionado con la operación de cambio de variable ligada es llamada α-convertibilidad y se denota como

$$M =_\alpha N$$

para dos expresiones M y N en donde a partir de una cantidad finita de cambios de variable ligada en M o parte de M se pueda obtener N.

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. Por ejemplo, consideremos la expresión

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

utilizando índices de De Bruijn se escribe 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 un criterio de equivalencia entre expresiones. La idea básica de este criterio 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 N puede ser obtenida a partir de M por una cantidad finita de reducciones de aplicaciones, reducciones inversas y cambios de variable ligada.

La reducción inversa es la transformación de un término de la forma $M[x:=N]$ al término $((\lambda x.M)N)$, de tal manera que si existe una reducción inversa de P a P' entonces P' se reduce a P. Este proceso de transformación inverso a la reducción de aplicaciones se puede ilustrar considerando que

$$x \not\to_\beta ((\lambda y.y)x)$$

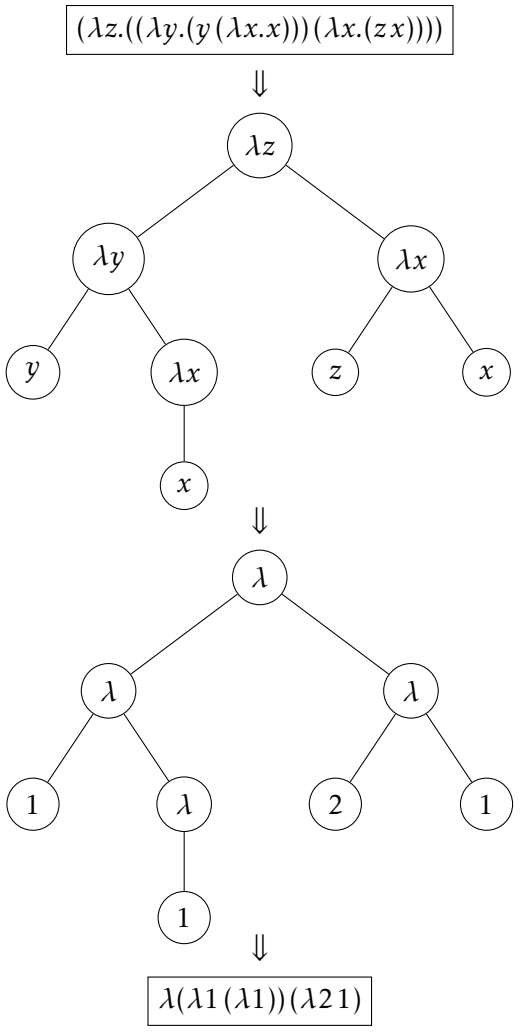


Figura 1.1: Transformación gráfica para notación de De Bruijn

pero ya que

$$((\lambda y.y)\,x) \rightarrow_\beta x$$

hay una reducción inversa de x a $((\lambda y.y)\,x)$.

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

1. Toda expresión M es equivalente a sí misma, es decir, $M \sim M$.
2. Si una expresión M es relacionada con una equivalencia a otra expresión N, entonces N también es relacionada a M, es decir $M \sim N \implies N \sim M$.
3. Si una expresión M se relaciona con una equivalencia a otra expresión N y N se relacio-

na con la misma equivalencia a P, entonces, M y P se relacionan con esta equivalencia, es decir, $M \sim N,\ N \sim P \implies M \sim P$.

La equivalencia sintáctica corresponde a la propiedad 1 la cuál es llamada *reflexividad*; al igual que la α-conversión y la β-conversión, la equivalencia sintáctica no está asociada a una regla de inferencia. En las propiedades 2 y 3 se tienen inferencias que parten de expresiones equivalentes y basado en si estas expresiones son equivalentes o no, ciertas propiedades se deben cumplir. La propiedad 2 es llamada *simetría*, mientras que la propiedad 3 es llamada *transitividad*.

La α-conversión y la β-conversión fueron definidas como criterios de equivalencia por separado 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 propiedades que la equivalencia deba de cumplir.

Al igual que Haskell Curry en [14, p. 59] se utilizan las letras griegas α y β para referirse a las ecuaciones relacionadas con el cambio de variable ligada y la reducción de aplicaciones, mientras que las letras ρ, σ y τ se refieren a las ecuaciones relacionadas con las propiedades de reflexividad, simetría y transitividad respectivamente, se retoma esta convención para elaborar la siguiente definición de una relación de equivalencia representada con el símbolo de igualdad =

$$(\lambda x.M) = (\lambda y.M[x := y]) \tag{α}$$

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

$$M = M \tag{ρ}$$

$$M = N \implies N = M \tag{σ}$$

$$M = N,\ N = P \implies M = P \tag{τ}$$

Estas ecuaciones resultan describir una equivalencia muy parecida a la β-conversión, una manera de verificar esto es analizando las expresiones que se consideran equivalentes en una equivalencia se consideran equivalentes en la otra:

- Si $M \to_\alpha N$ entonces $M = N$ por la ecuación (α)
- Si $M \to_\beta N$ entonces $M = N$ por la ecuación (β)
- Si $M \equiv N$ entonces $M = N$ por la ecuación (ρ)
- Ya que $\to_\alpha$ y $\equiv$ son simétricas y la β-conversión contempla reducciones inversas: Si $M =_\beta N$, entonces $M = N$ por la ecuación (σ)
- Ya que la β-conversión contempla una cantidad finita de transformaciones: Si $M =_\beta N$, entonces $M = N$ por la ecuación (τ)

Sin embargo $M = N$ no implica que $M =_\beta N$ ya que la β-conversión contempla la transformación de partes de una expresión y = no. Para ilustrar esta diferencia consideremos la convertibilidad

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

La secuencia detallada de transformaciones realizadas para pasar de la expresión de la izquierda a la de la derecha es

$$(\lambda f.((\lambda x.(f\,x))\,y)) \overset{((\lambda x.(f\,x))\,y)}{\underset{\beta}{\rightarrow}} (\lambda f.(f\,x)[x:=y]) \rightarrow (\lambda f.(f\,y))$$

En el primer paso se reduce una aplicación interna a la expresión, sin embargo, ni una ecuación de = establece una equivalencia entre la expresión original y el resultado de la primer transformación.

Para capturar la definición de β-convertibilidad con ecuaciones, es necesario considerar otras ecuaciones en =. Las siguientes reglas, nombradas por Curry [14, p. 59] como ν, μ y ξ, junto con las anteriores reglas de = completan la definición declarativa de β-convertibilidad:

$$M = N \implies (M\,Z) = (N\,Z) \qquad (\nu)$$

$$M = N \implies (Z\,M) = (Z\,N) \qquad (\mu)$$

$$M = N \implies (\lambda x.M) = (\lambda x.N) \qquad (\xi)$$

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

$$((\lambda x.(f\,x))\,y) =_\beta (f\,y) \qquad \text{por } \beta \qquad (1.3)$$

$$(\lambda f.((\lambda x.(f\,x))\,y)) =_\beta (\lambda f.(f\,y)) \qquad \text{por (1.3)} \qquad (1.4)$$

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 como ecuación de = de la siguiente manera:

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

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 [10].

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 primera 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 tercera 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 [15] 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, 10].

2.1. Términos λ

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 [17].

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. En estas definiciones formales es común encontrar diferentes tipos de "flechas" que denotan diferentes cosas, en este trabajo se utiliza la flecha $\Longrightarrow$ para denotar una implicación lógica, $\longrightarrow$ para denotar una producción en una gramática y $\Rightarrow$ para denotar 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 numerable $\{v_0,\ v_{00},\ v_{000},\ ...\}$ de variables. Λ es el conjunto más pequeño que satisface:

$$x \in V \implies x \in \Lambda \tag{2.1a}$$

$$M \in \Lambda,\ x \in V \implies (\lambda x.M) \in \Lambda \tag{2.1b}$$

$$M,\ N \in \Lambda \implies (M\,N) \in \Lambda \tag{2.1c}$$

Cada parte de esta definición corresponde a las tres clases de término λ:

(2.1a) establece que todo elemento de V es un término λ a los cuales se les llama *átomos*;
(2.1b) 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*;
(2.1c) establece que las cadenas de la forma $(M\,N)$ 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 [17], 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 las *categorías sintácticas* T, E, F y A, las cuales denotan las reglas para derivar términos λ, aplicaciones, abstracciones y átomos respectivamente; los *símbolos terminales* $\{(,\ ),\ .,\ \lambda,\ v,\ _0\}$, los cuales son los símbolos que conforman a las cadenas en Λ; el *símbolo inicial* T, el cual es el símbolo del que se derivan todos los términos

λ. Las reglas de producción de G son:

$$T \to E \mid F \mid A \tag{2.2a}$$
$$A \to v_0 \mid A_0 \tag{2.2b}$$
$$F \to (\lambda\ A\ .\ T\ ) \tag{2.2c}$$
$$E \to (\ T\ T\ ) \tag{2.2d}$$

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 de la Definición 2.1.2 se debe presentar una derivación de la cadena a partir de la categoría sintáctica T.

A continuación se considera un ejemplo en donde una cadena de símbolos es elemento de Λ. Sea $M = (\lambda v_0.(v_{00}(\lambda v_{00}.v_{000})))$, la cadena M es un término λ ya que

Por definición inductiva

$$\begin{aligned}
v_{000} \in V &\implies v_{000} \in \Lambda;\\
v_{00} \in V,\ v_{000} \in \Lambda &\implies (\lambda v_{00}.v_{000}) \in \Lambda,\ v_{00} \in \Lambda;\\
v_{00},\ (\lambda v_{00}.v_{000}) \in \Lambda &\implies (v_{00}(\lambda v_{00}.v_{000})) \in \Lambda;\\
v_0 \in V,\ (v_{00}(\lambda v_{00}.v_{000})) \in \Lambda &\implies (\lambda v_0.(v_{00}(\lambda v_{00}.v_{000}))).
\end{aligned}$$

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.

$$\begin{aligned}
T &\Rightarrow F \Rightarrow (\ \lambda\ A\ .\ T\ ) \Rightarrow (\ \lambda\ v_0\ .\ T\ ) \Rightarrow (\ \lambda\ v_0\ .\ E\ ) \Rightarrow (\ \lambda\ v_0\ .\ (\ T\ T\ )\ )\\
&\Rightarrow (\ \lambda\ v_0\ .\ (\ A\ T\ )\ ) \Rightarrow (\ \lambda\ v_0\ .\ (\ A_0\ T\ )\ ) \Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ T\ )\ )\\
&\Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ F\ )\ ) \Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ (\ \lambda\ A\ .\ T\ )\ )\ )\\
&\Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ (\ \lambda\ A_0\ .\ T\ )\ )\ ) \Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ (\ \lambda\ v_{00}\ .\ T\ )\ )\ )\\
&\Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ (\ \lambda\ v_{00}\ .\ A\ )\ )\ ) \Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ (\ \lambda\ v_{00}\ .\ A_0\ )\ )\ )\\
&\Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ (\ \lambda\ v_{00}\ .\ A_{00}\ )\ )\ ) \Rightarrow (\ \lambda\ v_0\ .\ (\ v_{00}\ (\ \lambda\ v_{00}\ .\ v_{000}\ )\ )\ ).
\end{aligned}$$

Al considerar una cadena que no es elemento de Λ se sigue un procedimiento similar. Sea $N = ((\lambda v_{00}.v_0\, v_{00})v_0)$, 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

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

Por gramática Realizando una derivación por la izquierda se debería de poder derivar por completo a N, sin embargo esto no es posible.

$$T \Rightarrow E \Rightarrow (\ T\ T\ ) \Rightarrow (\ F\ T\ ) \Rightarrow (\ (\ \lambda\ A\ .\ T\ )\ T\ )$$
$$\Rightarrow (\ (\ \lambda\ A_0\ .\ T\ )\ T\ ) \Rightarrow (\ (\ \lambda\ v_{00}\ .\ T\ )\ T\ )$$
$$\not\Rightarrow (\ (\ \lambda\ A\ .\ v_0\, v_{00}\ )\ T\ ).$$

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 vinculadas. Éste abuso de notación es consistente con la representación de funciones de varias variables usada por Schönfinkel [26].

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

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

y la consideración de 4*c* como

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

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:

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

2.1.1. Estructura

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

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

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

Debido a que la definición considera la cantidad de átomos en M y la longitud de un átomo es 1, se infiere que para cualquier término M, 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 estríctamente menor a su longitud.

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 M, los casos base de la inducción son aquellos en donde la estructura no es compuesta (en átomos cuya longitud

siempre es 1) y la hipótesis de inducción considera que P se cumple para los subtérminos de M cuya longitud siempre es estrictamente menor que $\|M\|$.

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

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

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

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

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

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

Considerando el término $M \equiv \lambda x.x\,y\,(\lambda z.y)$, lo siguiente se cumple:

- el término $(x\,y) \subset M$;
- el átomo $z \not\subset M$ pero si aparece en M, debido a que z acompaña a una λ;
- el término $y(\lambda z.y)$ a pesar de parecer ser un subtérmino de M no lo es, esto se puede corroborar escribiendo los términos sin el abuso de notación: $y\,(\lambda z.y) \equiv (y\,(\lambda z.y))$ y $M \equiv \lambda x.x\,y\,(\lambda z.y) \equiv (\lambda x.((x\,y)\,(\lambda z.y)))$, 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 del 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 vinculada* si y sólo si es la x que acompaña la λ de $(\lambda x.M)$ en P;
- *variable libre* en otro caso.

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

Cuando se abordó el concepto de reducción en la Subsección 1.3.3 la distinción entre una variable libre y una ligada era importante ya que las variables libres nunca son sustituídas en una reducción ya que el procedimiento relacionaba únicamente a las variables ligadas en el alcance de una abstracción activa. Considerando el término $M \equiv x(\lambda y.xy)$ lo siguiente se cumple:

- 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 vinculada 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 M se denota $\mathrm{FV}(M)$ y se define de manera inductiva como:

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

Cuando $\mathrm{FV}(M) = \emptyset$ se dice que M es un *combinador* o *término cerrado*. Por ejemplo, considerando los términos $(x(\lambda x.xyz))$, $(\lambda xyz.y)$ y $((\lambda y.x)\lambda x.y)$ lo siguiente se cumple:

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

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

En ocaciones es importante distinguir los términos 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:

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

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)**.** Sea M un término λ tal que $\mathrm{FV}(M) \neq \emptyset$. La clausura de M es un término

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

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

Un término puede tener varias clausuras, por ejemplo, considerando que

$$M \equiv \lambda z.x\,y\,z$$

Dos posibles clausuras de M son $(\lambda x\,y.\lambda z.x\,y\,z)$ y $(\lambda y\,x\,z.x\,y\,z)$, mientras que $(\lambda z\,x\,y.\lambda z.x\,y\,z)$, a pesar de no tener variables libres, no es clausura de M debido a la primera abstracción con variable vinculada z.

2.1.3. Sustitución de términos

En la Subsección 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 M, N y x, se define

$$M[x := N]$$

como el resultado de sustituir cada aparición libre de x por N en M de acuerdo a las siguientes reglas:

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

A continuación se muestran procedimientos de sustituciones para cada uno de los casos de la Definición 2.1.10.

- Caso $x[x := N]$

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

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

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

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

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

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

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

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

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

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

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

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

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

  En este 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 $y \notin \mathrm{FV}(M)$, de otro modo, M tiene variables sustituíbles y que no serán sustituídas con $M[x := P]$.
- $M[x := y][y := x]$
  Esta operación es equivalente a M siempre y cuando $y \notin \mathrm{FV}(M)$, 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 $y \notin \mathrm{FV}(P)$ y $x \notin \mathrm{FV}(Q)$ 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* $(y\,x) \notin \mathrm{FV}(L)$ $y\ x \not\equiv y$, *entonces*

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

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

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 huecos de $C[\ \ ]$. Al realizar esto, las variables libres de M pueden convertirse en variables ligadas de $C[M]$. Consideremos el contexto $C[\ \ ] \equiv \lambda x.x\,\lambda y.[\ \ ]$ y el término $M \equiv (xy)$.

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

El caso análogo con la sustitución es

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

2.2. Los cálculos de la conversión 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 $\boldsymbol{\lambda K \beta}$, se pueden realizar modificaciones y extensiones a esta teoría y los siguientes conceptos permiten estudiar las implicaciones de estos cambios.

2.2.1. Teorías formales

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

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

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

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

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

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

Observación. En la literatura se pueden encuentrar diferentes maneras de trabajar con teorías formales, dependiendo de su estilo y definición, por ejemplo en [1] las reglas se definen como conjuntos de secuencias $\langle A_1, \ldots, A_{n+1}\rangle$ con n premisas y una conclusión, en donde los axiomas se definen como elementos de $\mathcal{R}$ con cero premisas. La definición de teoría formal presentada en este trabajo es del estilo Hilbert y está basada en [15, 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. Se dice que B es *demostrable* en $\mathcal{T}$ suponiendo Γ si existe una deducción para una fórmula B, esto se denota

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

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

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

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

En el contexto de las teorias que formalizan los cálculos 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 $\boldsymbol{\lambda}$ 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 $\boldsymbol{\lambda}$ como equivalentes. Cuando $\boldsymbol{\lambda} \vdash M = N$ se dice que M y N son términos *convertibles*, también denotado $M =_{\lambda} N$.

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

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

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

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

La modificación al lenguaje formal puede ser únicamente de relevancia sintáctica, por ejemplo modificar una teoría $\boldsymbol{\lambda}$ cuyo lenguaje de términos es el conjunto Λ^0 para que los términos sean escritos con el índice de DeBruijn no tendría implicaciones fuertes en la convertibilidad de la teoría, ya que hay una correspondencia uno a uno entre estas dos notaciones. Por otro lado, modificar una teoría $\boldsymbol{\lambda}$ con términos Λ de tal manera que se consideren únicamente los términos cerrados Λ^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 $\boldsymbol{\lambda}$ es añadir o quitar axiomas y reglas de inferencia. Para abordar la modificación de estas dos componentes de una teoría consideramos que los axiomas son reglas sin premisas.

Cuando se considera extender una teoría $\boldsymbol{\lambda}$ con una nueva regla ϕ lo primero que se debe estudiar es si ϕ es *derivable* en $\boldsymbol{\lambda}$, es decir, si para cada instancia de ϕ, su conclusión es deducible en $\boldsymbol{\lambda}$ considerando sus premisas como suposiciones. Formalmente, para cada

instancia $\langle A_1, \ldots, A_n, B\rangle$ de ϕ, ϕ es derivable en $\boldsymbol{\lambda}$ si y sólo si

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

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

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

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

Si una regla es derivable, entonces es admisible, sin embargo, una regla admisible no siempre es derivable. Consideremos una instancia r de una regla admisible en $\boldsymbol{\lambda}$ 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 $\boldsymbol{\lambda}$, 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 $\boldsymbol{\lambda}$ y $\boldsymbol{\lambda}'$ dos teorías formales con el mismo conjunto de fórmulas.

- Se dice que las teorías son *equivalentes en teoremas* cuando cada regla y axioma de $\boldsymbol{\lambda}$ es admisible en $\boldsymbol{\lambda}'$ y viceversa.
- Se dice que las teorías son *equivalentes en reglas* cuando cada regla y axioma de $\boldsymbol{\lambda}$ es derivable en $\boldsymbol{\lambda}'$ y viceversa.

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

2.2.2. Teoría $\boldsymbol{\lambda K\beta}$

La teoría $\boldsymbol{\lambda K\beta}$ 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 $\boldsymbol{\lambda}$ se estará refiriendo a la teoría $\boldsymbol{\lambda K\beta}$ y cuando se hable de *las* teorías $\boldsymbol{\lambda}$ se estará refiriendo a la familia de teorías que formalicen los cálculos λ.

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

ecuaciones de la forma:

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

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

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

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

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

Las reglas $\mathscr{R}$ de $\boldsymbol{\lambda K \beta}$ 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 $\boldsymbol{\lambda K \beta}$ de los siguientes términos

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

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

La teoría $\boldsymbol{\lambda}$ 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 procedimientos y procesos que describen los programas o algoritmos, cuyas igualdades son *intensionales*, es decir, si dos programas computan la misma función matemática, no nece-

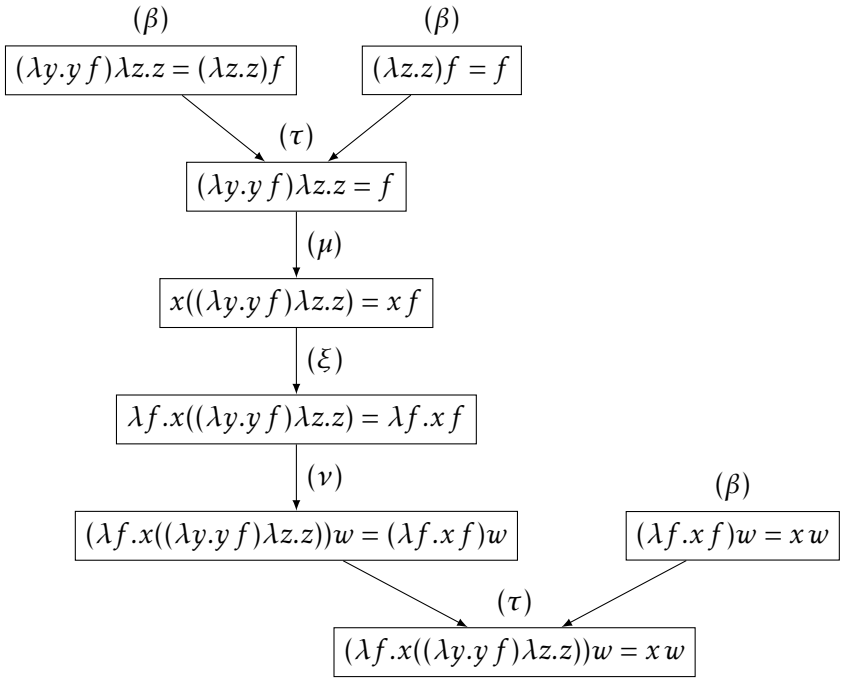


Figura 2.1: Árbol de deducción para demostrar la convertibilidad entre dos términos

sariamente 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 $\boldsymbol{\lambda}$ también es intensional ya existen dos términos F 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 $\boldsymbol{\lambda}$?. A continuación se presentan tres diferentes agregados a la teoría $\boldsymbol{\lambda}$ las cuales incluyen la propiedad de extensionalidad y que han sido propuestas en la literatura [15, 5]. Las teorías extendidas son llamadas $\boldsymbol{\lambda\zeta}$, $\boldsymbol{\lambda + ext}$ y $\boldsymbol{\lambda\eta}$ de acuerdo a la regla que se añade a la Definición 2.2.2.

Definición 2.2.3 (Reglas de extensionalidad)**.** Cada una de las siguientes reglas nos permite añadir a $\boldsymbol{\lambda}$ la propiedad de extensionalidad.

Reglas de inferencia

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

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

Axiomas

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

Hindley y Seldin en [15, 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 $\boldsymbol{\lambda}$:

- La teoría $\boldsymbol{\lambda\zeta}$ es igual que $\boldsymbol{\lambda}$ pero incluyendo también la regla ζ;
- La teoría $\boldsymbol{\lambda\eta}$ es igual que $\boldsymbol{\lambda}$ 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 $\boldsymbol{\lambda\zeta}$ y $\boldsymbol{\lambda\eta}$ son equivalentes en reglas.*

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

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

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

1. Se parte de la primicia $(M\,x) =_{\lambda\eta} (N\,x)$ si $x \notin \mathrm{FV}(M)$;
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 [15, 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 grán cantidad de definiciones, teoremas y extensiones.

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

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 $\{v_0,\ v_{00},\ v_{000},\ ...\}$ y C el conjunto de constantes atómicas $\{\boldsymbol{I},\ \boldsymbol{K},\ \boldsymbol{S}\}$. El conjunto de términos de la lógica combinatoria, denotado CL se define de manera inductiva de la siguiente manera:

$$x \in V \implies x \in CL \tag{a}$$

$$x \in C \implies x \in CL \tag{b}$$

$$X,\ Y \in CL \implies (X\,Y) \in CL \tag{c}$$

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 $\boldsymbol{CL\omega}$)**.** El conjunto de fórmulas $\mathcal{F}$ en $\boldsymbol{CL\omega}$ tiene como elementos ecuaciones de la forma:

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

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

$$\boldsymbol{I}\,X = X \tag{$\boldsymbol{I}$}$$

$$\boldsymbol{K}\,X\,Y = X \tag{$\boldsymbol{K}$}$$

$$\boldsymbol{S}\,X\,Y\,Z = X\,Z(Y\,Z) \tag{$\boldsymbol{S}$}$$

$$X = X \tag{ρ}$$

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

$$\frac{X = X'}{Z\,X = Z\,X'} \tag{μ}$$

$$\frac{X = X'}{X\,Z = X'\,Z} \tag{ν}$$

$$\frac{X = Y \quad Y = Z}{X = Z} \tag{τ}$$

$$\frac{X = Y}{Y = X} \tag{σ}$$

Existen equivalencias entre modificaciones de teorías del cálculo λ y la lógica combinatoria [15, pp. 92–106], pero en este trabajo se muestra una introducción a las similitudes entre $\boldsymbol{\lambda}$ y $\boldsymbol{CL\omega}$, en particular, la correspondencia entre las clases de términos de $\boldsymbol{CL\omega}$ que se pueden representar en $\boldsymbol{\lambda}$ y viceversa.

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

Definición 2.2.6 (Constantes $\boldsymbol{I}$, $\boldsymbol{K}$ y $\boldsymbol{S}$ en Λ)**.**

$$\begin{aligned} \boldsymbol{I} &\equiv \lambda x.x \\ \boldsymbol{K} &\equiv \lambda xy.x \\ \boldsymbol{S} &\equiv \lambda xyz.xz(yz) \end{aligned}$$

Se corroboran estas definiciones considerando a M, N y Z en Λ utilizando el axioma (β) de $\boldsymbol{\lambda}$.

$$\begin{aligned} \boldsymbol{I}M &=_{\lambda} (\lambda x.x)M \\ &=_{\lambda} M \\ \boldsymbol{K}MN &=_{\lambda} (\lambda xy.x)MN \\ &=_{\lambda} (\lambda y.M)N \\ &=_{\lambda} M \\ \boldsymbol{S}MNZ &=_{\lambda} (\lambda xyz.xz(yz))MNZ \\ &=_{\lambda} (\lambda yz.Mz(yz)) \\ &=_{\lambda} (\lambda z.Mz(Nz)) \\ &=_{\lambda} MZ(NZ) \end{aligned}$$

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 $\boldsymbol{CL\omega}$ tienen sus contrapartes en $\boldsymbol{\lambda}$. Por lo tanto, todo término en CL puede ser representado con términos en Λ y toda deducción realizada en $\boldsymbol{CL\omega}$ puede también realizarse en $\boldsymbol{\lambda}$ considerando la Definición 2.2.6.

Por otra parte, la lógica combinatoria puede representar gran parte del cálculo λ, pero no todo. En particular la regla (ξ) en $\boldsymbol{\lambda}$ no es admisible en la teoría $\boldsymbol{CL\omega}$ [15, p. 92].

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

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

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

   b) $M, N \in \mathfrak{X}^+ \implies (MN) \in \mathfrak{X}^+$.
2. Sea $\mathcal{P}, \mathfrak{X} \subset \Lambda$. $\mathfrak{X}$ es una *base* para $\mathcal{P}$ si para toda $M \in \mathcal{P}$ existe $N \in \mathfrak{X}^+$ tal que $N = M$.
3. $\mathfrak{X}$ es llamada una *base* si $\mathfrak{X}$ es una base para Λ^0.

Proposición 2.2.1. *El conjunto* $\{\boldsymbol{S}, \boldsymbol{K}, \boldsymbol{I}\}$ *es una base, es decir, para todo término* $M \in \Lambda^0$*, existe un término* M' *compuesto de aplicaciones de* $\boldsymbol{S}$, $\boldsymbol{K}$ *e* $\boldsymbol{I}$ *tal que* $M =_\lambda M'$.

La demostración de la Proposición 2.2.1 consiste en la construcción de un algoritmo para transformar M a M'. El siguiente lema es utilizado en esa demostración:

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

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

Demostración. Utilizando la Definición 2.2.6

$$
\begin{aligned}
\boldsymbol{I}\, N &=_\lambda N && (1)\\
&=_\lambda (\lambda x.x) N\\
&=_\lambda (\lambda x.M) N\\
\boldsymbol{K}\, M\, N &=_\lambda (\lambda x\, y.x)\, M\, N && (2)\\
&=_\lambda (\lambda y.M) N\\
&=_\lambda (\lambda x.M) N\\
\boldsymbol{S}\, (\lambda x.P)\, (\lambda x.Q) &=_\lambda (\lambda a\, b\, c.(a\, c)\, (b\, c))\, (\lambda x.P)\, (\lambda x.Q) && (3)\\
&=_\lambda \lambda c.((\lambda x.P)\, c)\, ((\lambda x.Q)\, c)\\
&=_\lambda \lambda c.P[x := c] Q[x := c]\\
&=_\lambda \lambda c.(P\, Q)[x := c]\\
&=_\lambda \lambda x.P\, Q\\
&=_\lambda \lambda x.M
\end{aligned}
$$

□

Demostración de la Proposición 2.2.1. Sea $M \in \Lambda^0$, se construye un término $M' \in \{\boldsymbol{S}, \boldsymbol{K}, \boldsymbol{I}\}^+$ tal que $M' =_\lambda M$ con el Algoritmo 2.1.

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 $\boldsymbol{I}$ en M.

Algoritmo 2.1 SKI

Entrada: $M \in \Lambda^0$
Salida: $M' \in \{\boldsymbol{S}, \boldsymbol{K}, \boldsymbol{I}\}$

```
M' ← M
𝓛 ← {A ∈ Sub(M') | A ≡ λx.N}
mientras 𝓛 ≠ ∅ hacer
  A ← λx.N ∈ 𝓛 | ||λx.N|| ≤ A', ∀A' ∈ 𝓛
  si A ≡ λx.a entonces
    si a ≡ x entonces
      M'[A] ← I
    de lo contrario y si a ≢ x entonces
      M'[A] ← K a
    fin si
  de lo contrario y si A ≡ λx.P Q entonces
    si x ∉ FV(P) ∧ x ∉ FV(Q) entonces
      M'[A] ← S (K P) (K Q)
    de lo contrario y si x ∉ FV(P) ∧ x ∈ FV(Q) entonces
      M'[A] ← S (K P) (λx.Q)
    de lo contrario y si x ∈ FV(P) ∧ x ∉ FV(Q) entonces
      M'[A] ← S (λx.P) (K Q)
    de lo contrario y si x ∈ FV(P) ∧ x ∈ FV(Q) entonces
      M'[A] ← S (λx.P) (λx.Q)
    fin si
    M'[A] ← SKI( M'[A] )
  fin si
  𝓛 ← 𝓛 \ {A}
fin mientras
regresar M'
```

b) Si $a \neq x$ se transforma $\lambda x.N$ a $(\boldsymbol{K}\, a)$ en M.

2. $N = (P\, Q)$
   a) Si $x \notin \mathrm{FV}(P)$ y $x \notin \mathrm{FV}(Q)$ se transforma $\lambda x.P\, Q$ a $\boldsymbol{S}\,(\boldsymbol{K}\, P)\,(\boldsymbol{K}\, Q)$ en M.
   b) Si $x \notin \mathrm{FV}(P)$ y $x \in \mathrm{FV}(Q)$ se transforma $\lambda x.P\, Q$ a $\boldsymbol{S}\,(\boldsymbol{K}\, P)\,(\lambda x.Q)$ en M.
   c) Si $x \in \mathrm{FV}(P)$ y $x \notin \mathrm{FV}(Q)$ se transforma $\lambda x.P\, Q$ a $\boldsymbol{S}\,(\lambda x.P)\,(\boldsymbol{K}\, Q)$ en M.
   d) Si $x \in \mathrm{FV}(P)$ y $x \in \mathrm{FV}(Q)$ se transforma $\lambda x.P\, Q$ a $\boldsymbol{S}\,(\lambda x.P)\,(\lambda x.Q)$ 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 vinculadas) 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 $a' \colon \mathbb{N} \to \mathbb{N}$:

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

Para la cota superior de la cantidad de pasos realizados por el algoritmo para un término M con n abstracciones, se plantea la siguiente relación de recurrencia basada en a' y en el hecho de que la cantidad de variables ligadas y no vinculadas 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 $p \colon \mathbb{N} \to \mathbb{N}$:

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

□

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 $\boldsymbol{\lambda}$. Esto pudiera parecer un resultado meramente teórico, sin embargo este algoritmo de transformación es similar al procedimiento que realizaría un compilador del cálculo λ a la lógica combinatoria, y ya que la lógica combinatoria consiste básicamente de aplicaciones

y tres "instrucciones" $\boldsymbol{S}$, $\boldsymbol{K}$ e $\boldsymbol{I}$ es más sencillo construír una máquina (real o virtual) que realice las reducciones. En el artículo "A New Implementation Technique for Applicative Languages" de David Turner [36] se describe una máquina que executa de manera eficiente una versión extendida de la lógica combinatoria.

2.3. Teoría de reducción

El concepto de reducción fue introducido en la Subsección 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 esta sección se desarrolla un complemento al estudio del cálculo λ de la Sección 2.2. Primero se aborda la reducción en general, independiente de la regla (β) como es 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.

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 $\boldsymbol{R}$ 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 $\boldsymbol{\lambda}$. 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 $\boldsymbol{R}$ sobre Λ las reglas (μ), (ν) y (ξ) de la teoría $\boldsymbol{\lambda}$ se

dice que $\boldsymbol{R}$ es *compatible*. Formalmente, una relación es compatible cuando

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

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 $\boldsymbol{\lambda}$, es decir

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

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 $\boldsymbol{\lambda}$, se dice que es una *relación de igualdad* o *congruencia*, debido a que además de ser compatible, es simétrica, reflexiva y transitiva. Formalmente una relacion $\boldsymbol{R}$ sobre Λ es de congruencia cuando es una relación de reducción y además

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

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 $\boldsymbol{\beta}$ se define como

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

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

- La clausura reflexiva de $\boldsymbol{R}$ es la relación más pequeña que extiende a $\boldsymbol{R}$ para que sea reflexiva;
- La clausura transitiva de $\boldsymbol{R}$ es la relación más pequeña que extiende a $\boldsymbol{R}$ para que sea transitiva;
- La clausura compatible de $\boldsymbol{R}$ es la relación más pequeña que extiende a $\boldsymbol{R}$ 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 $\boldsymbol{R}$ una noción de reducción sobre Λ. $\boldsymbol{R}$ introduce las siguientes relaciones binarias

- R-contracción, denotada $\to_{\mathrm{R}}$, es la clausura compatible de $\boldsymbol{R}$:

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

- R-reducción, denotada $\twoheadrightarrow_{\mathrm{R}}$, es la clausura reflexiva y transitiva de $\to_{\mathrm{R}}$:

$$\begin{aligned}M \to_{\mathrm{R}} N &\implies M \twoheadrightarrow_{\mathrm{R}} N\\ M &\twoheadrightarrow_{\mathrm{R}} M\\ M \twoheadrightarrow_{\mathrm{R}} N,\ N \twoheadrightarrow_{\mathrm{R}} L &\implies M \twoheadrightarrow_{\mathrm{R}} L\end{aligned}$$

- R-convertibilidad, denotada $=_{\mathrm{R}}$, es la relación de equivalencia generada por $\twoheadrightarrow_{\mathrm{R}}$:

$$\begin{aligned}M \twoheadrightarrow_{\mathrm{R}} N &\implies M =_{\mathrm{R}} N\\ M =_{\mathrm{R}} N &\implies N =_{\mathrm{R}} M\\ M =_{\mathrm{R}} N,\ N =_{\mathrm{R}} L &\implies M =_{\mathrm{R}} L\end{aligned}$$

Observación (Notación). Las nociones de reducción serán denotadas con letras en negritas, por ejemplo $\boldsymbol{\beta}$ y sus relaciones derivadas omitirán esta convención, por ejemplo $\to_\beta$, $\twoheadrightarrow_\beta$, $=_\beta$.

Considerando la noción de reducción $\boldsymbol{\beta}$ se cumple lo siguiente

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

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 $\boldsymbol{\alpha}$ es

$$\boldsymbol{\alpha} = \{(\lambda x.M,\ \lambda y.M[x:=y]) \mid M \in \Lambda,\ y \notin \mathrm{FV}(M)\}$$

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 $\boldsymbol{\beta}$ no considera cambios de variables ligadas entre términos, sin embargo, la noción de reducción $\boldsymbol{\alpha\beta} = \boldsymbol{\alpha} \cup \boldsymbol{\beta}$ sí. De hecho, $=_{\boldsymbol{\alpha\beta}}$ tiene las mismas propiedades que la convertibilidad $=_{\boldsymbol{\lambda}}$ de la teoría formal $\boldsymbol{\lambda}$.También se puede definir una noción de reducción para el axioma de extensionalidad de la teoría $\boldsymbol{\lambda\eta}$ y estudiar las transformaciones en $\boldsymbol{\alpha} \cup \boldsymbol{\beta} \cup \boldsymbol{\eta}$ es equivalente a estudiar la demostrabilidad en la teoría $\boldsymbol{\lambda\eta}$ donde

$$\boldsymbol{\eta} = \{(\lambda x.M\,x,\ M) \mid M \in \Lambda,\ x \notin \mathrm{FV}(M)\}$$

Proposición 2.3.1 (Equivalencia $\boldsymbol{\alpha\beta}$ y $\boldsymbol{\lambda}$)**.** *Para todo* $M,\ N \in \Lambda$

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

Demostración. Se demuestran las implicaciones por separado

($\Longleftarrow$) Ya que $\boldsymbol{\lambda} \vdash M = N$, existe un árbol de deducción que a partir de los axiomas y reglas de $\boldsymbol{\lambda}$ 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 $\boldsymbol{\alpha\beta}$ para pasar al siguiente paso:

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

($\Longrightarrow$) Por las definiciones de las relaciones que genera $\boldsymbol{\alpha\beta}$

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

□

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

Ya que las nociones de reducción son estudiadas como cómputos 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 $\boldsymbol{R}$:

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

- Un *R-redex* es un término M tal que $(M,N) \in \boldsymbol{R}$ para algún término N;
- Si $(M,N) \in \boldsymbol{R}$, 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 $M =_{\mathrm{R}} N$ 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". Considerando el término $M \equiv (\lambda x.x\,x)(\lambda y.y)z$

- $(\lambda x.x\,x)(\lambda y.y)$ es un β-redex de M;
- $(\lambda y.y)z$ no está en β-forma normal;
- M tiene la β-forma normal z.

Las nociones de reducción pueden visualizarse y estudiarse como pseudográficas dirigidas. Sea $\boldsymbol{R}$ una noción de reducción, la R-gráfica de un término M, denotada $G_R(M)$ es la pseudográfica con el conjunto de vértices $\{N \in \Lambda \mid M \twoheadrightarrow_{\mathrm{R}} N\}$ y conjunto de aristas $\rightarrow_{\mathrm{R}}$. Las siguientes gráficas de reducción son ejemplos de $\boldsymbol{\beta}$-gráficas y fueron obtenidas de [5, p. 57].

1. $G_\beta(\boldsymbol{I}\,x)$
   Este término es un redex el cual se contrae a x, por lo tanto se identifican dos vértices, el término original y el resultado de su contracción:

2. $G_\beta(\boldsymbol{I}\,(\boldsymbol{I}\,x))$
   Este término tiene dos redex: $\boldsymbol{I}\,(\boldsymbol{I}\,x)$ y $\boldsymbol{I}\,x$, la contracción de ambos resulta en el término $\boldsymbol{I}\,x$ el cuál a su vez se contrae a x, por lo tanto se identifican tres vértices, el término original, el término de la primer contracción y x, ya que hay dos maneras de pasar del primer vértice al segundo se tienen dos aristas:

3. $G_\beta((\lambda x.x\,x)(\lambda x.x\,x))$
   Este término es un redex y su contracción es él mismo, es decir

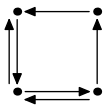
$$(\lambda x.x\,x)(\lambda x.x\,x) \rightarrow (\lambda x.x\,x)(\lambda x.x\,x)$$

por lo tanto se identifica un sólo vértice con un bucle.

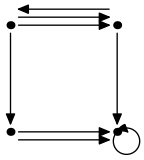
4. $G_\beta(\boldsymbol{W}\,\boldsymbol{W}\,\boldsymbol{W})$ donde $\boldsymbol{W} \equiv \lambda x y.x\,y\,y$
   Este término es bastante más complejo, para obtener su gráfica de reducción se van identificando los vértices a partir de todas las contracciones posibles:

$$\begin{aligned}
\boldsymbol{W}\,\boldsymbol{W}\,\boldsymbol{W} &\to (\lambda y.\boldsymbol{W}\,y\,y)\,\boldsymbol{W}\\
(\lambda y.\boldsymbol{W}\,y\,y)\,\boldsymbol{W} &\to \boldsymbol{W}\,\boldsymbol{W}\,\boldsymbol{W}\\
(\lambda y.\boldsymbol{W}\,y\,y)\,\boldsymbol{W} &\to (\lambda y.(\lambda z.y\,z\,z)\,y)\,\boldsymbol{W}\\
(\lambda y.(\lambda z.y\,z\,z)\,y)\,\boldsymbol{W} &\to (\lambda z.\boldsymbol{W}\,z\,z)\,\boldsymbol{W} =_\alpha (\lambda y.\boldsymbol{W}\,y\,y)\,\boldsymbol{W}\\
(\lambda y.(\lambda z.y\,z\,z)\,y)\,\boldsymbol{W} &\to (\lambda y.y\,y\,y)\,\boldsymbol{W}\\
(\lambda y.y\,y\,y)\,\boldsymbol{W} &\to \boldsymbol{W}\,\boldsymbol{W}\,\boldsymbol{W}
\end{aligned}$$

   La gráfica de reducción resultante es



5. $G_\beta(M\,M)$ con $M \equiv \lambda x.(\lambda y.y\,y)x$
   Este término también es complejo y se identifican los vértices y aristas de su gráfica de reducción igual al ejemplo anterior. La gráfica resultante debe ser



6. $G_\beta((\lambda x.x\,x\,x)(\lambda x.x\,x\,x))$
   Este término no tiene forma normal y es un redex. Sus contracciones siempre tendrán un sólo redex ya que

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

   de tal manera que su gráfica de reducción es

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

$$M_0 \overset{\Delta_0}{\to}_{\mathrm{R}} M_1 \overset{\Delta_1}{\to}_{\mathrm{R}} M_2 \to_{\mathrm{R}} \ldots$$

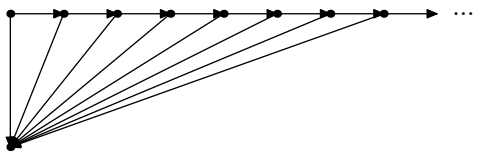
donde Δ_i es el R-redex de M_i que es contraído en cada paso de las R-contracciones. Sea M un término λ, se puede categorizar su R-reducción a partir de $G_R(M)$ y sus caminos con las siguientes definiciones

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.

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

$$M \equiv (\lambda x.\boldsymbol{I})((\lambda x.x\,x\,x)(\lambda x.x\,x\,x))$$

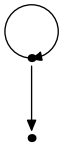
el cual tiene β-forma normal $\boldsymbol{I}$, al dibujar su β-gráfica se obtiene



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

$$M \equiv (\lambda x.\boldsymbol{I})((\lambda x.x\,x)(\lambda x.x\,x))$$

su β-forma normal es $\boldsymbol{I}$ y su β-gráfica es



Sin embargo, tiene el β-camino de reducción infinito

$$(\lambda x.\boldsymbol{I})((\lambda x.x\,x)(\lambda x.x\,x)) \overset{(\lambda x.x\,x)(\lambda x.x\,x)}{\underset{\beta}{\rightarrow}} (\lambda x.\boldsymbol{I})((\lambda x.x\,x)(\lambda x.x\,x)) \overset{(\lambda x.x\,x)(\lambda x.x\,x)}{\underset{\beta}{\rightarrow}} \ldots$$

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 $\boldsymbol{\beta}$ 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 ×. Las transformaciones que realiza este sistema de reducción son muy similares a las que realiza una calculadora simplificada, por ejemplo:

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

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

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

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:

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

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

$$\begin{aligned}f(g(0)) &\to g(0)\\ &\to 0\end{aligned}\qquad\qquad\begin{aligned}f(g(0)) &\to f(0)\\ &\to f(1)\\ &\to ...\end{aligned}$$

Por un lado se obtiene el resultado final 0 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:

$$f(1) \to f(0) \to f(1) \to ...$$

Al estudiar los sistemas de reducción desde un punto de vista computacional, indepen-

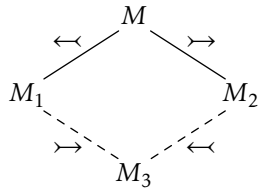
dientemente de su utilidad, el segundo ejemplo puede nunca llegar a una forma normal (no tener un resultado final) y además el orden en el que se apliquen las reducciones es importante para determinar el resultado. En el caso del primer ejemplo, ya que la suma y la multiplicación son conmutativas y el 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 $\rightarrowtail$ una relacion binaria sobre Λ. Se dice que $\rightarrowtail$ satisface la *propiedad de diamante*, si para toda M, M_1 y M_2

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

Gráficamente, esta propiedad se visualiza



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

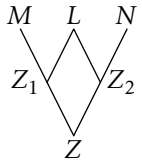
El siguiente teorema y corolario relacionan algunos conceptos abordados en la anterior sección con la propiedad de confluencia.

Teorema 2.3.1. *Sea $\boldsymbol{R}$ una noción CR. Entonces*

$$M =_{\mathrm{R}} N \implies \textit{existe } Z \textit{ tal que } M \twoheadrightarrow_{\mathrm{R}} Z \textit{ y } N \twoheadrightarrow_{\mathrm{R}} Z$$

Demostración. Por inducción de la definición de $=_{\mathrm{R}}$:

- Si $M =_{\mathrm{R}} N$ es una consecuencia directa de que $M \twoheadrightarrow_{\mathrm{R}} N$, entonces $Z \equiv N$.
- Si $M =_{\mathrm{R}} N$ es una consecuencia directa de que $N =_{\mathrm{R}} M$, Z puede ser encontrada por la hipótesis de inducción.
- Si $M =_{\mathrm{R}} N$ es una consecuencia directa de que $M =_{\mathrm{R}} L$ y $L =_{\mathrm{R}} N$, Z puede ser encontrada usando la hipótesis de inducción y la suposición de que $\twoheadrightarrow_{\mathrm{R}}$ es CR, gráficamente



□

Corolario. *Sea* $\boldsymbol{R}$ *una noción CR. Entonces*

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

Demostración. 1. Sea $M =_{\mathrm{R}} N$ y N un término en R-forma normal. Para algúna Z se tiene que $M \twoheadrightarrow_{\mathrm{R}} Z$ y $N \twoheadrightarrow_{\mathrm{R}} Z$, pero ya que N está en forma normal $Z \equiv N$. Por lo tanto $M \twoheadrightarrow_{\mathrm{R}} N$.

2. Supongamos que N_1 y N_2 son dos R-formas normales de M. Ya que $M \twoheadrightarrow_{\mathrm{N}_1}$ y $M \twoheadrightarrow_{\mathrm{N}_2}$, por la propiedad de confluencia, $N_1 \twoheadrightarrow_{\mathrm{R}} Z$ y $N_2 \twoheadrightarrow_{\mathrm{R}} Z$, 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 $\boldsymbol{R}$ es *sustitutiva* si para todo término M, N y L y toda variable x

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

Es importante observar que si una noción $\boldsymbol{R}$ es sustitutiva, entonces las relaciones generadas $\rightarrow_{\mathrm{R}}$, $\twoheadrightarrow_{\mathrm{R}}$ y $=_{\mathrm{R}}$ 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.

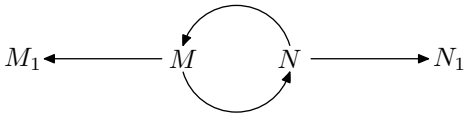
Definición 2.3.6 (Propiedad débil de diamante)**.** Una relación binaria $\rightarrowtail$ satisface la propiedad *débil* de diamante si para todo término M, M_1 y M_2

$$M \rightarrowtail M_1 \text{ y } M \rightarrowtail M_2 \implies \text{existe } M_3 \text{ tal que } M_1 \overset{*}{\underset{=}{\rightarrowtail}} M_3 \text{ y } M_2 \overset{*}{\underset{=}{\rightarrowtail}} M_3$$

donde $\overset{*}{\underset{=}{\rightarrowtail}}$ es la clausura transitiva y reflexiva de $\rightarrowtail$.

Definición 2.3.7 (Confluencia local)**.** Una noción de reducción ***R*** se dice ser Church-Rosser débil, abreviado WCR, si $\rightarrow_{\mathrm{R}}$ 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 $\rightarrow_{\mathrm{R}}$ tiene como clausura transitiva y reflexiva a $\twoheadrightarrow_{\mathrm{R}}$ y todo lo que se puede reducir con $\rightarrow_{\mathrm{R}}$ se puede reducir con $\twoheadrightarrow_{\mathrm{R}}$, 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 $M \rightarrow_{\mathrm{R}} M_1$ y $M \rightarrow_{\mathrm{R}} N$ y que también $M_1 \twoheadrightarrow_{\mathrm{R}} M_1$ y $N \twoheadrightarrow_{\mathrm{R}} M_1$, de manera similar comenzando desde M_1, N y N_1. Sin embargo, la noción no es CR ya que $M \twoheadrightarrow_{\mathrm{R}} M_1$ y $M \twoheadrightarrow_{\mathrm{R}} N_1$, pero no existe un término Z en donde $M_1 \twoheadrightarrow_{\mathrm{R}} Z$ y $N_1 \twoheadrightarrow_{\mathrm{R}} Z$.

Alonzo Church y Barkley Rosser en un artículo publicado en 1936 [11] 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 [15, 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 Algoritmo 2.2.

Algoritmo 2.2 Potencial ciclo infinito

```
x ← 4
si x ≠ 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].

En el estudio de lenguajes de programación, las estrategias de reducción son importantes debido a que se relacionan con las estrategias de evaluación de expresiones e instrucciones de los lenguajes. Para un tratamiento del tema extendido el libro "Types and Programming Languages" de Benjamin C. Pierce [23] provee un puente entre los conceptos del cálculo λ y la teoría de lenguajes de programación.

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

$$M \twoheadrightarrow_{\mathrm{R}} F(M)$$

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

$$M \to_{\mathrm{R}} F(M)$$

En este trabajo consideraremos tres estrategias de reducción para el cálculo λ muy utilizadas en la especificación de lenguajes de programación [23]:

- 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 (subrayados en el término)

$$\underline{(\lambda x.(\lambda y.\boldsymbol{I}\,y)\,x)\,z}$$
$$(\lambda x.\underline{(\lambda y.\boldsymbol{I}\,y)\,x})\,z$$
$$(\lambda x.(\lambda y.\underline{\boldsymbol{I}\,y})\,x)\,z$$

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

$$((((\boldsymbol{I}\,x)(\boldsymbol{I}\,y))(\boldsymbol{I}\,z))(\boldsymbol{I}\,w))$$

tiene los redex

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

El redex más a la izquierda es $(\boldsymbol{I}\,x)$ y el más a la derecha es $(\boldsymbol{I}\,w)$.

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 $M \equiv \boldsymbol{I}\,(\boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z))$, su reducción bajo esta estrategia es

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

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

$$(\lambda x.x\,x\,x)((\lambda y.\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,y)\,z)$$

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

$$((\lambda y.\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,y)\,z)((\lambda y.\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,y)z)((\lambda y.\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,y)\,z)$$

Por lo tanto, las reducciones realizadas sobre $((\lambda y.\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,y)\,z)$ 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 $M \equiv \boldsymbol{I}\,(\boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z))$, su reducción bajo esta estrategia es

$$\underline{\boldsymbol{I}\,(\boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z))} \to_\beta \boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z)$$
$$\underline{\boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z)} \to_\beta \lambda z.\boldsymbol{I}\,z$$

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 $M \equiv \boldsymbol{I}\,(\boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z))$, su reducción bajo esta estrategia es

$$\boldsymbol{I}\,(\underline{\boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z)}) \to_\beta \boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z)$$
$$\underline{\boldsymbol{I}\,(\lambda z.\boldsymbol{I}\,z)} \to_\beta \lambda x.\boldsymbol{I}\,z$$

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

$$(\lambda x.x\,x\,x)\,((\lambda y.\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,y)\,z)$$

Primero se reduce $(\lambda y.\boldsymbol{I}\,\boldsymbol{I}\,\boldsymbol{I}\,y)z$ 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, por ejemplo, las expresiones condicionales evalúan la condición antes que el consecuente y la alternativa.

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 [9, 35]. Esto puede parecer extraño ya que claramente los programas de computadora pueden realizar cómputos 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 en él.

En la primera 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 [34, 31, 29, 30, 32].

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 $(\boldsymbol{K}\, x) \twoheadrightarrow_\beta (\lambda x.y)$. Sin embargo es posible codificar los valores de verdad como elementos de Λ y construir abstracciones que emulen las propiedades de las operaciones booleanas bajo la β-reducción. De esta manera se pueden escribir términos que, de acuerdo con la codificación establecida, representen expresiones booleanas y términos 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 {falso, verdadero}. Por ejemplo, al escribir un programa en donde se necesite tomar una desición a partir de si un número n es positivo o negativo se escribiría (en pseudocódigo):

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

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

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

Al igual que el cálculo 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 desición (en este caso P) y proveer dos objetos que podemos distinguir entre sí (en este caso la manzana y la pera). Las desiciones de esta persona pueden ser los conceptos de falso y verdadero si nunca podemos conocer los valores booleanos.

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

Supongamos que P es un término 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 λ M y N se β-reduce a M si la oración O es verdadera y a M si es falsa:

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

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

Para fines prácticos no es necesario saber cómo es P ni O, lo importante es que cuando O es cierta, D eligirá M y si O es falsa, eligirá N. Por lo tanto, $(P\,O) = D$ es un término 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 $\boldsymbol{T}$ y $\boldsymbol{F}$ respectivamente.

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

Utilizar $\boldsymbol{T}$ y $\boldsymbol{F}$ en términos 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

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

y la negación es una operación unaria definida en

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

Las tablas de verdad en la Tabla 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

Tabla 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:

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

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

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

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

- Combinar valores de verdad
- Programar las operaciones

La primera 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 $\{\boldsymbol{T}, \boldsymbol{F}\}^+$.

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

Combinaciones de valores de verdad

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

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

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

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

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

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

- Al igual que la primera reducción $(\boldsymbol{F}\,\boldsymbol{T})$ se reduce a $\boldsymbol{I}$, por lo tanto se concluye que para

cualesquiera términos $M \in \Lambda$, $N \in \{\boldsymbol{F},\ \boldsymbol{T}\}$

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

- $(\boldsymbol{T}\, \boldsymbol{F})$ se reduce a la abstracción constante de $\boldsymbol{F}$, esto significa que para cualquier término $M \in \Lambda$
$$\boldsymbol{\lambda} \vdash (\boldsymbol{T}\, \boldsymbol{F}\, M) = \boldsymbol{F}$$
- Al igual que la tercera reducción $(\boldsymbol{T}\, \boldsymbol{T})$ se reduce a $(\boldsymbol{K}\, \boldsymbol{T})$, por lo tanto se concluye que para cualesquiera términos $M \in \Lambda$, $N \in \{\boldsymbol{F},\ \boldsymbol{T}\}$
$$\boldsymbol{\lambda} \vdash (\boldsymbol{T}\, N\, M) = N$$

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

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

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

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

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

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

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

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

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

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

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

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

Al observar la Tabla 3.3, se distinguen algúnos patrones en los resultados de las reducciones, por ejemplo, si P es un valor de verdad cualquiera, $(P\,\boldsymbol{F}\,\boldsymbol{F})$ se reduce a $\boldsymbol{F}$ y $(P\,\boldsymbol{T}\,\boldsymbol{T})$ se reduce a $\boldsymbol{T}$, las combinaciones mas interesantes se presentan en los renglones 2, 3, 6 y 7.

En búsqueda de las operaciones binarias de conjunción y disyunción se desarrollan tablas de verdad con las posibles combinaciones de dos términos $P, Q \in \{\boldsymbol{F},\ \boldsymbol{T}\}$. La cantidad de combinaciones de estos valores es $2 \times \binom{3}{2} = 2 \times \frac{3!}{2!} = 6$ y son

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

Las tablas de verdad de estas combinaciones intercambiando las posiciones de P y Q serían las mismas ya que ambos términos toman los valores de falso y verdadero en las tablas de verdad. En la Tabla 3.4 se muestran estas tablas.

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

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

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

Ya que $\boldsymbol{\lambda} \vdash (P\,Q\,\boldsymbol{F}) = (\wedge\,P\,Q)$ y $\boldsymbol{\lambda} \vdash (P\,\boldsymbol{T}\,Q) = (\vee\,P\,Q)$, 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

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

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

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

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

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

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

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

Esta metodología para encontrar operaciones del álgebra booleana, aplicando los términos codificados de los valores de verdad, es tediosa pero hasta cierto grado efectiva. Como en los casos de las operaciones no básicas mostradas en la Tabla 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 $(\boldsymbol{T}\,M\,N) \twoheadrightarrow_\beta M$ y $(\boldsymbol{F}\,M\,N) \twoheadrightarrow_\beta N$, si $M \twoheadrightarrow_\beta M'$ y $N \twoheadrightarrow_\beta N'$, entonces

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

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

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

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

Esta aplicación de un valor de verdad a dos términos 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

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

Un programa de la forma

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

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

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

Algoritmo 3.1 Negación de p

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

pseudocódigo se traduce al cálculo λ como

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

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

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

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

Algoritmo 3.2 Conjunción de p_1 y p_2

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

```
si p1 entonces
  si p2 entonces
    regresar verdadero
  de lo contrario
    regresar false
  fin si
de lo contrario
  regresar falso
fin si
```

Traducido al cálculo λ como

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

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

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

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

Se traduce al cálculo λ como

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

Algoritmo 3.3 Disyunción de p_1 y p_2

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

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

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

Utilizando esta técnica, se puede obtener el término 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	$P \odot Q$
$\boldsymbol{F}$	$\boldsymbol{F}$	x_1
$\boldsymbol{F}$	$\boldsymbol{T}$	x_2
$\boldsymbol{T}$	$\boldsymbol{F}$	x_3
$\boldsymbol{T}$	$\boldsymbol{T}$	x_4

El procedimiento generalizado es
Y la traducción al cálculo λ es

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

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

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

```
si p1 entonces
  si p2 entonces
    regresar x4
  de lo contrario
    regresar x3
  fin si
de lo contrario
  si p2 entonces
    regresar x2
  de lo contrario
    regresar x1
  fin si
fin si
```

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

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

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

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

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

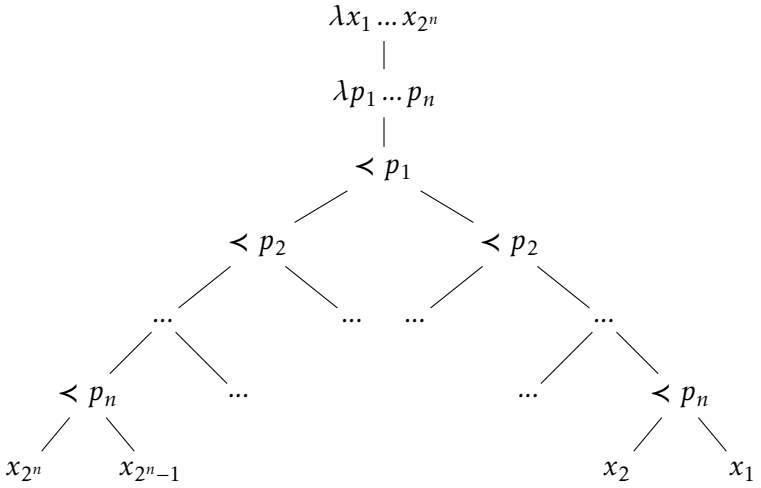
La operación *NOR* se denota $P \downarrow Q$ y tiene la siguiente tabla de verdad

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

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

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

Un aspecto interesante de este método de traducción de operaciones booleanas es que se puede adaptar para operaciones n-árias. Un bosquejo de la forma de estas generalizaciones es



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 la Definición 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

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

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

Definición 3.1.8 (Condicional trivalente)**.**

$$\prec_{\mathbf{3}} \equiv \lambda p\, m\, n\, o.p\, m\, n\, o$$

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

$$(\prec_{\mathbf{3}}\ P\, M\, N\, O) \twoheadrightarrow_{\beta} \begin{cases} M & P \equiv \boldsymbol{T}; \\ N & P \equiv \boldsymbol{F}; \\ O & P \equiv \boldsymbol{U}. \end{cases}$$

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

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

El procedimiento en pseudocódigo que la describe se muestra en el Algoritmo 3.5 y es traducido al cálculo λ como

$$\lambda x_1\, x_2\, x_3\, x_4\, x_5\, x_6\, x_7\, x_8\, x_9.(\lambda p_1\, p_2.(\prec_{\mathbf{3}}\ p_1\, R_1\, R_2\, R_3))$$

Donde

$$\begin{aligned} R_1 &\equiv (\prec_{\mathbf{3}}\ p_2\, x_1\, x_2\, x_3) \\ R_2 &\equiv (\prec_{\mathbf{3}}\ p_2\, x_4\, x_5\, x_6) \\ R_3 &\equiv (\prec_{\mathbf{3}}\ p_2\, x_7\, x_8\, x_9) \end{aligned}$$

3.2. Aritmética

La aritmética es una de las ramas más antiguas de las matemáticas. Consiste en el estudio de los números y de las operaciones elementales como la suma y la multiplicación. El manejo

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

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

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

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 manejan números naturales y expresiones aritméticas como por ejemplo en la Definición 2.1.3 de longitud. Al igual que el álgebra boolena, 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 la Sección 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

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

En la codificación del álgebra booleana se presenta la representación de valores de verdad como una decisión entre dos valores dados. En la Subsección 3.1.3 se extiende la representación de la decisión 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 [10, p. 28].

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

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

Los primeros 5 numerales son:

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

Al manipular numerales de Church, se debe tener cuidado en la manera en que se reducen aplicaciones con otros términos. Con la codificación de valores de verdad es relativamente sencillo experimentar con la manera en la que $\boldsymbol{T}$ y $\boldsymbol{F}$ se combinan y corroborar manualmente

que una combinación se reduce a otra. Sin embargo, al aplicar un numeral de Church $\widehat{n}$ a un término cualquiera M, el término resultante de la contracción de dicha aplicación tendría aproximadamente una longitud de $\|M\| \times n$, lo cual resulta inconveniente de escribir en cada paso de una reducción.

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

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

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

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

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

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

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

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

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

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

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

Para cualquier numeral $\widehat{m}$:

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

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

anterior:

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

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

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

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

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

3.2.2. Operaciones elementales

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

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

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

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

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

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

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

Por lo tanto, la codificación de la exponenciación es

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

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

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

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

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

Por lo tanto, la codificación de la multiplicación es

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

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

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

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

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

Por lo tanto, la codificación de la adición es

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

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

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

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

Figura 3.1: Codificaciones de adición, multiplicación y exponenciación

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

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

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

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

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

Los números naturales nacieron a la par de la necesidad humana de *contar*. De manera similar a la analogía presentada al inicio de la Subsección 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Para demostrar que esta definición es correcta, consideremos la reducción de la aplicación

de $\widehat{+}_1$ en un numeral cualquiera $\widehat{n}$:

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

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

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

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

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

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

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

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

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

unaria, por la definición de multiplicación (3.11) se tiene que

$$m + m + ... + m = m + m + ... + m + 0 = (m + (m + ... + (m + 0)...))$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figura 3.2: Codificaciones de adición, multiplicación y exponenciación

La segunda columna muestra las tres codificaciones escritas de manera compacta, se observa que en los tres casos el átomo n corresponde al operando derecho de la operación y es el que determina la cantidad de veces que se aplicará un procedimiento. Coloreados con azúl se encuentran los valores iniciales a los que se aplica el procedimiento, estos corresponden al caso trivial de la operación, es decir, 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 tercera columna se encuentran las codificaciones escritas sin abuso de notación, coloreado en magenta están los términos a los que n es aplicado primero, esto es, los términos que serán aplicados una y otra vez. Estos términos se encuentran en función de la operación anterior (en orden de menor a mayor complejidad). Al ver las definiciones se puede saber que la expoenciación es repetición de multiplicaciones con caso base 1, la multiplicación es repetición de adiciones con caso base 0 y la adición es repetición de sucesiones con caso base m.

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

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

$$\begin{aligned}
-_1 0 &= 0 \\
-_1 n &= n - 1
\end{aligned}$$

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

Henk Barendreght, en el artículo titulado "The Impact of Lambda Calculus in Logic and Computer Science" [4], menciona que la codificación del predecesor en el cálculo 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

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

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

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

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

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

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

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

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

El problema de codificar el predecesor se reduce ahora a la construcción del sucesor de un número con la codificación modificada. Esta construcción resulta ser casi tan sencilla como la codificación de $\widehat{+}_1$. Primero se analiza cómo cambia la estructura de la codificación de $\widehat{n}'$ a $\widehat{n+1}'$:

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

Cuando se computa el sucesor de $\widehat{0}'$, la única variable enlazada es y y el resultado es $z\,y$, por lo tanto, la y será 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 para un caso concreto, se considera la

aplicación $(\widehat{3'}\, x\,(z\,y)\,x)$:

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

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

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

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

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

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

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

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

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

Teniendo la codificación del predecesor se puede plantear una codificación de la resta,

después de todo, restar n de m es aplicar la función predecesor n veces a m.

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

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

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

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 $(\widehat{n}(\widehat{-}\,\widehat{m})\,\widehat{0})$ ya que el procedimiento de reducción consistiría en primero calcular $m - 0 = m$, después calcular $m - m = 0$, después calcular $m - 0 = m$, y así sucesivamente hasta dejar de repetir el procedimiento. En pocas palabras, $(\widehat{n}(\widehat{-}\,\widehat{m})\,\widehat{0})$ se reduce a $\widehat{m}$ cuando $\widehat{n}$ es impar y a $\widehat{0}$ cuando $\widehat{n}$ es par.

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

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

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

Para el caso de la función logaritmo, el problema es similar a la división. El resultado de calcular el logaritmo base 3 de 81 es 4 porque 4 es la cantidad de veces que hay que dividir al 81 en 3 hasta llegar al 1 (el caso trivial para la división):

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

Para el caso de la función raíz, el problema es aún más grande que con la división y

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

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

3.2.3. Iteración

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

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

Algoritmo 3.6 Factorial de n

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

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

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

La clave para codificar este algoritmo es determinar los valores que representan el cómpu-

to. Las variables r y a describen el estado del cómputo en el caso trivial $n = 0$. Cuando $n \neq 0$, en cada iteración, toda la información del cómputo sigue estando en las variables r y a, aún más, los valores de estas variables describen una propiedad interesante del algoritmo: Antes y después de cada iteración, se cumple que $n! = a \times r!$.

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

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

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

Para 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

El primer predicado que se define es el que a partir de un numeral $\widehat{n}$, determina si es el $\widehat{0}$. Este predicado es muy utilizado ya que en muchos algoritmos, la condición de paro es cuando un valor numérico tiene el valor cero. A partir de la estructura de $\widehat{n} \equiv (\lambda x\,y.x^n\,y)$ se puede encontrar una manera de aplicarle dos términos P y Q a $\widehat{n}$, tal que al reducirse resulte en $\boldsymbol{T}$ si $n = 0$ y a $\boldsymbol{F}$ si $n > 0$. El $\widehat{0}$ no tiene aplicaciones internas y simplemente es reducido al

segundo término al que fue aplicado, por lo tanto $(\widehat{n}\,P\,\boldsymbol{T})$ debe reducirse a $\boldsymbol{T}$ cuando $n = 0$. Cuando el numeral es mayor a cero, la primera aplicación de $(P\,\boldsymbol{T})$ debe reducirse a $\boldsymbol{F}$ y las siguientes aplicaciones deben ser $(P\,\boldsymbol{F})$ y reducirse también a $\boldsymbol{F}$. Un término que al ser aplicado a cualquier término es reducido a $\boldsymbol{F}$ es $(\boldsymbol{K}\,\boldsymbol{F})$. Por lo tanto, la codificación de este predicado se define

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

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

$$\begin{aligned}
\widehat{0_?}\,\widehat{0} &\equiv (\lambda n.n\,(\boldsymbol{K}\,\boldsymbol{F})\,\boldsymbol{T})\,\widehat{0}\\
&\to_\beta \widehat{0}\,(\boldsymbol{K}\,\boldsymbol{F})\,\boldsymbol{T}\\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^0\,\boldsymbol{T}\\
&\equiv \boldsymbol{T}\\
\widehat{0_?}\,\widehat{n} &\equiv (\lambda n.n\,(\boldsymbol{K}\,\boldsymbol{F})\,\boldsymbol{T})\,\widehat{n}\\
&\to_\beta \widehat{n}\,(\boldsymbol{K}\,\boldsymbol{F})\,\boldsymbol{T}\\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^n\,\boldsymbol{T}\\
&\equiv (\boldsymbol{K}\,\boldsymbol{F})^{n-1}\,(\boldsymbol{K}\,\boldsymbol{F}\,\boldsymbol{T})\\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{n-1}\,\boldsymbol{F}\\
&\equiv (\boldsymbol{K}\,\boldsymbol{F})^{n-2}\,(\boldsymbol{K}\,\boldsymbol{F}\,\boldsymbol{F})\\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{n-2}\,\boldsymbol{F}\\
&\;\;\ldots\\
&\twoheadrightarrow_\beta (\boldsymbol{K}\,\boldsymbol{F})^{n-n}\,\boldsymbol{F}\\
&\equiv \boldsymbol{F}
\end{aligned}$$

El siguiente predicado que se define es el que a partir de dos numerales $\widehat{m}$ y $\widehat{n}$, determina si $m \leq n$. La codificación de este predicado se basa en la observación de al restar $\widehat{n}$ de $\widehat{m}$ el resultado será $\widehat{0}$ si $\widehat{m}$ es menor o igual a $\widehat{n}$. Por lo tanto, la codificación de este predicado se define

$$\widehat{\leq_?} \;\equiv\; \lambda m\,n.\widehat{0_?}\,(\widehat{-}\,m\,n) \tag{3.33}$$

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

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

$$\widehat{\geq_?} \;\equiv\; \lambda m\,n.\widehat{\leq_?}\,n\,m \tag{3.34}$$

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

$$\widehat{=_?} \equiv \lambda m\, n.\, \wedge\, (\widehat{\leq_?}\, m\, n)(\widehat{\geq_?}\, m\, n) \tag{3.35}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

las codificaciones que se definen, serán versiones discretas, cerradas y por lo tanto inexactas de las que usualmente se utilizan.

El algoritmo en el que se basan estas cuatro operaciones considera dos números m y n; una operación inversa $\odot$; una condición de trivialidad err; y un valor de trivialidad t. La idea del algoritmo es regresar t cuando $\text{err}(m,n)$ 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: $m,\ n,\ t \in \mathbb{N},\ \odot\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N},\ err\colon \mathbb{N}\times\mathbb{N} \to \{\text{verdadero, falso}\}$
Salida: $m \odot^{-1} n$

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

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 del Algoritmo 3.7 para que su codificación sea más directa.

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

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

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

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

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

La codificación del Algoritmo 3.8 es

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

La definición es difícil de leer y comprender si se escribe en un solo renglón, por ello, se utilizaron varios renglones para escribir el término, la convención para escribirlo fue: los saltos de renglón se dan en cada término condicional para que la condición esté en el mismo renglón que $\prec$, el consecuente en el siguiente renglón horizontalmente alineado con la condición y la alternativa en el siguiente renglón horizontalmente alineado con el

consecuente. También hay un salto de renglón en la aplicación de m, de tal manera que los dos términos a los que es aplicado estén alineados horizontalmente.

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

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

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

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

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

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

3.2.4. Hiperoperaciones

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

En el artículo "Mathematics and Computer Science: Coping with Finiteness" [18], Donald E. Knuth introduce la notación de flecha para expresar números finitos gigantescos. La notación de flecha ya fue utilizada en el término de la exponenciación, ya que, de acuerdo a Knuth, $x \uparrow n = x^n$.

La definición de la notación de flecha es la siguiente:

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

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

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

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

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

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

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

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

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

Se definen los términos $\widehat{\uparrow}_i$ de la siguiente manera

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

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

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

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

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

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

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

3.3. Procesos recursivos

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

La recursividad está detrás de una gran 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 [13].

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* [13]. 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 la Definición 2.1.3 y la Definición 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] [30].

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 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 el Algoritmo 3.6 como el Algoritmo 3.9 como la codificación (3.28) describen el mismo proceso computacional.

Un tercer algoritmo para el cálculo del factorial es el Algoritmo 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:

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

- 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.

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; 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 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:

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

El problema con esta definición es que antes de definir $\widehat{!}$, se hace referencia a ella en $(\widehat{!}(\widehat{-}_1 n))$, por lo que no es posible establecer el valor del término factorial antes de terminar de escribir su definición.

Hay algunos trucos que se pueden implementar para simular que se tiene definida la codificación de factorial antes de definirla. Por ejemplo, si $\widehat{!}$ fuera una abstracción la cual espera ser aplicada a sí misma y a un número, pudiera definirse exactamente como (3.47), pero con una variable enlazada f que será sustituída por el término $\widehat{!}$ dentro de la definición.

$$\begin{aligned}\widehat{!} \equiv\ & \lambda f\, n.(\prec (\widehat{0}_? n) \\ & \widehat{1} \\ & (\widehat{\times} n (f\, (\widehat{-}_1 n))))\end{aligned} \tag{3.48}$$

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

$$\widehat{!!}\widehat{n} \twoheadrightarrow_\beta (\widehat{\times}\widehat{n}(\widehat{!}(\widehat{-}_1 \widehat{n})))$$

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

$$(\,\widehat{\times}\,\widehat{n}\,(\,\widehat{!}\ \underbrace{[\ \ ]}_{\text{Debe de ir } \widehat{!}}\ \ (\,\widehat{-}_1\,\widehat{n}\,))\,)$$

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

$$\begin{aligned}\widehat{!} \equiv\ & \lambda f\, n.(\prec (\widehat{0}_? n) \\ & \widehat{1} \\ & (\widehat{\times} n (f\, f\, (\widehat{-}_1 n))))\end{aligned} \tag{3.49}$$

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

$$\widehat{!!} \twoheadrightarrow_\beta \lambda n.(\prec (\widehat{0}_? n)\widehat{1}(\widehat{\times} n (\widehat{!!}(\widehat{-}_1 n))))$$

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

$$
\begin{array}{l}
((\lambda f\, n.(\prec\ (\widehat{0}_?\, n)\\
\qquad \widehat{1}\\
\qquad (\widehat{\times} n\, (f\, f\, (\widehat{-}_1\, n)))))\\
\quad (\lambda f\, n.(\prec\ (\widehat{0}_?\, n)\\
\qquad \widehat{1}\\
\qquad (\widehat{\times} n\, (f\, f\, (\widehat{-}_1\, n))))))
\end{array}
\twoheadrightarrow_\beta
\begin{array}{l}
(\lambda n.(\prec\ (\widehat{0}_?\, n)\\
\qquad \widehat{1}\\
\qquad (\widehat{\times} n\\
\qquad\quad ((\lambda f\, n.(\prec\ (\widehat{0}_?\, n)\\
\qquad\qquad \widehat{1}\\
\qquad\qquad (\widehat{\times} n\, (f\, f\, (\widehat{-}_1\, n)))))\\
\qquad\quad (\lambda f\, n.(\prec\ (\widehat{0}_?\, n)\\
\qquad\qquad \widehat{1}\\
\qquad\qquad (\widehat{\times} n\, (f\, f\, (\widehat{-}_1\, n)))))\\
\qquad\quad (\widehat{-}_1\, n)))))
\end{array}
$$

3.3.3. Combinador genérico de recursividad

A pesar de tener este problema resuelto, la solución no es buena. Esta técnica obliga a que en cada algoritmo recursivo que se codifique, cada aplicación recursiva se deba tener el primer argumento aplicado a sí mismo. Lo que se necesita para tener un buen mecanismo de recursividad es separar la auto-aplicación de una codificación recursiva de la definición misma. Lo ideal es poder codificar un algoritmo similar a la definición (3.48) y mediante algún procedimiento genérico, hacer que se auto-aplique el procedimiento a si mismo.

La "factorización" del mecanismo de recursión se puede lograr combinando la idea de la auto-aplicación de la ecuación (3.49) y la idea de escribir los términos recursivos como en la ecuación (3.48).

En (3.48) se esperaba que $\widehat{!}$ sea aplicado a sí mismo, sin embargo en su definición solo aplica f a $(\widehat{-}_1\, n)$, por lo que es conveniente tratar de reducir la aplicación de $\widehat{!}$ a $(\widehat{!!})$, de tal manera que la reducción sea

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

Esto hace que la codificación funcione en el primer paso recursivo. Sin embargo, $\widehat{(!\,!)}(\widehat{-}_1\, n))$ será reducido de tal manera que el siguiente paso recursivo no se aplique $\widehat{!}$ a si mismo.

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

El término $\widehat{(!\,(\widehat{-}_1\, n))}$ debería de ser $\widehat{(!\,!\,(\widehat{-}_1\, n))}$ de nuevo para que el siguiente paso recursivo funcione.

En la Sección 1.2 se mostró un término que es útil para este tipo de situaciones:

$$\omega \;\equiv\; \lambda x.x\,x$$

La propiedad interesante de ω es que $(\omega\,\omega) \twoheadrightarrow_\beta (\omega\,\omega)$, que es justo lo que deseamos en nuestro mecanismo recursivo, el combinador que se necesita es uno similar a ω, llamado ω' tal que $(\omega'\,\omega') \twoheadrightarrow_\beta \widehat{(!(\omega'\,\omega'))}$.

Suponiendo que $\widehat{!}$ ya es una variable enlazada, el combinador ω' debe ser una abstracción que espera ser aplicada a si misma, tiene la forma $\lambda\omega'.M$. Ya que la reducción de ω' aplicada a sí misma resulta en $\widehat{(!(\omega'\,\omega'))}$, M debe de ser la aplicación $\widehat{(!N)}$, para completar la regla de reducción $N \;\equiv\; (\omega'\,\omega')$.

$$\omega' \;\equiv\; \lambda\omega'.\widehat{!}(\omega'\,\omega') =_\alpha \lambda x.\widehat{!}(x\,x)$$

Para completar la definición del mecanismo de recursividad se plantea un combinador que espere ser aplicado a un término como $\widehat{!}$ de (3.48) e internamente aplique ω' a sí misma. Este mecanismo es llamado combinador $\boldsymbol{Y}$.

$$\boldsymbol{Y} \;\equiv\; \lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x)) \tag{3.50}$$

Al aplicar el combinador $\boldsymbol{Y}$ a la definición (3.48) del factorial, se obtiene que

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

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

$$\begin{aligned}
&(\lambda n.(\prec\, \widehat{(0_?\, n)} \\
&\quad \widehat{1} \qquad\qquad\qquad\qquad \twoheadrightarrow_\beta \quad \widehat{(\times \widehat{n}\,(\boldsymbol{Y}\widehat{!}\,(\widehat{-}_1\, \widehat{n})))} \\
&\quad \widehat{(\times n\,(\boldsymbol{Y}\widehat{!}\,(\widehat{-}_1\, n)))}))\widehat{n}
\end{aligned}$$

Es entonces que $(\boldsymbol{Y}\widehat{!})$ puede ser β-convertido nuevamente a $\widehat{(!(\boldsymbol{Y}\widehat{!}))}$ 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 3.3.1 es uno de los resultados básicos en el cálculo λ.

Teorema 3.3.1 (Teorema de punto fijo)**.** $\forall F \exists X \colon F\,X =_\beta X$

Demostración. Sea $W \equiv \lambda x.F(x\,x)$ y $X \equiv W\,W$. Entonces

$$X \equiv (\lambda x.F(x\,x))\,W \to_\beta F(W\,W) \equiv F\,X$$

□

Esta demostración es algo peculiar ya que se inicia con X y se reduce este término a $(F\,x)$, en lugar de partir hacer el proceso inverso. Sin embargo, la reducción presentada es válida de acuerdo a la definición de β-convertibilidad.

El combinador $\boldsymbol{Y}$ derivado en la Subsección 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 [15, p. 34], es decir, si M es un combinador de punto fijo, entonces, para toda F

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

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

$$\begin{aligned} \boldsymbol{Y}\,F &\equiv (\lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x)))F \\ &\to_\beta (\lambda x.F\,(x\,x))(\lambda x.F\,(x\,x)) \\ &\to_\beta F((\lambda x.F\,(x\,x))(\lambda x.F\,(x\,x))) \\ &=_\beta F(\boldsymbol{Y}\,F) \end{aligned} \tag{3.52}$$

Existen combinadores con una propiedad más fuerte que (3.51). Alan Turing descubrió el combinador $\boldsymbol{\Theta}$ en 1937, su definicón es

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

La propiedad interesante de $\boldsymbol{\Theta}$ es que puede computar puntos fijos únicamente con la β-reducción. Se corrobora esto de la siguiente manera

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

Esta propiedad no la tiene $\boldsymbol{Y}$, para realizar el último paso del desarrollo (3.52), se tuvo que hacer una reducción inversa.

La utilidad de los combinadores de punto fijo va más allá que el de permitir la codificación de procesos recursivos. Estos combinadores son especialmente útiles para resolver el siguiente tipo de problema:

Sea Z un término λ, con variables libres f y $\vec{x}$, encuentra el término F tal que

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

En este planteamiento F puede no aparecer en Z y la solución es el mismo término a que si apareciera. Una instancia de este problema puede ser una reformulación de la codificación recursiva del factorial. Sea $Z \equiv (\prec \widehat{(0}\,?\,n)\widehat{1}\,(\widehat{\times} n\,(f\,(\widehat{-}_1\,n))))$, encuentra el término $\widehat{!}$ tal que

$$\widehat{!}\widehat{n} =_\beta Z\left[f := \widehat{!}\right][n := \widehat{n}]$$

Solución. Por la regla (β) se tiene que

$$\widehat{!}\widehat{n} =_\beta (\lambda f\,n.\prec \widehat{(0}\,?\,n)\widehat{1}\,(\widehat{\times} n\,(\widehat{!}\,(\widehat{-}_1\,n))))\widehat{!}\widehat{n}$$

Por la regla (ν) se tiene que

$$\widehat{!} =_\beta (\lambda f\,n.\prec \widehat{(0}\,?\,n)\widehat{1}\,(\widehat{\times} n\,(\widehat{!}\,(\widehat{-}_1\,n))))\widehat{!}$$

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

$$\widehat{!} \equiv \boldsymbol{Y}\,(\lambda f\,n.\prec \widehat{(0}\,?\,n)\widehat{1}\,(\widehat{\times} n\,(f\,(\widehat{-}_1\,n))))$$

□

En general, la solución para este tipo de problemas es

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

3.4. Estructuras recursivas

Utilizando combinadores de punto fijo como $\boldsymbol{Y}$ y $\boldsymbol{\Theta}$ y la solución general (3.55) se pueden codificar procedimientos recursivos como el Algoritmo 3.10 del factorial. Sin embargo, los procesos recursivos se tornan más interesantes cuando la información que manipulan es también recursiva. En esta sección se muestran técnicas para codificar estructuras recursivas en el cálculo 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 "construír en memoria". Este operador toma dos objetos a y d y construye en memoria un objeto que contiene a ambos.

Matemáticamente, el objeto resultante de aplicar el operador *cons* es un par ordenado. La notación que se utiliza en este trabajo para escribir el par conformado por a y d es $\langle a : d \rangle$.

Un par ordenado no es más que un estado con dos elementos, para codificarse en el cálculo λ, se deben plantear los mecanismos para construír pares y obtener sus elementos. A continuación se construyen las definiciones del constructor p y los selectores a y d para el par ordenado (basadas en las definiciones (3.30) y (3.31)).

$$p \equiv \lambda a\, d.\lambda q.q\, a\, d \tag{3.56}$$

$$a \equiv \lambda c.c\,(\lambda a\, d.a) \tag{3.57}$$

$$d \equiv \lambda c.c\,(\lambda a\, d.d) \tag{3.58}$$

Estas tres ecuaciones cumplen con las reducciones

$$(a\,(p\,M\,N)) \twoheadrightarrow_\beta M$$

$$(d\,(p\,M\,N)) \twoheadrightarrow_\beta N$$

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

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

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

En el resto de esta sección se abordan diferentes maneras en las que se puede emplear la estructura del par para construír estructuras más complejas.

3.4.1. Listas

Las listas son secuencias de valores, en donde cada valor en la lista tiene una posición fija. La codificación de las listas en el cálculo 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 $\emptyset$.

$$\langle \widehat{n_1} : \langle \widehat{n_2} : \langle \widehat{n_3} : \langle ... \langle \widehat{n_k} : \emptyset \rangle ... \rangle \rangle \rangle \rangle$$

Denotada de manera abreviada como

$$\langle \widehat{n_1},\ \widehat{n_2},\ \widehat{n_3},\ \ldots,\ \widehat{n_k} \rangle$$

La codificación de $\emptyset$ debe elegirse con mucho cuidado. En los algoritmos que manipulan listas es crucial determinar cuando se ha llegado al final de la lista, por lo tanto se hace uso de un predicado para determinar si un determinado objeto es un par o es el valor nulo (de manera similar a la comparación de un número con el cero en los algoritmos aritméticos).

Así como el predicado $\widehat{0_?}$ fue construído asumiendo que sería aplicado a un numeral, el término $\emptyset$ se construye asumiendo que será aplicado a una lista de números. Formalmente una lista de numeros, o es un par cuyo primer elemento es un numeral de Church y cuyo segundo elemento es otra lista, o es el término nulo (lista con cero elementos).

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

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

$$\emptyset_?(\lambda q.q\,\widehat{n}\,\mathscr{L}) \twoheadrightarrow_\beta \boldsymbol{F}$$
$$\emptyset_?\emptyset \twoheadrightarrow_\beta \boldsymbol{T}$$

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

$$(\lambda x\,y.\boldsymbol{F})M\,N \twoheadrightarrow_\beta \boldsymbol{F}$$

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

$$\begin{aligned}(\lambda q.q\,\widehat{n}\,\mathscr{L})(\lambda x\,y.\boldsymbol{F}) &\to_\beta (\lambda x\,y.\boldsymbol{F})\,\widehat{n}\,\mathscr{L}\\ &\twoheadrightarrow_\beta \boldsymbol{F}\end{aligned}$$

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

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

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

$$\begin{aligned}\emptyset_?\,(\boldsymbol{K}\,\boldsymbol{T}) &\equiv (\lambda l.l\,(\lambda x\,y.\boldsymbol{F}))(\boldsymbol{K}\,\boldsymbol{T})\\ &\to_\beta \boldsymbol{K}\,\boldsymbol{T}\,(\lambda x\,y.\boldsymbol{F})\\ &\twoheadrightarrow_\beta \boldsymbol{T}\end{aligned}$$

Por lo tanto

$$\emptyset \equiv \boldsymbol{K}\,\boldsymbol{T} \tag{3.63}$$

Observación (Sobre predicados). La aplicación de un predicado como $\widehat{0}_?$ o $\emptyset_?$ a un término λ M no necesariamente se β-reduce a una codificación de valor de verdad. Estos predicados son construídos para ser aplicados a numerales o listas respectivamente y el resultado de reducir otro tipo de términos no es de importancia para la codificación de algoritmos.

Con estas codificaciones se pueden construír algoritmos que manipulen listas de números. Consideremos el Algoritmo 3.11 que dada una lista de números compute una lista con la misma cantidad de elementos pero con cada número de la lista original multiplicado por 2. En cada paso recursivo se verifica que la lista $\mathscr{L}$ no sea el valor nulo, en cuyo caso, será un

par cuyo primer elemento es n y segundo elemento es otra lista $\mathscr{L}'$; se construye otro par con el primer elemento multiplicado por 2 y con el segundo elemento el resultado de realizar este mismo proceso con $\mathscr{L}'$.

Algoritmo 3.11 Procedimiento recursivo dobles($\mathscr{L}$)

Entrada: $\mathscr{L} \in \{\langle n_1, \ldots, n_k\rangle \mid n \in \mathbb{N}\} \cup \emptyset$
Salida: $\langle 2 \times n_1, \ldots, 2 \times n_k\rangle$ ó $\emptyset$
  si $\mathscr{L} = \emptyset$ **entonces**
    regresar $\emptyset$
  de lo contrario
    $\langle n : \mathscr{L}'\rangle \leftarrow \mathscr{L}$
    regresar $\langle 2 \times n : \text{dobles}(\mathscr{L}')\rangle$
  fin si

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

$$\begin{aligned} &\boldsymbol{Y}\,(\lambda f\,\mathscr{L}.\prec(\emptyset_?\,\mathscr{L}) \\ &\qquad \emptyset \\ &\qquad (p\;(\widehat{\times 2}\,(a\,\mathscr{L})) \\ &\qquad\quad (f\,(d\,\mathscr{L})))) \end{aligned} \tag{3.64}$$

Teniendo esta codificación, es relativamente fácil generalizarla para que a cada número en la lista se le aplique algún término g dado, el cual puede ser reducido a otro número.

$$\begin{aligned} &\boldsymbol{Y}\,(\lambda f\,g\,\mathscr{L}.\prec(\emptyset_?\,\mathscr{L}) \\ &\qquad \emptyset \\ &\qquad (p\;(g\,(a\,\mathscr{L})) \\ &\qquad\quad (f\,g\,(d\,\mathscr{L})))) \end{aligned} \tag{3.65}$$

Aún más, se puede generalizar la estructura del resultado al abstraer el término final $\emptyset$ y el constructor del par p de la siguiente manera.

$$\begin{aligned} &\boldsymbol{Y}\,(\lambda f\,p\,\emptyset\,g\,\mathscr{L}.\prec(\emptyset_?\,\mathscr{L}) \\ &\qquad \emptyset \\ &\qquad (p\;(g\,(a\,\mathscr{L})) \\ &\qquad\quad (f\,p\,\emptyset\,g\,(d\,\mathscr{L})))) \end{aligned} \tag{3.66}$$

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 $\langle n_1, \ldots, n_k \rangle$, entonces

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

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: $\mathcal{L} \in \{\langle n_1, \ldots, n_k \rangle \mid n \in \mathbb{N}\} \cup \emptyset$
Salida: $\langle n'_1, \ldots, n'_{k'} \rangle$ ó $\emptyset \mid n'_i \in \mathcal{L},\ n'_i$ impar

```
si L = ∅ entonces
    regresar ∅
de lo contrario
    ⟨n : L'⟩ ← L
    si n impar entonces
        regresar ⟨n : impares(L')⟩
    de lo contrario
        regresar impares(L')
    fin si
fin si
```

La codificación de este algoritmo es bastante directa también, sin embargo, se necesita codificar un predicado que determine si un numeral de Church es impar o no. Un algoritmo recursivo puede ser elegido para la codificación del predicado impar, sin embargo, es más fácil en este caso utilizar la estructura de los numerales para lograr el resultado deseado. Sea $\widehat{imp_?}$ la codificación del predicado impar, se deben de satisfacer las siguientes reducciones:

$$\begin{aligned}
\widehat{imp_?}(\lambda x\, y.y) &\twoheadrightarrow_\beta \boldsymbol{F} \\
\widehat{imp_?}(\lambda x\, y.x\, y) &\twoheadrightarrow_\beta \boldsymbol{T} \\
\widehat{imp_?}(\lambda x\, y.x\, (x\, y)) &\twoheadrightarrow_\beta \boldsymbol{F} \\
\widehat{imp_?}(\lambda x\, y.x\, (x\, (x\, y))) &\twoheadrightarrow_\beta \boldsymbol{T} \\
&\ldots
\end{aligned}$$

Una manera sencilla de codificar este término es

$$\widehat{imp_?} \equiv \lambda n.n \neg \boldsymbol{F}$$

De esta manera

$$\begin{aligned}
\widehat{imp_?}(\lambda x\, y.y) &\twoheadrightarrow_\beta \boldsymbol{F} \\
\widehat{imp_?}(\lambda x\, y.x\, y) &\twoheadrightarrow_\beta \neg \boldsymbol{F} \\
\widehat{imp_?}(\lambda x\, y.x\,(x\, y)) &\twoheadrightarrow_\beta \neg(\neg \boldsymbol{F}) \\
\widehat{imp_?}(\lambda x\, y.x\,(x\,(x\, y))) &\twoheadrightarrow_\beta \neg(\neg(\neg \boldsymbol{F})) \\
&\ldots
\end{aligned}$$

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

$$\begin{aligned}
&\boldsymbol{Y}(\lambda f\, \mathscr{L}. \prec (\emptyset_?\, \mathscr{L}) \\
&\qquad \emptyset \\
&\qquad (\prec (\widehat{imp_?}\,(a\, \mathscr{L})) \\
&\qquad\quad (p\,(a\, \mathscr{L})\,(f\,(d\, \mathscr{L}))) \\
&\qquad\quad (f\,(d\, \mathscr{L}))))
\end{aligned} \tag{3.67}$$

Esta abstracción también es posible generalizarla con pocas modificaciones para que, además de una lista considere un predicado $g_?$ que sea reducido a una codificación booleana cuando sea aplicado a un numeral de Church. El término (3.68) muestra este término más general.

$$\begin{aligned}
&\boldsymbol{Y}(\lambda f\, g_?\, \mathscr{L}. \prec (\emptyset_?\, \mathscr{L}) \\
&\qquad \emptyset \\
&\qquad (\prec (g_?\,(a\, \mathscr{L})) \\
&\qquad\quad (p\,(a\, \mathscr{L})\,(f\, g_?\,(d\, \mathscr{L}))) \\
&\qquad\quad (f\, g_?\,(d\, \mathscr{L}))))
\end{aligned} \tag{3.68}$$

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

$$\mathscr{F}\, \widehat{imp_?}\, \mathscr{L} =_\beta \mathscr{I}\, \mathscr{L}$$

Algoritmos de procesamiento de listas más complejos pueden ser codificados ya sea utilizando (3.65), (3.66), (3.68) o términos con una estructura similar. Algo que es importante notar de las generalizaciones planteadas en los ejemplos es que la lista $\mathscr{L}$ puede tener términos λ que no sean numerales, al utilizar los algoritmos genéricos sólo se debe tener cuidado

con que la forma de los términos en $\mathscr{L}$ sea conocida para g y $g_?$.

3.4.2. Árboles

Los árboles son estructuras no-lineales jerárquicas compuestos de *vértices* (también llamados *nodos*) y *aristas* que establecen una relación entre dos vértices.

Utilizando pares y listas es posible construír árboles. Consideremos el árbol mostrado en la Figura 3.3.

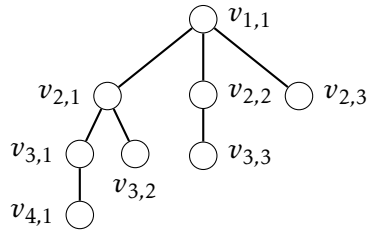


Figura 3.3: Ejemplo de árbol

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

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

Esta definición nos permite codificar árboles como el valor nulo o como un par cuyo primer elemento sea un vértice v y cuyo segundo elemento sea una lista de los vértices en los que v incide. El vértice puede ser representado por cualquier término λ, en ocaciones es útil asociar un término a cada vértice del árbol para construír, por ejemplo, árboles de numerales.

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

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

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

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

Por lo que el predicado $\emptyset_?$ funciona para diferenciar árboles nulos de árboles con vértices y si $\mathcal{T}$ es un árbol no nulo, entonces $(\dot{a}\,\mathcal{T})$ se reduce al vértice del árbol y $(\dot{d}\,\mathcal{T})$ se reduce a los subárboles del árbol.

Esta similitud en la representación de árboles y listas corresponde con la codificación de listas como "pares de pares" y la codificación de árboles como "listas de listas". Para mostrar las posibilidades que estas relaciones nos permiten se considera el Algoritmo 3.13, el cual calcula la cantidad de vértices en un árbol. El procedimiento suma calcula la suma de los elementos de una lista de numeros y el procedimiento map es el que es codificado por (3.65).

Algoritmo 3.13 Procedimiento recursivo cuenta($\mathcal{T}$)

Entrada: $\mathcal{T} \in \{\langle v : \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle \rangle\} \cup \emptyset$
Salida: n = cantidad de vértices en $\mathcal{T}$

```
si T = ∅ entonces
    regresar 0
de lo contrario
    ⟨v : L_T⟩ ← T
    regresar 1 + fold(+, 0, cuenta, L_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

$$\begin{aligned} &\boldsymbol{Y}\,(\lambda f\,\mathcal{T}.\prec(\emptyset_?\,\mathcal{T}) \\ &\qquad \widehat{0} \\ &\qquad (\widehat{+}_1\,(\mathcal{F}\widehat{+}f\,(\dot{d}\,\mathcal{T})))) \end{aligned}$$

3.4.3. Gráficas

Las gráficas son una generalización de árboles en donde se admiten *ciclos*, es decir, a partir de un vértice, se pudieran encontrar dos caminos de aristas para llegar a otro vértice. Consideremos la gráfica mostrada en la Figura 3.4.

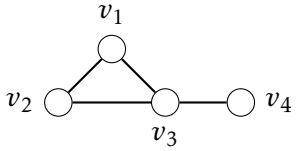


Figura 3.4: Ejemplo de gráfica

Una posible codificación para las gráficas es como *lista de adyacencia*: Una gráfica $\mathcal{G}$ es representada como una lista con un elemento por vértice; cada elemento de esta lista es un par $\langle v : \mathcal{L}_v \rangle$ donde v es un término que representa al vértice y $\mathcal{L}_v$ es una lista con un elemento por vértice en el que v incida; cada elemento de $\mathcal{L}_v$ es un término que representa a un vértice.

La gráfica de la Figura 3.4 se representa como lista de adyacencia de la siguiente manera:

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

La única restricción que se debe incorporar a la codificación es que los vértices puedan ser distinguidos entre sí, esto se puede lograr por ejemplo, utilizando numerales de Church para representar vértices.

Ya que las gráficas son codificadas utilizando términos λ conocidos, los algoritmos que manipular gráficas pueden ser codificados de manera similar a los presentados en la Subsección 3.4.1 y 3.4.2.

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 algúno. Este técnica de representar un lenguaje en sí mismo es común, por ejemplo, en compiladores de `C` escritos en `C` o en intérpretes de `Lisp` escritos en `Lisp`.

Partiendo de la Definición 2.1.1 de término λ, se modifica el conjunto V para que en lugar de ser $\{v_0,\ v_{00},\ v_{000},\ ...\}$, sea $\{0,\ 1,\ 2,\ ...\}$, es decir, que los átomos sean numeros naturales; además en lugar de construír las abstracciones y aplicaciones a partir de símbolos, se construyen a partir de pares.

Definición 3.4.1 (Términos λ)**.** El conjunto Λ' tiene elementos que son pares y números

naturales. Λ' es el conjunto más pequeño que satisface:

$$n \in \mathbb{N} \implies n \in \Lambda' \tag{a}$$
$$M \in \Lambda',\ n \in \mathbb{N} \implies \langle n : M \rangle \in \Lambda' \tag{b}$$
$$M,\ N \in \Lambda' \implies \langle M : N \rangle \in \Lambda' \tag{c}$$

El problema con esta definición de términos 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:

$$n \in \mathbb{N} \implies \langle 1 : n \rangle \in \Lambda' \tag{a}$$
$$M \in \Lambda',\ n \in \mathbb{N} \implies \langle 2 : \langle n : M \rangle \rangle \in \Lambda' \tag{b}$$
$$M,\ N \in \Lambda' \implies \langle 3 : \langle M : N \rangle \rangle \in \Lambda' \tag{c}$$

Con esta definición, el término $(\lambda x.x)$ se puede representar como $\langle 2 : \langle \langle 1 : n \rangle : \langle 1 : n \rangle \rangle \rangle$ y el término $(x\,x)$ se puede representar como $\langle 3 : \langle \langle 1 : n \rangle : \langle 1 : n \rangle \rangle \rangle$.

Codificaciones adecuadas para los constructores y selectores de esta representación de términos λ se basan en las técnicas abordadas a lo largo de este capítulo.

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

$$\text{atomo}_? \equiv \lambda\mathcal{T}.\widehat{=}_?\,\widehat{1}(a\,\mathcal{T}) \tag{3.71}$$
$$\text{abstraccion}_? \equiv \lambda\mathcal{T}.\widehat{=}_?\,\widehat{2}(a\,\mathcal{T}) \tag{3.72}$$
$$\text{aplicacion}_? \equiv \lambda\mathcal{T}.\widehat{=}_?\,\widehat{3}(a\,\mathcal{T}) \tag{3.73}$$

Utilizando estos tres predicados y el término $\prec$ se puede codificar un término que funcione como una estructura de control similar a $\prec$ que a partir de un término codificado $\mathcal{T}$ y cuatro términos M_1, M_2, M_3 y M_4 compute M_1 si $\mathcal{T}$ es un átomo, M_2 si es una abstracción, M_3 si es una aplicación y M_4 en otro caso:

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

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

$$\text{atomo} \equiv \lambda n.p\,\widehat{1}\,n \tag{3.75}$$

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

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

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

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

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

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

Los selectores para cada clase de término 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.

$$\text{atomo}_n \equiv \lambda x.d\,x \tag{3.78}$$

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

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

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

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

La técnica de etiquetar pares para poder determinar el "tipo" de estructura es muy uti-

lizada 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 resultan contraproducente para interesar a un lector que nunca ha sido expuesto al cálculo λ.

Algunos temas que se pueden estudiar después de leer este trabajo son las extensiones del cálculo con lógica ilativa [5, pp. 573–576], teoría de tipos [23], semántica denotacional [33] y programación funcional.

Apéndice A

Programación de codificaciones

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 funciones anónimas, también llamadas *clausuras* e informalmente llamadas *lambdas*.

A.1. Scheme

```
;;
;; 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? (lambda (n) ((n (K F)) T)))
(define SUCC (lambda (n) (lambda (x) (lambda (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 + (lambda (m) (lambda (n) ((n SUCC) m))))
(define * (lambda (m) (lambda (n) ((n (+ m)) N:0))))
(define ^ (lambda (m) (lambda (n) ((n (* m)) N:1))))
(define N:0* (lambda (x) (lambda (y) (lambda (z) y))))
(define SUCC* (lambda (n) (lambda (x) (lambda (y) (lambda (z) (((n x) (z y)) x))))))
(define PRED (lambda (n) (lambda (x) (lambda (y) (((((n SUCC*) N:0*) x) y) I)))))
(define - (lambda (m) (lambda (n) ((n PRED) m))))
```

A.2. Haskell

```
--
-- 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 = \x 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 = \x y z -> n x (z y) x
N_Pred n = \x y -> n N_Succ2 N_02 x y C_I
N_Sub m n = n N_Pred m
```

Apéndice B

Intérprete Lambda

Este apéndice consiste de un programa literario[1] conformado por un intérprete y un editor estructural para el cálculo λ. Se utilizó la herramienta `Noweb`, escrita por Norman Ramsey[2], 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.

[1] La 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.

[2] Para más información, visitar el sitio `http://www.cs.tufts.edu/~nr/noweb/`

Lambda

Un sistema computacional basado en el cálculo λ

por

Eduardo Acuña Yeomans

Universidad de Sonora
Licenciatura en Ciencias de la Computación
Departamento de Matemáticas
División de Ciencias Exactas y Naturales

Hermosillo, Sonora
2016

Índice general

Capítulo 1

Introducción

El cálculo λ es estudiado usualmente desde un punto de vista matemático, ya sea formulado como una teoría ecuacional o como un sistema de reducción, esto implica que antes de poder trabajar con él, es necesario entender su formalización.

Este documento presenta una alternativa a esta manera de trabajar con el cálculo λ ya que describe la implementación de un sistema computacional en lugar de uno matemático. El sistema computacional `Lambda`, al ser un programa de computadora, puede ser utilizado para estudiar y trabajar con el cálculo λ de manera interactiva.

1.1 Programas literarios

Este documento no solo describe la implementación de `Lambda`, también *es* la implementación. La herramienta `noweb` para "programación literaria" genera tanto el documento como el código para `Lambda` a partir de un mismo archivo fuente. Este archivo consiste de pedazos intercalados de prosa y códigos etiquetados llamados *fragmentos*. Los fragmentos son escritos en un orden conveniente para describir el programa, es decir, el orden en el que aparecen en este documento, en lugar de un orden determinado por el lenguaje de programación de la implementación. El programa `noweave` toma el archivo fuente y produce un archivo en LaTeX, el cuál es compilado para producir este documento. El programa `notangle` extrae todo el código del archivo fuente y lo vacía en varios archivos en un orden adecuado para su compilación o interpretación.

Los fragmentos contienen código fuente o referencias a otros fragmentos. Los fragmentos de definiciones son precedidos por sus etiquetas entre paréntesis angulares. Por ejemplo, el código

1a ⟨*suma elementos del vector* v 1a⟩≡ 2▸
```
(define total
  (let itera ((suma 0)
              (i    0))
    (if (= i (vector-length v))
        suma
        (itera ⟨aumenta la suma total 1b⟩ (+ i 1)))))
```

Define:
`total`, utilizado en fragmento 2.

1b ⟨*aumenta la* suma *total* 1b⟩≡ (1a)
```
(+ suma (vector-ref v i))
```

suma los elementos del vector v. La utilización de fragmentos es mostrada como ⟨*aumenta la* suma *total* 1b⟩ del fragmento 1a. Varios fragmentos pueden tener la misma etiqueta; notangle concatena sus contenidos para producir un único fragmento. noweave identifica esta concatenación utilizando el símbolo +≡ en lugar de ≡ en fragmentos continuados:

2 ⟨*suma elementos del vector* v 1a⟩+≡ ◁1a

```
(printf "La suma de los elementos de ~a es ~a~%" v total)
```

Utiliza total 1a.

El programa notangle obtiene el código de un programa expandiendo un fragmento. Si en el contenido de este fragmento hay referencias a otros fragmentos, estos también son expandidos y así sucesivamente.

Los fragmentos incluyen descripciones de definición y referencia de identificadores en la parte inferior del código e indicadores de inclusión y continuación de fragmentos a la derecha de su etiqueta para auxiliar al lector en la navegación del documento, los números en las descripciones e indicadores se refieren a páginas del documento.

1.2 Cómo leer este documento

Lambda está escrito para leerse de inicio a fin, sin embargo algunas partes del programa son tediosas de leer y otras, a pesar de poderse considerar interesantes, no se relacionan propiamente con el cálculo λ, si no con aspectos relacionados con programación.

Ya que los que estamos familiarizadas con la programación acostumbramos leer una implementación en el orden impuesto por el lenguaje, es posible que cueste trabajo en un inicio entender el código presente en este documento, sin embargo, una manera de leer el código es identificando los archivos que son generados por notangle, estos tienen etiquetas con extensión .rkt y su definición establece los fragmentos de código que son parte del archivo.

Otra manera alternativa de leer este documento es hojearlo por completo prestando atención solo a la primer parte de cada capítulo para tener una idea global del programa. Posteriormente leer con más cuidado desde el inicio. Cualquiera que sea el método de lectura puede ser importante correr Lambda y explorarlo desde el punto de vista de un usuario.

1.3 Estructura del programa

La implementación de Lambda se divide en tres componentes:
lambda Una biblioteca con las estructuras y funciones asociadas al cálculo λ;
lambda-gui Una interfaz gráfica que permite estudiar el cálculo de manera interactiva;
lambda-init Una serie de procedimientos modificables por el usuario.

La componente referida como **lambda** contiene la implementación de los aspectos fundamentales del cálculo, por ejemplo, la definición de sus expresiones y procedimientos para leerlas, escribirlas y manipularlas.

La componente referida como **lambda-gui** contiene las estructuras y procedimientos para manejar las ventanas de la interfaz gráfica, la cuál hace uso de **lambda** para funcionar.

La componente referida como **lambda-init** contiene extensiones a **lambda** las cuales pueden ser modificadas y aumentadas por el usuario sin miedo a que el sistema deje de funcionar.

Capítulo 2

Piezas fundamentales

2.1 Objetos elementales

Existen diversas clases de objetos que pueden existir en el mundo del cálculo λ, estas clases de objetos representan ideas abstractas asociadas a cadenas de un lenguaje formal denotado Λ. Este lenguaje formal determina qué secuencias de símbolos son válidas expresar dentro del cálculo.

Definición de expresión Las clases de objetos cuya representación simbólica son parte de Λ son llamados *términos*, en este programa se implementan los tres tipos de términos más básicos del cálculo λ: las *variables*, las *abstracciones* y las *aplicaciones*. Las clases de objetos del sistema `Lambda` tienen como representación simbólica elementos del lenguaje formal Ω llamados *expresiones*, los elementos de Ω incluyen a todos los términos y a representaciones de otros dos tipos de objetos: los *huecos* y las *metainstrucciones*. El predicado `expresión?` determina si un objeto tiene representación simbólica en Ω.

3a ⟨*Definición de expresión* 3a⟩≡ (11)
```
(define/contract (expresión? x)
  (any/c . -> . boolean?)
  (or (variable? x) (abstracción? x) (aplicación? x)
      (hueco? x) (metainstrucción? x)))
```
Define:
  `expresión?`, utilizado en fragmentos 4–6, 8a, 13, 14, 17b, 19a, 20d, 23b, 27, 28, y 31a.
Utiliza `abstracción` 4c, `aplicación` 5a, `hueco` 5c, `metainstrucción` 6a, y `variable` 3b.

Definición de variable Las variables son una clase de objetos sencilla, se conforman de un *nombre* el cuál es usado para referirnos a ellas, por ejemplo, `Eduardo` es un nombre que puede referirse a una variable y usualmente diremos que `Eduardo` *es* la variable. Las variables son implementadas como una estructura compuesta de una cadena de caracteres. Utilizo el comando `define-struct/contract` para definir una estructura cuyo constructor verifica que el valor del campo `nombre` satisface el predicado `string?`.

3b ⟨*Definición de variable* 3b⟩≡ (11)
```
(define-struct/contract variable
  ([nombre string?])
  #:transparent
  #:mutable)
```

Define:
variable, utilizado en fragmentos 3–10, 12, 15–17, 19–21, 23, 26–28, 32a, 37, 47, y 57–60.

Al programar las variables de esta manera Racket define varios procedimientos: variable es un constructor que recibe una cadena de caracteres y regresa un objeto de este tipo; variable? es un predicado que recibe un valor y determina si es un valor de tipo variable; variable-nombre es un selector que dado un valor de tipo variable regresa la cadena de caracteres del campo nombre. La palabra clave #:transparent permite imprimir un valor de tipo variable de manera legible en el entorno interactivo de Racket, mientras que la palabra clave #:mutable hace que los campos muedan ser modificados, en este caso con el procedimiento set-variable-nombre!.

Para verificar que esta implementación tiene la funcionalidad esperada se define el caso de prueba "Implementación de variables".

4a ⟨*Casos de prueba* 4a⟩≡ (33) 4d▸
```
(test-case "Implementación de variables"
  (check-true (variable? (variable "x")))
  (check-false (variable? 42))
  (check-exn exn:fail:contract? (lambda () (variable 42)))
  (check-equal? (variable-nombre (variable "x")) "x"))
```
Utiliza variable 3b.

La definición de los casos de prueba se realiza utilizando los procedimientos de la biblioteca rackunit, la cual es invocada con el comando require.

4b ⟨*Dependencias de casos de prueba* 4b⟩≡ (33) 7b▸
```
(require rackunit)
```

Definición de abstracción Las abstracciones son clases de objetos compuestos que representan el concepto de *función*, se conforman de un *argumento* y un *cuerpo*, el argumento es una variable y el cuerpo es una expresión. El constructor de abstracciones verifica que las partes de la abstracción son del tipo adecuado por medio de los predicados variable? y expresión?.

4c ⟨*Definición de abstracción* 4c⟩≡ (11)
```
(define-struct/contract abstracción
  ([argumento variable?] [cuerpo expresión?])
  #:transparent
  #:mutable)
```
Define:
abstracción, utilizado en fragmentos 3–5, 7, 9, 12, 17–19, 21, 23, 26–28, 31a, 32a, 37, 47, y 57–60.
Utiliza expresión? 3a y variable 3b.

Se verifica el correcto funcionamiento de la implementación de las abstracciones con el caso de prueba "Implementación de abstracciones".

4d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂4a 5b▸
```
(test-case "Implementación de abstracciones"
  (check-true (abstracción? (abstracción (variable "x") (variable "y"))))
  (check-false (abstracción? (variable "x")))
  (check-exn exn:fail:contract? (lambda () (abstracción "x" "y")))
  (check-equal? (abstracción-argumento (abstracción (variable "x") (variable "y")))
                (variable "x"))
  (check-equal? (abstracción-cuerpo (abstracción (variable "x") (variable "y")))
```

```
                    (variable "y")))
```

Utiliza abstracción 4c y variable 3b.

Definición de aplicación Al igual que las abstracciones, las aplicaciones también son clases de objetos compuestos, sin embargo estos se conforman de dos expresiones cualesquiera llamadas *operador* y *operando*. El constructor de aplicaciones verifica que las partes de la aplicación son del tipo adecuado utilizando el predicado expresión?.

5a ⟨*Definición de aplicación* 5a⟩≡ (11)

```
(define-struct/contract aplicación
  ([operador expresión?] [operando expresión?])
  #:transparent
  #:mutable)
```

Define:
aplicación, utilizado en fragmentos 3a, 5–7, 9c, 10a, 12, 17, 18, 20, 21, 23, 26–28, 31a, 32a, 37, 47, y 57–60.
Utiliza expresión? 3a.

Se verifica que la implementación es correcta con el caso de prueba "Implementación de aplicaciones".

5b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂4d 5d▸

```
(test-case "Implementación de aplicaciones"
  (check-true (aplicación? (aplicación (variable "x") (variable "y"))))
  (check-false (aplicación? (abstracción (variable "x") (variable "y"))))
  (check-exn exn:fail:contract? (lambda () (aplicación "x" "y")))
  (check-equal? (aplicación-operador (aplicación (variable "x") (variable "y")))
                (variable "x"))
  (check-equal? (aplicación-operando (aplicación (variable "x") (variable "y")))
                (variable "y")))
```

Utiliza abstracción 4c, aplicación 5a, y variable 3b.

Definición de hueco Los huecos son una clase de objetos que representan información faltante. Las expresiones que contienen huecos son útiles únicamente para describir expresiones incompletas o expresiones cuya información faltante será descrita posteriormente. La implementación de los huecos consiste en una estructura sin campos.

5c ⟨*Definición de hueco* 5c⟩≡ (11)

```
(define-struct/contract hueco
  ()
  #:transparent)
```

Define:
hueco, utilizado en fragmentos 3a, 5d, 7d, 10c, 18, 21, 23, 26–28, 32a, 37, 47, y 59a.

La verificación de que esta implementación es correcta se realiza con el caso de prueba "Implementación de huecos" y es más sencilla que en las anteriores clases de objetos.

5d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂5b 6b▸

```
(test-case "Implementación de huecos"
  (check-true (hueco? (hueco)))
  (check-false (hueco? 42)))
```

Utiliza hueco 5c.

Definición de metainstrucción La última clase de objetos de las expresiones de Lambda es las metainstrucciones, estas representan operaciones que manipulan otras expresiones y pueden generar como resultado objetos que no cumplen con el predicado expresión?. La implementación de las metainstrucciones se divide en dos partes: primero la implementación de las expresiones que denotan una invocación de una metainstrucción y posteriormente la definición de una metainstrucción como procedimiento de Racket que se encarga de realizar un cómputo.

La invocación de metainstrucciones se conforma de un *nombre* y de una lista de expresiones llamada *argumentos*. El constructor de invocaciones de metainstrucciones verifica que sus campos son del tipo adecuado utilizando el predicado string? y (*list/c expresión?).

6a ⟨*Definición de metainstrucción* 6a⟩≡ (11) 6c▸

```
(define-struct/contract metainstrucción
  ([nombre string?] [argumentos (*list/c expresión?)])
  #:transparent
  #:mutable)
```

Define:
metainstrucción, utilizado en fragmentos 3a, 6, 7a, 10, 20, 21, 23, 26–28, 31a, 32a, 37, 47, 50, y 57–60.
Utiliza expresión? 3a.

El caso de prueba "Implementación de metainstrucciones" verifica que esta implementación funciona correctamente.

6b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂5d 7a▸

```
(test-case "Implementación de metainstrucciones"
  (check-true (metainstrucción? (metainstrucción "f" (list (variable "x")))))
  (check-false (metainstrucción? (aplicación (variable "x") (variable "y"))))
  (check-exn exn:fail:contract?
             (lambda () (metainstrucción (variable "f") (variable "x"))))
  (check-equal? (metainstrucción-nombre
                 (metainstrucción "f" (list (variable "x")))) "f")
  (check-equal? (metainstrucción-argumentos
                 (metainstrucción "f" (list (variable "x"))))
                (list (variable "x"))))
```

Utiliza aplicación 5a, metainstrucciones 6c, metainstrucción 6a, y variable 3b.

La definición de metainstrucciones consiste en la definición de un procedimiento de Racket y la asociación del nombre de la metainstrucción a este procedimiento en la tabla hash llamada metainstrucciones. Esta funcionalidad se implementa con el macro define-metainstrucción y en la sección 4 se muestra la implementación de varias metainstrucciones.

6c ⟨*Definición de metainstrucción* 6a⟩+≡ (11) ◂6a

```
(define metainstrucciones (make-hash))

(define-syntax define-metainstrucción
  (syntax-rules ()
    [(define-metainstrucción (nombre . argumentos)
       cómputo cómputos ...)
     (begin
      (define (nombre . argumentos)
        cómputo cómputos ...)
      (hash-set! metainstrucciones
                 (symbol->string 'nombre)
                 nombre))]))
```

Define:
  define-metainstrucción, utilizado en fragmentos 7a, 32a, 50, y 57–60.
  metainstrucciones, utilizado en fragmentos 6b, 7a, 10c, 23b, 31a, 50, y 51.
Utiliza metainstrucción 6a.

Para verificar que el macro define-metainstrucción tiene la funcionalidad esperada se define el caso de prueba "Definición de metainstrucciones".

7a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂6b 7d▸
```
(test-case "Definición de metainstrucciones"
  (check-equal? (syntax->datum (expand-only #'(define-metainstrucción (f x)
                                                (if (variable? x) 1 0))
                                              (list #'define-metainstrucción)))
                '(begin
                  (define (f x) (if (variable? x) 1 0))
                  (hash-set! metainstrucciones (symbol->string 'f) f))))
```
Utiliza define-metainstrucción 6c, metainstrucciones 6c, metainstrucción 6a, y variable 3b.

Para poder expander el macro en el caso de prueba es necesario hacer uso de la biblioteca macro-debugger/expand.

7b ⟨*Dependencias de casos de prueba* 4b⟩+≡ (33) ◂4b
```
(require macro-debugger/expand)
```

Definición de término Teniendo implementadas las estructuras con representación simbólica en Ω es conveniente definir un procedimiento para determinar si una expresión es un término. El predicado término? inspecciona una expresión para verificar que se conforma únicamente de variables, abstracciones o aplicaciones.

7c ⟨*Definición de término* 7c⟩≡ (11)
```
(define (término? x)
  (cond [(variable? x)    true]
        [(abstracción? x) (término? (abstracción-cuerpo x))]
        [(aplicación? x)  (and (término? (aplicación-operador x))
                               (término? (aplicación-operando x)))]
        [else             false]))
```
Define:
  término?, utilizado en fragmentos 7d, 8a, y 57a.
Utiliza abstracción 4c, aplicación 5a, y variable 3b.

El caso de prueba "Predicado para términos" se encarga de verificar que la funcionalidad de este procedimiento es correcta.

7d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂7a 8c▸
```
(test-case "Predicado para términos"
  (check-true
   (término? (abstracción (variable "x")
                          (aplicación (variable "x") (variable "x")))))
  (check-false
   (término? (abstracción (variable "x") (aplicación (variable "x") (hueco))))))
```
Utiliza abstracción 4c, aplicación 5a, hueco 5c, término? 7c, y variable 3b.

Comparación de expresiones El uso del mecanismo `define-struct` de `Racket` tiene la ventaja de que la verificación de que dos expresiones son del mismo tipo y tienen las mismas subexpresiones se puede realizar con el procedimiento genérico `equal?` y al igual que todos los objetos de cualquier tipo, la comparación `eq?` puede ser utilizada para determinar si dos objetos de cualquier tipo son el mismo objeto en memoria. Sin embargo, es útil implementar otros procedimientos que comparen aspectos particulares de dos expresiones, por ejemplo, si dos metainstrucciones tienen la misma cantidad de argumentos, si dos abstracciones tienen el mismo cuerpo, etc.

Se definen los predicados `expresión=?` y `término=?` como procedimientos que además de utilizar `equal?` verifican el tipo de valores que reciben.

8a ⟨*Comparación de expresiones* 8a⟩≡ (11)

```
(define/contract (expresión=? e1 e2)
  (expresión? expresión? . -> . boolean?)
  (equal? e1 e2))

(define/contract (término=? t1 t2)
  (término? término? . -> . boolean?)
  (equal? t1 t2))
```

Define:
`expresión=?`, utilizado en fragmento 9.
`término=?`, nunca utilizado.
Utiliza `expresión?` 3a y `término?` 7c.

Comparación de variables Dos variables con el mismo nombre son consideradas diferentes ya que la importancia de la variable radica en cómo es utilizada. Es por ello que es conveniente referirnos a todas las variables con el mismo nombre como "átomo" (ya que son objetos sin estructura, es decir, no se puede separar en partes). Los predicados `variable=?` y `átomo=?` permiten comparar dos variables de acuerdo a estos criterios.

8b ⟨*Comparación de variables* 8b⟩≡ (11)

```
(define/contract (variable=? v1 v2)
  (variable? variable? . -> . boolean?)
  (eq? v1 v2))

(define/contract (átomo=? v1 v2)
  (variable? variable? . -> . boolean?)
  (equal? v1 v2))
```

Define:
`variable=?`, utilizado en fragmento 8c.
`átomo=?`, utilizado en fragmentos 8c y 9a.
Utiliza `variable` 3b.

Para verificar el correcto funcionamiento de estos predicados se define el caso de prueba "`Predicados de variables`".

8c ⟨*Casos de prueba* 4a⟩+≡ (33) ◂7d 9b▸

```
(test-case "Predicados de variables"
  (let ([v (variable "x")])
    (check-true (variable=? v v)))
  (check-false (variable=? (variable "x") (variable "x")))
  (check-true (átomo=? (variable "x") (variable "x")))
  (check-false (átomo=? (variable "x") (variable "y"))))
```

Utiliza `variable` 3b, `variable=?` 8b, y `átomo=?` 8b.

Comparación de abstracciones En el caso de las abstracciones consideramos dos comparaciones: entre argumentos y entre cuerpos.

9a ⟨*Comparación de abstracciones* 9a⟩≡ (11)
```
(define/contract (argumento=? a1 a2)
  (abstracción? abstracción? . -> . boolean?)
  (átomo=? (abstracción-argumento a1)
           (abstracción-argumento a2)))

(define/contract (cuerpo=? a1 a2)
  (abstracción? abstracción? . -> . boolean?)
  (expresión=? (abstracción-cuerpo a1)
               (abstracción-cuerpo a2)))
```
Define:
argumento=?, utilizado en fragmento 9b.
cuerpo=?, utilizado en fragmento 9b.
Utiliza abstracción 4c, expresión=? 8a, y átomo=? 8b.

Para verificar el correcto funcionamiento de estos predicados se define el caso de prueba "Predicados de abstracciones".

9b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂8c 10a▸
```
(test-case "Predicados de abstracciones"
  (check-true (argumento=? (abstracción (variable "x") (variable "y"))
                           (abstracción (variable "x") (variable "z"))))
  (check-false (argumento=? (abstracción (variable "y") (variable "x"))
                            (abstracción (variable "z") (variable "x"))))
  (check-true (cuerpo=? (abstracción (variable "y") (variable "x"))
                        (abstracción (variable "z") (variable "x"))))
  (check-false (cuerpo=? (abstracción (variable "x") (variable "y"))
                         (abstracción (variable "x") (variable "z")))))
```
Utiliza abstracción 4c, argumento=? 9a, cuerpo=? 9a, y variable 3b.

Comparación de aplicaciones De manera similar a las abstracciones, se implementan predicados para verificar si dos aplicaciones comparten operador u operando.

9c ⟨*Comparación de aplicaciones* 9c⟩≡ (11)
```
(define/contract (operador=? a1 a2)
  (aplicación? aplicación? . -> . boolean?)
  (expresión=? (aplicación-operador a1)
               (aplicación-operador a2)))

(define/contract (operando=? a1 a2)
  (aplicación? aplicación? . -> . boolean?)
  (expresión=? (aplicación-operando a1)
               (aplicación-operando a2)))
```
Define:
operador=?, utilizado en fragmento 10a.
operando=?, utilizado en fragmento 10a.
Utiliza aplicación 5a y expresión=? 8a.

El caso de prueba "Predicados de aplicaciones" verifica que estos dos predicados sean correctos.

10a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂9b 10c▸
```
(test-case "Predicados de aplicaciones"
  (check-true (operador=? (aplicación (variable "x") (variable "y"))
                          (aplicación (variable "x") (variable "z"))))
  (check-false (operador=? (aplicación (variable "y") (variable "x"))
                           (aplicación (variable "z") (variable "x"))))
  (check-true (operando=? (aplicación (variable "y") (variable "x"))
                          (aplicación (variable "z") (variable "x"))))
  (check-false (operando=? (aplicación (variable "x") (variable "y"))
                           (aplicación (variable "x") (variable "z")))))
```
Utiliza aplicación 5a, operador=? 9c, operando=? 9c, y variable 3b.

Comparación de metainstrucciones Para las metainstrucciones se implementan tres predicados de comparación: comparación de nombres, comparación de cantidad de argumentos y comparación de argumentos.

10b ⟨*Comparación de metainstrucciones* 10b⟩≡ (11)
```
(define/contract (nombre=? m1 m2)
  (metainstrucción? metainstrucción? . -> . boolean?)
  (string=? (metainstrucción-nombre m1)
            (metainstrucción-nombre m2)))

(define/contract (cantidad-argumentos=? m1 m2)
  (metainstrucción? metainstrucción? . -> . boolean?)
  (= (length (metainstrucción-argumentos m1))
     (length (metainstrucción-argumentos m2))))

(define/contract (argumentos=? m1 m2)
  (metainstrucción? metainstrucción? . -> . boolean?)
  (equal? (metainstrucción-argumentos m1)
          (metainstrucción-argumentos m2)))
```
Define:
argumentos=?, utilizado en fragmento 10c.
cantidad-argumentos=?, utilizado en fragmento 10c.
nombre=?, utilizado en fragmento 10c.
Utiliza metainstrucción 6a.

El caso de prueba "Predicados de metainstrucciones" verifica que la implementación es correcta.

10c ⟨*Casos de prueba* 4a⟩+≡ (33) ◂10a 15b▸
```
(test-case "Predicados de metainstrucciones"
  (check-true (nombre=? (metainstrucción "f" null)
                        (metainstrucción "f" (list (variable "x")))))
  (check-false (nombre=? (metainstrucción "f" (list (variable "x")))
                         (metainstrucción "g" null)))
  (check-true (cantidad-argumentos=? (metainstrucción "f" (list (variable "x")))
                                     (metainstrucción "g" (list (hueco)))))
  (check-false (cantidad-argumentos=? (metainstrucción "f" (list (variable "x")))
                                      (metainstrucción "f" null)))
  (check-true (argumentos=? (metainstrucción "f" (list (variable "x")
                                                       (variable "y")))
```

```
                              (metainstrucción "g" (list (variable "x")
                                                         (variable "y")))))
    (check-false (argumentos=? (metainstrucción "f" (list (variable "x")))
                               (metainstrucción "f" (list (variable "x")
                                                          (variable "y"))))))
```

Utiliza `argumentos=?` 10b, `cantidad-argumentos=?` 10b, hueco 5c, `metainstrucciones` 6c, `metainstrucción` 6a, `nombre=?` 10b, y `variable` 3b.

Estructuras de las expresiones Los fragmentos de código presentados anteriormente conforman la implementación de las clases de objetos con representación simbólica en Ω, es decir, la implementación de las expresiones del sistema `Lambda`. El orden de los fragmentos de código es de la siguiente manera:

11 ⟨*Estructuras de las expresiones* 11⟩≡ (32b)
- ⟨*Definición de expresión* 3a⟩
- ⟨*Definición de término* 7c⟩
- ⟨*Comparación de expresiones* 8a⟩
- ⟨*Definición de variable* 3b⟩
- ⟨*Comparación de variables* 8b⟩
- ⟨*Definición de abstracción* 4c⟩
- ⟨*Comparación de abstracciones* 9a⟩
- ⟨*Definición de aplicación* 5a⟩
- ⟨*Comparación de aplicaciones* 9c⟩
- ⟨*Definición de hueco* 5c⟩
- ⟨*Definición de metainstrucción* 6a⟩
- ⟨*Comparación de metainstrucciones* 10b⟩

2.2 Lenguaje de expresiones

De Ω a expresiones y viceversa Tener implementadas las estructuras de las expresiones es lo único necesario para poder escribir algoritmos que analicen y manipulen expresiones en el sistema `Lambda`, sin embargo es inconveniente definir expresiones en particular y leer expresiones que sean resultado de algún algoritmo.

Como se menciona al inicio de esta sección las estructuras implementadas son representaciones computacionales de elementos del lenguaje formal Ω, estos elementos son cadenas de símbolos los cuáles son más amenos de escribir y leer que utilizando los constructores y representaciones textuales de las estructuras programadas en `Racket`.

La representación simbólica de una variable con nombre "`x`" es x, en el caso de una abstracción de la forma `(abstracción x M)` su representación simbólica es la cadena $(\lambda x.M)$ donde x y M son las representaciones simbólicas de `x` y `M`, de manera similar, una aplicación de la forma `(aplicación M N)` tiene como representación simbólica $(M\,N)$. Estas notaciones son estándar en el estudio formal del cálculo λ.

En el caso de los huecos y las metainstrucciones, tomaremos una notación simbólica inventada, para un hueco será `[ ]` y para una metainstrucción de la forma `(metainstrucción f (list M1 ... Mn))` será $f[M_1, ..., M_n]$.

Se define el lenguaje libre de contexto Ω de manera informal de la siguiente manera. Sea $D = \{`(', `)', `[', `]', `.', `,', `\lambda', ` '\}$ un conjunto de símbolos delimitadores donde ` ` es el símbolo de espacio en blanco y sea N el conjunto de nombres compuestos de uno o más

símbolos $a \notin D$

$$\begin{aligned} x &\in \Omega && \text{Si } x \in N \\ (\lambda x.P) &\in \Omega && \text{Si } x \in N \text{ y } P \in \Omega \\ (P\,Q) &\in \Omega && \text{Si } P,\ Q \in \Omega \\ [\quad] &\in \Omega \\ x[P_1, P_2, ..., P_n] &\in \Omega && \text{Si } x \in N \text{ y } P_i \in \Omega \end{aligned}$$

Conociendo la sintaxis de los elementos de Ω podemos comparar representaciones diferentes de una misma expresión, por ejemplo, un término muy famoso es el *combinador Y*, simbólicamente es escrito:

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

mientras que su representación con las estructuras que se implementaron es:

12 ⟨*El combinador Y* 12⟩≡

```
(abstracción (variable "f")
             (aplicación (abstracción (variable "x")
                                      (aplicación (variable "f")
                                                  (aplicación (variable "x")
                                                              (variable "x"))))
                         (abstracción (variable "x")
                                      (aplicación (variable "f")
                                                  (aplicación (variable "x")
                                                              (variable "x"))))))
```

Utiliza abstracción 4c, aplicación 5a, y variable 3b.

También se contemplan abusos de notación a la representación simbólica, las cuales son:

$$(\lambda x_1.(\lambda x_2.(...(\lambda x_n.P)...))) \rightarrowtail \lambda x_1\,x_2\,...\,x_n.P$$
$$((...(P_1\,P_2)...)\,P_n) \rightarrowtail P_1\,P_2\,...\,P_n$$

Esto nos permite escribir al combinador Y, de manera más breve como

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

Sintaxis del lenguaje Para definir de manera rigurosa las cadenas de símbolos que son elementos de Ω y el abuso de notación se implementa un *analizador sintáctico* (o *parser*) y un *sintetizador sintáctico* (o *escritor*).

La tarea del parser consiste en transformar una cadena de símbolos en su estructrura correspondiente, en esta implementación los símbolos son caracteres leídos de un puerto de entrada el cuál puede extraer la información de un archivo en disco o de una cadena de caracteres. Ya que no todos los lenguajes de programación tienen la capacidad de manejar letras griegas como 'λ' se considera el caracter '\' como un reemplazo de la letra lambda. Si se dispone de una codificación de caracteres extendida como Unicode, entonces el caracter lambda (con código 955) puede ser utilizado también.

La tarea del escritor consiste en transformar una estructura a su cadena de símbolos correspondiente, ya que la cadena de símbolos puede o no usar el abuso de notación, se implementan dos algoritmos de escritura: uno formal y otro breve.

Una consideración del lenguaje de Lambda que no es explícitamente mencionada en la definición de arriba es que el símbolo '`;`' es utilizado para separar expresiones.

La implementación de la sintaxis del lenguaje tiene la siguiente forma:

13a ⟨*Sintaxis del lenguaje* 13a⟩≡ (32b)
⟨*Configuración de la sintaxis* 13b⟩
⟨*Parseo del lenguaje* 13c⟩
⟨*Escritura del lenguaje* 22⟩

Configuración de la sintaxis Tanto el parser como el escritor deben de poder manejar el caso en que la codificación del archivo o cadena a leer o escribir no soporte Unicode, la variable hay-lambda? debe de tener un valor booleano que indique si se puede leer y escribir el caracter 'λ' con código 955. De igual manera la variable símbolo-lambda debe ser el valor del caracter lambda.

13b ⟨*Configuración de la sintaxis* 13b⟩≡ (13a)
```
(define hay-lambda? true)

(define/contract (símbolo-lambda? x)
  (char? . -> . boolean?)
  (and hay-lambda? (char=? (integer->char 955) x)))
```
Define:
hay-lambda?, utilizado en fragmentos 24c y 29.
símbolo-lambda?, utilizado en fragmentos 18d y 19b.

2.2.1 Analizador sintáctico

Parseo del lenguaje El parser es implementado con un algorimo *descendente recursivo* el cuál toma como argumento un puerto de entrada de Racket y construye la estructura correspondiente a la cadena de símbolos leída. La estructura de la implementación del parser es la siguiente:

13c ⟨*Parseo del lenguaje* 13c⟩≡ (13a)
⟨*Procedimientos auxiliares del parser* 15c⟩
⟨*Algoritmo del parser* 13d⟩
⟨*Punto de entrada del parser* 14a⟩

El procedimiento que implementa el algoritmo para analizar los caracteres del puerto de entrada es parsear-expresión, este procedimiento recibe un puerto de entrada p, un caracter c (que usualmente es el primer caracter de p) y un acumulador de expresiones a cuyo valor es la expresión parcialmente parseada; el valor resultante es una expresión en caso de éxito, un objeto "fin de archivo" cuando no hay símbolos que leer del puerto o un caracter cuando no se pudo parsear una expresión.

13d ⟨*Algoritmo del parser* 13d⟩≡ (13c)
```
(define/contract (parsear-expresión p c a)
  (input-port? (or/c char? eof-object?) (or/c expresión? null?)
               . -> . (or/c expresión? eof-object? char?))
  (cond ⟨Casos del algoritmo de parseo 14c⟩))
```
Define:
parsear-expresión, utilizado en fragmentos 14–20.
Utiliza expresión? 3a y parsear 14a.

El procedimiento parsear es el punto de entrada del parser, se encarga de intentar parsear una expresión utilizando el procedimiento parsear-expresión y emitir un error en caso que los símbolos del puerto no hayan podido conformar un elemento de Ω válido.

14a ⟨*Punto de entrada del parser* 14a⟩≡ (13c) 14b▸
```
(define/contract (parsear p)
  (input-port? . -> . (or/c expresión? eof-object?))
  (let ([e (parsear-expresión p (peek-char p) null)])
    (cond [(eof-object? e) e]
          [(expresión? e)  e]
          [(char=? #\; e)  (parsear p)]
          [else
           (error 'parsear "No fue posible parsear una expresión válida")])))
```
Define:
parsear, utilizado en fragmentos 13–21, 25b, 30–32, 35, 51, y 59b.
Utiliza expresión? 3a y parsear-expresión 13d.

Un procedimiento que resulta útil para hacer pruebas simples de parseo es parsear-cadena el cuál se encarga de convertir una cadena de caracteres en un puerto de entrada y parsear el puerto de entrada.

14b ⟨*Punto de entrada del parser* 14a⟩+≡ (13c) ◂14a
```
(define/contract (parsear-cadena s)
  (string? . -> . (or/c expresión? eof-object?))
  (parsear (open-input-string s)))
```
Define:
parsear-cadena, utilizado en fragmentos 15, 16, 21, 25b, 30–32, 35, 51, y 59b.
Utiliza expresión? 3a y parsear 14a.

Casos del algoritmo de parseo El procedimiento parsear-expresión comienza conociendo el primer caracter c del puerto de entrada p, se realiza un análisis de casos con cond para determinar la estructura sintáctica de la que c forma parte tomando en cuenta el parseo parcial acumulado a.

Antes de abordar cada caso es importante precisar el rol que juega el acumulador a. Ya que el parser debe ser capaz de construír una estructura a partir de una representación simbólica formal, parcialmente formal o con la mayor cantidad de abuso de notación posible, es necesario "recordar" la forma de las aplicaciones que se han leído, por ejemplo una expresión de la forma $(P_1\ P_2\ P_3)$ es formalmente $((P_1\ P_2)\ P_3)$, sin embargo, la lectura de izquierda a derecha nos obliga a leer primero P_1, acumular su estructura y al ver que es posible leer a P_2 construír la aplicación $(P_1\ P_2)$ y acumularla para intentar leer la siguiente expresión P_3 y formar la aplicación final.

Cuando se termina de parsear Hay dos situaciones en donde se debe terminar de parsear una expresión, la primera es cuando se lee un objeto "fin de archivo" y la otra es cuando se lee un *delimitador final*.

Si c satisface el predicado eof-object? es cuando ya no es posible extraer más caracteres de p.

14c ⟨*Casos del algoritmo de parseo* 14c⟩≡ (13d) 15e▸
```
[(eof-object? c)
 ⟨Se leyó un eof-object 15a⟩]
```

En este caso se debe regresar este objeto "fin de archivo" a menos que el valor del acumulador a no sea nulo, cuando a no es nulo es una expresión correctamente parseada, por lo tanto debe ser el resultado del parseo.

15a ⟨*Se leyó un* `eof-object` 15a⟩≡ (14c)
```
(if (null? a) c a)
```

El caso de prueba "`Lectura de fin de archivo`" verifica que este caso se maneje correctamente.

15b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂10c 15d▸
```
(test-case "Lectura de fin de archivo"
  (check-equal? (parsear-cadena "") eof)
  (check-equal? (parsear-expresión (open-input-string "") eof (variable "x"))
                (variable "x")))
```
Utiliza `parsear` 14a, `parsear-cadena` 14b, `parsear-expresión` 13d, y `variable` 3b.

En caso que el parser se encuentre un delimitador final el algoritmo debe regresar, los delimitadores finales son aquellos objetos que delimitan expresiones por la derecha. El predicado `delimitador-final?` determina si un objeto es delimitador final.

15c ⟨*Procedimientos auxiliares del parser* 15c⟩≡ (13c) 17a▸
```
(define/contract (delimitador-final? x)
  ((or/c char? eof-object?) . -> . boolean?)
  (or (eof-object? x) (char=? #\] x) (char=? #\) x)
      (char=? #\. x)  (char=? #\, x) (char=? #\; x)))
```
Define:
`delimitador-final?`, utilizado en fragmentos 15 y 19b.

El caso de prueba "`Predicado para delimitadores`" verifica que este procedimiento sea correcto.

15d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂15b 16a▸
```
(test-case "Predicados para delimitadores"
  (check-true (delimitador-final? eof))
  (check-true (delimitador-final? #\)))
  (check-false (delimitador-final? #\x))
  (check-false (delimitador-final? #\())
  (check-exn exn:fail:contract? (lambda () (delimitador-final? 42))))
```
Utiliza `delimitador-final?` 15c.

Si `c` satisface el predicado `delimitador-final?` el algoritmo ha terminado de parsear.

15e ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂14c 16b▸
```
[(delimitador-final? c)
 ⟨Se leyó un delimitador final 15f⟩]
```
Utiliza `delimitador-final?` 15c.

De manera similar al caso anterior el resultado depende de `a`, cuando es nulo se debe consumir el delimitador del puerto de entrada y regresarse como resultado, de lo contrario, el delimitador debe permanecer en `p` y el resultado es la expresión acumulada.

15f ⟨*Se leyó un delimitador final* 15f⟩≡ (15e)
```
(if (null? a) (read-char p) a)
```

El caso de prueba "Lectura de delimitador final" verifica que este caso se maneje correctamente.

16a ⟨*Casos de prueba* 4a⟩+≡ (33) ◃15d 16d▹
```
(test-case "Lectura de delimitador final"
  (check-exn exn:fail? (lambda () (parsear-cadena ")")))
  (check-equal? (parsear-expresión (open-input-string ")") #\) null) #\))
  (let ([p (open-input-string ")")])
    (check-equal? (parsear-expresión p #\) (variable "x")) (variable "x"))
    (check-equal? (read-char p) #\)))
```
Utiliza parsear 14a, parsear-cadena 14b, parsear-expresión 13d, y variable 3b.

Cuando se lee espacio en blanco Los espacios en blanco, también conocidos como caracteres "whitespace" son ignorador por el parser. La implementación de este caso hace uso del predicado char-whitespace? de Racket para determinar si c es un espacio en blanco, este predicado reconoce al menos los caracteres espacio, tab y nueva línea.

16b ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◃15e 16e▹
```
[(char-whitespace? c)
 ⟨Se leyó un espacio en blanco 16c⟩]
```

Para ignorar a c y seguir parseando expresiones se debe consumir el caracter del puerto p e invocar de manera recursiva parsear-expresión.

16c ⟨*Se leyó un espacio en blanco* 16c⟩≡ (16b)
```
(read-char p)
(parsear-expresión p (peek-char p) a)
```
Utiliza parsear 14a y parsear-expresión 13d.

El caso de prueba "Lectura de espacio en blanco" verifica que este caso se maneje correctamente.

16d ⟨*Casos de prueba* 4a⟩+≡ (33) ◃16a 17c▹
```
(test-case "Lectura de espacio en blanco"
  (check-equal? (parsear-cadena " \t\n \n\t")
                (parsear-cadena ""))
  (check-equal? (parsear-expresión (open-input-string " \t\n \n\t)")
                                   #\space
                                   (variable "x"))
                (parsear-expresión (open-input-string ")") #\) (variable "x"))))
```
Utiliza parsear 14a, parsear-cadena 14b, parsear-expresión 13d, y variable 3b.

Cuando se parsea una subexpresión En el análisis de casos inicial del algoritmo, la única manera de comenzar a parsear una subexpresión es leyendo el caracter de paréntesis abierto.

16e ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◃16b 18a▹
```
[(char=? #\( c)
 ⟨Se leyó un paréntesis abierto 17d⟩]
```

En este caso se debe consumir el paréntesis abierto del puerto p, obtener la expresión dentro del paréntesis con un procedimiento auxiliar parsear-paréntesis e invocar de manera recursiva a parsear-expresión.

El procedimiento parsear-paréntesis recibe un puerto de entrada e invoca a parsear-expresión, eventualmente el parseo termina al encontrar un paréntesis cerrado, en cuyo caso se debe verificar que la expresión leída sea una abstracción o una aplicación, ya que estas son las únicas expresiones que pueden ir dentro de un paréntesis.

17a ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂15c 17b▸

```
(define/contract (parsear-paréntesis p)
  (input-port? . -> . (or/c abstracción? aplicación?))
  (let ([e (parsear-expresión p (peek-char p) null)])
    (unless (char=? #\) (read-char p))
      (error 'parsear-paréntesis "Se esperaba leer el caracter ‘)’"))
    e))
```

Define:
parsear-paréntesis, utilizado en fragmento 17d.
Utiliza abstracción 4c, aplicación 5a, parsear 14a, y parsear-expresión 13d.

La invocación recursiva a parsear-expresión debe de realizarse con cuidado, considerando a e la expresión resultante de parsear-paréntesis, si el valor del acumulador es nulo, la llamada recursiva se hace con e como acumulador, de lo contrario se deberá acumular la aplicación de a con e. Esta verificación se realiza con el procedimiento auxiliar aplicación/identidad.

17b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂17a 18b▸

```
(define/contract (aplicación/identidad a e)
  ((or/c expresión? null?) expresión? . -> . expresión?)
  (if (null? a) e (aplicación a e)))
```

Define:
aplicación/identidad, utilizado en fragmentos 17, 18, y 20c.
Utiliza aplicación 5a y expresión? 3a.

El caso de prueba “Acumulación de expresiones” se encarga de verificar el correcto funcionamiento de este procedimiento.

17c ⟨*Casos de prueba* 4a⟩+≡ (33) ◂16d 21a▸

```
(test-case "Acumulación de expresiones"
  (check-equal? (aplicación/identidad null (variable "x"))
                (variable "x"))
  (check-equal? (aplicación/identidad (variable "x") (variable "y"))
                (aplicación (variable "x") (variable "y"))))
```

Utiliza aplicación 5a, aplicación/identidad 17b, y variable 3b.

Con estos procedimientos auxiliares implementados se define el caso de leer un paréntesis abierto.

17d ⟨*Se leyó un paréntesis abierto* 17d⟩≡ (16e)

```
(read-char p)
(let ([e (parsear-paréntesis p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```

Utiliza aplicación 5a, aplicación/identidad 17b, parsear 14a, parsear-expresión 13d, y parsear-paréntesis 17a.

Cuando se parsea un hueco En el análisis de casos inicial del algoritmo, la única manera de comenzar a parsear un hueco es leyendo el caracter de corchete abierto, este caracter también forma parte de la representación simbólica de las metainstrucciones, sin embargo, en el caso de las metainstrucciones un nombre tiene que ser leído antes del caracter '`[`'.

18a ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂16e 18d▸
```
[(char=? #\[ c)
 ⟨Se leyó un corchete abierto 18c⟩]
```

En este caso se hace uso del procedimiento auxiliar `parsear-hueco`, el cuál admite la lectura de cero o más espacios en blanco antes de encontrarse con un corchete cerrado.

18b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂17b 19a▸
```
(define/contract (parsear-hueco p)
  (input-port? . -> . hueco?)
  (let ([c (peek-char p)])
    (cond [(char-whitespace? c)
           (read-char p)
           (parsear-hueco p)]
          [(char=? #\] c)
           (read-char p)
           (hueco)]
          [else
           (error 'parsear-hueco "Se esperaba leer el caracter ']'")])))
```
Define:
`parsear-hueco`, utilizado en fragmento 18c.
Utiliza `hueco` 5c y `parsear` 14a.

La implementación del caso consume el caracter '`[`' del puerto `p` y acumula el hueco parseado con `parsear-hueco` en una invocación recursiva a `parsear-expresión`.

18c ⟨*Se leyó un corchete abierto* 18c⟩≡ (18a)
```
(read-char p)
(let ([e (parsear-hueco p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Utiliza `aplicación` 5a, `aplicación/identidad` 17b, `hueco` 5c, `parsear` 14a, `parsear-expresión` 13d, y `parsear-hueco` 18b.

Cuando se lee una lambda En la representación simbólica de las abstracciones, los paréntesis externos pueden omitirse por abuso de notación, por lo que un símbolo inicial válido para una abstracción es el caracter 'λ'.

18d ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂18a 20a▸
```
[(or (símbolo-lambda? c) (char=? #\\ c))
 ⟨Se leyó una lambda 18e⟩]
```
Utiliza `símbolo-lambda?` 13b.

La implementación de este caso es similar a los anteriores dos, se consume el caracter leído, se utiliza un procedimiento auxiliar para parsear la abstracción y se invoca de manera recursiva `parsear-expresión` acumulando la abstracción resultante.

18e ⟨*Se leyó una lambda* 18e⟩≡ (18d)
```
(read-char p)
(let ([e (parsear-abstracción p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Utiliza `abstracción` 4c, `aplicación` 5a, `aplicación/identidad` 17b, `parsear` 14a, `parsear-abstracción` 19a, y `parsear-expresión` 13d.

El procedimiento auxiliar parsear-abstracción es un poco más complejo que los anteriores procedimientos auxiliares. Realiza una iteración acumulando las variables leídas cuidando los casos en que se lea un delimitador antes que el caracter '.' que separa los argumentos del cuerpo e ignorando los espacios en blanco de variable a variable. Cuando el punto separador es leído, se parsea el cuerpo de la abstracción y se construye la estructura completa aplicando un algoritmo de "currificación" con las variables acumuladas.

19a ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂18b 19b▸

```
(define/contract (parsear-abstracción p)
  (input-port? . -> . abstracción?)
  (let itera ([c  (peek-char p)]
              [vs null])
    (cond [(and (char? c) (char-whitespace? c))
           (read-char p)
           (itera (peek-char p) vs)]
          [(and (char? c) (char=? #\. c))
           (read-char p)
           (if (null? vs)
               (error 'parsear-abstracción "Se esperaba al menos una variable")
               (let ([e (parsear-expresión p (peek-char p) null)])
                 (if (expresión? e)
                     (foldr abstracción e (reverse vs))
                     (error 'parsear-abstracción
                            "Se esperaba leer el cuerpo de una abstracción"))))]
          [(delimitador? c)
           (error 'parsear-abstracción "Se esperaba leer una abstracción completa")]
          [else
           (let ([v (variable (parsear-nombre p))])
             (itera (peek-char p) (cons v vs)))])))
```

Define:
parsear-abstracción, utilizado en fragmento 18e.
Utiliza abstracción 4c, delimitador? 19b, expresión? 3a, parsear 14a, parsear-expresión 13d, parsear-nombre 20b, y variable 3b.

El predicado delimitador? es implementado de manera similar a delimitador-final? solo que contempla todos los objetos que delimitan expresiones, incluyendo los delimitadores ignorados de espacios en blanco.

19b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂19a 20b▸

```
(define/contract (delimitador? x)
  ((or/c char? eof-object?) . -> . boolean?)
  (or (delimitador-final? x)
      (char=? #\( x) (char=? #\[ x)
      (símbolo-lambda? x) (char=? #\\ x)
      (char-whitespace? x)))
```

Define:
delimitador?, utilizado en fragmentos 19a y 20b.
Utiliza delimitador-final? 15c y símbolo-lambda? 13b.

En el parseo de abstracciones se hace uso del procedimiento parsear-nombre, este es descrito en el siguiente caso.

Cuando se lee un nombre El último caso del análisis de casos es leer un caracter que no es delimitador, lo cuál significa que constituye un nombre.

20a ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂18d
```
[else
 ⟨Se leyó un constituyente 20c⟩]
```

Para leer un nombre se consumen y acumulan caracteres constituyentes (no-delimitadores) hasta encontrar algún otro caracter. El procedimiento `parsear-nombre` regresa una cadena con los caracteres consumidos del puerto `p`.

20b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂19b 20d▸
```
(define/contract (parsear-nombre p)
  (input-port? . -> . string?)
  (let itera ([c  (peek-char p)]
              [cs null])
    (if (delimitador? c)
        (if (null? cs)
            (error 'parsear-nombre "Se intentó leer un nombre de longitud cero")
            (list->string (reverse cs)))
        (begin
         (read-char p)
         (itera (peek-char p) (cons c cs))))))
```
Define:
`parsear-nombre`, utilizado en fragmentos 19a y 20c.
Utiliza `delimitador?` 19b, `longitud` 57a, y `parsear` 14a.

En la implementación del caso, se parsea un nombre y se determina el tipo de resultado, cuando el caracter después del nombre es '`[`' el resultado es una metainstrucción, de lo contrario es una variable.

20c ⟨*Se leyó un constituyente* 20c⟩≡ (20a)
```
((lambda (e)
   (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
 (let* ([x (parsear-nombre p)]
        [c (peek-char p)])
   (cond [(and (char? c) (char=? #\[ c))
          (read-char p)
          (metainstrucción x (parsear-corchetes p))]
         [else
          (variable x)])))
```
Utiliza `aplicación` 5a, `aplicación/identidad` 17b, `metainstrucción` 6a, `parsear` 14a, `parsear-corchetes` 20d, `parsear-expresión` 13d, `parsear-nombre` 20b, y `variable` 3b.

En el caso que el nombre sea de una metainstrucción, se parsea la lista de argumentos con el procedimiento auxiliar `parsear-corchetes`.

20d ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂20b
```
(define/contract (parsear-corchetes p)
  (input-port? . -> . (*list/c expresión?))
  (let itera ([e  (parsear-expresión p (peek-char p) null)]
              [es null])
    (cond [(eof-object? e)
           (error 'parsear-corchetes
                  "Se esperaba leer una metainstrucción completa")]
          [(char? e)
           (case e
             [(#\]) (reverse es)]
             [(#\,) (itera (parsear-expresión p (peek-char p) null) es)]
```

```
            [else
             (error 'parsear-corchetes
                    "Se esperaba leer una metainstrucción completa")])]
          [else
           (itera (parsear-expresión p (peek-char p) null)
                  (cons e es))])))
```

Define:
parsear-corchetes, utilizado en fragmento 20c.
Utiliza expresión? 3a, metainstrucción 6a, parsear 14a, y parsear-expresión 13d.

Para verificar el correcto funcionamiento del parser se definen dos casos de prueba para el parseo de representaciones simbólicas con y sin abuso de notación.

21a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂17c 21b▸

```
(test-case "Parsear sin abuso de notación"
  (check-equal? (parsear-cadena "foo")
                (variable "foo"))
  (check-equal? (parsear-cadena "(foo bar)")
                (aplicación (variable "foo") (variable "bar")))
  (check-equal? (parsear-cadena "((foo bar) baz)")
                (aplicación (aplicación (variable "foo") (variable "bar"))
                            (variable "baz")))
  (check-equal? (parsear-cadena "(foo (bar baz))")
                (aplicación (variable "foo")
                            (aplicación (variable "bar") (variable "baz"))))
  (check-equal? (parsear-cadena "(\\bar.bar)")
                (abstracción (variable "bar") (variable "bar")))
  (check-equal? (parsear-cadena "foo[bar,baz]")
                (metainstrucción "foo" (list (variable "bar") (variable "baz"))))
  (check-equal? (parsear-cadena "(\\foo.(\\bar.foo))")
                (abstracción (variable "foo")
                             (abstracción (variable "bar") (variable "foo"))))
  (check-equal? (parsear-cadena "  []  ")
                (hueco)))
```

Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, parsear 14a, parsear-cadena 14b, y variable 3b.

El caso de prueba para parsear expresiones escritas con abuso de notación debe ser más exhaustivo ya que el abuso de notación puede ser parcial o total.

21b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂21a 25b▸

```
(test-case "Parsear con abuso de notación"
  (check-equal? (parsear-cadena "foo bar")
                (aplicación (variable "foo") (variable "bar")))
  (check-equal? (parsear-cadena "foo bar baz")
                (aplicación (aplicación (variable "foo") (variable "bar"))
                            (variable "baz")))
  (check-equal? (parsear-cadena "foo (bar baz)")
                (aplicación (variable "foo")
                            (aplicación (variable "bar") (variable "baz"))))
  (check-equal? (parsear-cadena "\\foo.foo")
                (abstracción (variable "foo") (variable "foo")))
  (check-equal? (parsear-cadena "\\foo bar.foo")
                (abstracción (variable "foo")
                             (abstracción (variable "bar") (variable "foo"))))
  (check-equal? (parsear-cadena "(\\foo bar.foo)")
```

```
                (abstracción (variable "foo")
                             (abstracción (variable "bar") (variable "foo"))))
    (check-equal? (parsear-cadena "(\\foo.\\bar.foo)")
                (abstracción (variable "foo")
                             (abstracción (variable "bar")
                                          (variable "foo"))))
    (check-equal? (parsear-cadena "\\foo.bar baz")
                (abstracción (variable "foo")
                             (aplicación (variable "bar")
                                         (variable "baz"))))
    (check-equal? (parsear-cadena "\\foo bar.bar baz")
                (abstracción (variable "foo")
                             (abstracción (variable "bar")
                                          (aplicación (variable "bar")
                                                      (variable "baz")))))
    (check-equal? (parsear-cadena "(\\foo.foo)\\bar.bar")
                (aplicación (abstracción (variable "foo")
                                         (variable "foo"))
                            (abstracción (variable "bar")
                                         (variable "bar"))))
    (check-equal? (parsear-cadena "foo[\\bar.bar bar, \\baz.baz baz] quux")
                (aplicación
                 (metainstrucción "foo"
                                  (list (abstracción (variable "bar")
                                                     (aplicación (variable "bar")
                                                                 (variable "bar")))
                                        (abstracción (variable "baz")
                                                     (aplicación (variable "baz")
                                                                 (variable "baz")))))
                 (variable "quux")))
    (check-equal? (parsear-cadena "\\x.x [] x")
                (abstracción (variable "x")
                             (aplicación (aplicación (variable "x") (hueco))
                                         (variable "x")))))
```

Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, parsear 14a, parsear-cadena 14b, y variable 3b.

2.2.2 Sintetizador sintáctico

Escritura del lenguaje Para escribir estructuras de expresiones se identifican dos algoritmos básicos: uno para escritura formal y otro para escritura breve. Estos algoritmos se basan en un procedimiento de *recorrido a lo profundo* en *postorden* para transformar la estructura de una expresión a otro tipo de objetos, como por ejemplo cadenas de caracteres. Es importante implementar estos algoritmos de la manera más general posible, esto con la finalidad de poder definir escritores para diversos "back-ends" como cadenas de caracteres en texto plano o código del modo de matemáticas de LaTeX. La estructura de esta implementación es la siguiente:

22 ⟨*Escritura del lenguaje* 22⟩≡ (13a)
⟨*Algoritmos principales de escritura* 23a⟩
⟨*Escritores usuales* 24a⟩
⟨*Escritores misceláneos* 25c⟩

Constructor de escritores formales La técnica utilizada para implementar estos algoritmos genéricos es definir un constructor de escritores que reciba como argumentos procedimientos específicos para escribir cada clase de estructura. Consideremos primero el constructor de escritores formales llamado `escritor-formal`, toma cinco procedimientos como argumentos, uno por cada tipo de estructura y regresa un escritor concreto.

23a ⟨*Algoritmos principales de escritura* 23a⟩≡ (22) 27▸
```
(define/contract (escritor-formal e-variable e-abstracción e-aplicación
                                  e-hueco e-metainstrucción)
  ⟨Contrato para escritores formales 23b⟩
  ⟨Algoritmo de escritura formal 23c⟩)
```
Define:
`escritor-formal`, utilizado en fragmentos 24–26 y 47.
Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, y variable 3b.

El contrato de este constructor es que recibe cinco procedimientos y regresa un procedimiento que recibe una expresión y regresa algún objeto. Cada procedimiento de argumento corresponde en orden a escritor de variables, abstracciones, aplicaciones, huecos y metainstrucciones de tal manera que cada uno de estos procedimientos recibe de argumento la esctructura de la expresión correspondiente y los campos de dicha estructura.

23b ⟨*Contrato para escritores formales* 23b⟩≡ (23a)
```
((variable? string? . -> . any/c)                 ; Escritor de variables
 (abstracción? variable? expresión? . -> . any/c) ; Escritor de abstracciones
 (aplicación? expresión? expresión? . -> . any/c) ; Escritor de aplicaciones
 (hueco? . -> . any/c)                            ; Escritor de huecos
 (metainstrucción? string? (*list/c expresión?)   ; Escritor de metainstrucciones
                   . -> . any/c)
 . -> . (expresión? . -> . any/c))      ; Escritor de expresiones especializado
```
Utiliza abstracción 4c, aplicación 5a, expresión? 3a, hueco 5c, metainstrucciones 6c, metainstrucción 6a, y variable 3b.

El algoritmo en sí tiene la tarea de identificar el escritor especializado que invocar y extraer los campos de la expresión `e`.

23c ⟨*Algoritmo de escritura formal* 23c⟩≡ (23a)
```
(lambda (e)
  (cond [(variable? e)
         (e-variable e (variable-nombre e))]
        [(abstracción? e)
         (e-abstracción e (abstracción-argumento e) (abstracción-cuerpo e))]
        [(aplicación? e)
         (e-aplicación e (aplicación-operador e) (aplicación-operando e))]
        [(hueco? e)
         (e-hueco e)]
        [(metainstrucción? e)
         (e-metainstrucción e (metainstrucción-nombre e)
                            (metainstrucción-argumentos e))]))
```
Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, y variable 3b.

Escritor formal a texto plano La definición de un escritor formal a una cadena de caracteres en texto plano es relativamente fácil. El objeto que regresan los procedimientos

especializados es una cadena de caracteres y la escritura de estructuras complejas consiste en concatenar los resultados de la escritura de sus partes.

24a ⟨*Escritores usuales* 24a⟩≡ (22) 29▸
```
(define escribir-expresión-formal
  (escritor-formal
   ⟨Escribir formalmente variable a texto plano 24b⟩
   ⟨Escribir formalmente abstracción a texto plano 24c⟩
   ⟨Escribir formalmente aplicación a texto plano 24d⟩
   ⟨Escribir formalmente hueco a texto plano 24e⟩
   ⟨Escribir formalmente metainstrucción a texto plano 25a⟩))
```
Define:
escribir-expresión-formal, utilizado en fragmentos 24, 25, 35, y 45.
Utiliza escritor-formal 23a.

La representación simbólica de una variable es su nombre, ya que los nombres son internamente implementados como cadenas de caracteres es suficiente regresar este campo de la variable.

24b ⟨*Escribir formalmente variable a texto plano* 24b⟩≡ (24a)
```
(lambda (e nombre) nombre)
```

En el caso de las abstracciones se deben escribir de la forma $(\lambda x.M)$ donde x es la representación simbólica del argumento y M la representación simbólica del cuerpo. Para escribir el símbolo 'λ' se debe verificar si se puede usar su código en Unicode, de lo contrario usar el caracter '\'.

24c ⟨*Escribir formalmente abstracción a texto plano* 24c⟩≡ (24a)
```
(lambda (e argumento cuerpo)
  (format "(~a~a.~a)"
          (if hay-lambda? (integer->char 955) #\\)
          (escribir-expresión-formal argumento)
          (escribir-expresión-formal cuerpo)))
```
Utiliza escribir-expresión-formal 24a y hay-lambda? 13b.

Las aplicaciones son escritas de manera similar, invocando escribir-expresión-formal de manera recursiva sobre sus campos.

24d ⟨*Escribir formalmente aplicación a texto plano* 24d⟩≡ (24a)
```
(lambda (e operador operando)
  (format "(~a ~a)"
          (escribir-expresión-formal operador)
          (escribir-expresión-formal operando)))
```
Utiliza escribir-expresión-formal 24a.

La representación simbólica de los huecos siempre es [ ], así que simplemente se regresa esta cadena de caracteres.

24e ⟨*Escribir formalmente hueco a texto plano* 24e⟩≡ (24a)
```
(lambda (e) "[ ]")
```

Para escribir una metainstrucción se obtienen las representaciones simbólicas de cada argumento y se concatenan junto con la separación de coma entre cada argumento.

25a ⟨*Escribir formalmente metainstrucción a texto plano* 25a⟩≡ (24a)
```
(lambda (e nombre argumentos)
  (format "~a[~a]" nombre
          (apply string-append
                 (add-between (map escribir-expresión-formal argumentos)
                              ", "))))
```
Utiliza escribir-expresión-formal 24a.

El caso de prueba "Escritura formal a texto plano" verifica que la implementación de este escritor concreto sea correcta.

25b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂21b 30a▸
```
(test-case "Escritura formal a texto plano"
  (check-equal? (escribir-expresión-formal (parsear-cadena "foo"))
                "foo")
  (check-equal? (escribir-expresión-formal (parsear-cadena "\\foo.foo"))
                (format "(~afoo.foo)" (integer->char 955)))
  (check-equal? (escribir-expresión-formal
                 (parsear-cadena "foo bar (baz baz)"))
                "((foo bar) (baz baz))")
  (check-equal? (escribir-expresión-formal
                 (parsear-cadena "   [       ]"))
                "[ ]")
  (check-equal? (escribir-expresión-formal
                 (parsear-cadena "foo[\\bar.baz,baz bar]"))
                (format "foo[(~abar.baz), (baz bar)]" (integer->char 955))))
```
Utiliza escribir-expresión-formal 24a, parsear 14a, y parsear-cadena 14b.

Escritor formal a LaTeX La implementación básica del escritor formal a cadenas en texto plano es un buen punto de partida para definir escritores más especializados. Por ejemplo para escribir de manera formal a cadenas de caracteres con una sintáxis para el modo matemáticas de LaTeX se parte de la definición de escribir-expresión-formal para escribir 'λ' como \lambda, espacios pequeños como \, y espacios grandes como \quad.

25c ⟨*Escritores misceláneos* 25c⟩≡ (22) 26a▸
```
(define escribir-latex-expresión-formal
  (escritor-formal
   (lambda (e nombre) nombre)
   (lambda (e argumento cuerpo)
     (format "(\\lambda ~a.~a)"
             (escribir-latex-expresión-formal argumento)
             (escribir-latex-expresión-formal cuerpo)))
   (lambda (e operador operando)
     (format "(~a\\, ~a)"
             (escribir-latex-expresión-formal operador)
             (escribir-latex-expresión-formal operando)))
   (lambda (e) "[\\quad ]")
   (lambda (e nombre argumentos)
     (format "~a[~a]" nombre
             (apply string-append
                    (add-between (map escribir-latex-expresión-formal
```

```
                argumentos)
            ",\\, ")))))
```

Define:
escribir-latex-expresión-formal, nunca utilizado.
Utiliza escritor-formal 23a.

Con el procedimiento escribir-latex-expresión-formal sólo hace falta usar una plantilla de documento de LaTeX para poder generar un archivo interpretable válido. Por ejemplo, la variable plantilla-latex tiene como valor una cadena de caracteres de tal manera que si e es una expresión y p es un puerto de salida a un archivo en disco
(fprintf p plantilla-latex e)
escribe al archivo un documento compilable de LaTeX.

26a ⟨*Escritores misceláneos* 25c⟩+≡ (22) ◂25c 26b▸

```
(define plantilla-latex
#<<DOCUMENTO-LATEX

\documentclass[preview]{standalone}
\usepackage{amsmath}

\begin{document}
\( ~a \)
\end{document}

DOCUMENTO-LATEX
  )
```

Define:
plantilla-latex, nunca utilizado.

Escritura a otros lenguajes Traducir expresiones a LaTeX puede ser de utilidad para leer expresiones con tipografía más bonita. Sin embargo el algoritmo de escritura también puede ser utilizado para implementar escritores con poco valor estético pero grán valor práctico. Por ejemplo para escribir expresiones a JSON con la finalidad de transferir una expresión particular a un lenguaje con la habilidad de parsear este popular formato, se puede utilizar el procedimiento escribir-json-expresión-formal.

26b ⟨*Escritores misceláneos* 25c⟩+≡ (22) ◂26a 30b▸

```
(define escribir-json-expresión-formal
  (escritor-formal
   (lambda (e nombre)
     (format "{ ~s : ~s, ~s : ~s }"
             "tipo" "variable"
             "nombre" nombre))
   (lambda (e argumento cuerpo)
     (format "{ ~s : ~s, ~s : ~a, ~s : ~a }"
             "tipo" "abstracción"
             "argumento" (escribir-json-expresión-formal argumento)
             "cuerpo" (escribir-json-expresión-formal argumento)))
   (lambda (e operador operando)
     (format "{ ~s : ~s, ~s : ~a, ~s : ~a }"
             "tipo" "aplicación"
             "operador" (escribir-json-expresión-formal operador)
             "operando" (escribir-json-expresión-formal operando)))
```

```
(lambda (e)
  (format "{ ~s : ~s }"
          "tipo" "hueco"))
(lambda (e nombre argumentos)
  (format "{ ~s : ~s, ~s : ~s, ~s : [~a] }"
          "tipo" "metainstrucción"
          "nombre" nombre
          "argumentos"
          (apply string-append
                 (add-between (map escribir-json-expresión-formal
                                   argumentos)
                              ", "))))))
```

Define:
escribir-json-expresión-formal, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, escritor-formal 23a, hueco 5c, metainstrucción 6a, y variable 3b.

Constructor de escritores breves El algoritmo complicado de escritura es el que incluye abuso de notación. El parseo de expresiones admite abuso de notación parcial o total y para cualquier entrada se tiene una única salida, por otro lado en el caso de la escritura de expresiones con abuso, se tienen muchas posibles salidas para una misma entrada. En lambda se considera la escritura con abuso total de notación, es decir, si es posible aplicar un abuso de notación, se aplica.

De manera similar al constructor escritor-formal se implementa otro constructor llamado escritor-breve el cuál toma también procedimientos de escritura especializados para cada tipo de estructura. Los procedimientos especializados para las abstracciones y las aplicaciones deben satisfacer contratos diferentes a los de la escritura formal ya que estos dos tipos de expresiones son las que tienen abuso de notación.

27 ⟨*Algoritmos principales de escritura* 23a⟩+≡ (22) ◁23a

```
(define/contract (escritor-breve e-variable e-abstracción e-aplicación
                                 e-hueco e-metainstrucción)
  ((variable? string? . -> . any/c)
   ⟨Contrato para escritor formal de abstracciones 28a⟩
   ⟨Contrato para escritor formal de aplicaciones 28b⟩
   (hueco? . -> . any/c)
   (metainstrucción? string? (*list/c expresión?) . -> . any/c)
   . -> . (expresión? . -> . any/c))
  ⟨Algoritmo de escritura breve 28c⟩)
```

Define:
escritor-breve, utilizado en fragmentos 29, 30b, 37, y 47.
Utiliza abstracción 4c, aplicación 5a, expresión? 3a, hueco 5c, metainstrucción 6a, y variable 3b.

Consideremos una abstracción de la forma

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

tal que M no es abstracción. El abuso de notación nos permite escribirla como

$$\lambda x\, y\, z.M$$

en donde los cambios realizados son la omisión de paréntesis externos y la "descurrificación" de argumentos en abstracciones anidadas (abstracciones cuyos cuerpos son abstracciones). Por lo tanto el procedimiento de escritura breve de abstracciones debe conocer la lista de

argumentos de la descurrificación (en este ejemplo x, y y z) y el primer cuerpo en la anidación que no es abstracción (en este ejemplo M). La omisión de los paréntesis externos es la convención del abuso de notación, es posible que en una representación simbólica breve una abstracción esté encerrada entre paréntesis, sin embargo, la tarea de determinar si debe o no escribirse con paréntesis es del procedimiento de escritura de aplicaciones.

28a ⟨*Contrato para escritor formal de abstracciones* 28a⟩≡ (27)
```
(abstracción? (*list/c variable? variable?) expresión?
              . -> . any/c)
```
Utiliza abstracción 4c, expresión? 3a, y variable 3b.

En el caso de una aplicación de la forma

$$(((M\,N)\,O)\,P)$$

tal que M no es aplicación. El abuso de notación nos permite escribirla como

$$M\,N\,O\,P$$

en donde los cambios realizados son la omisión de paréntesis externos, remover la asociación a la izquierda y determinar cuáles de las expresiones M, N, O y P deben de tener paréntesis. Este último procedimiento considera únicamente el caso en que una expresión de la aplicación sea abstracción y otra aplicación: para las abstracciones, todas excepto la última (en este ejemplo P) debe tener paréntesis, mientras que en el caso de las aplicaciones, todas deben tener paréntesis. El procedimiento de escritura breve para aplicaciones debe conocer la lista de expresiones en la aplicación y la lista de criterios que determinan qué expresiones deben tener paréntesis.

28b ⟨*Contrato para escritor formal de aplicaciones* 28b⟩≡ (27)
```
(aplicación? (*list/c expresión? expresión? expresión?)
             (*list/c boolean? boolean? boolean?)
             . -> . any/c)
```
Utiliza aplicación 5a y expresión? 3a.

El algoritmo de escritura breve funciona de manera similar al de escritura formal, solo que debe extraer la información necesaria para los escritores de abstracciones y aplicaciones.

28c ⟨*Algoritmo de escritura breve* 28c⟩≡ (27)
```
(lambda (e)
  (cond [(variable? e)
         (e-variable e (variable-nombre e))]
        [(abstracción? e)
         (let itera ([cuerpo (abstracción-cuerpo e)]
                     [argumentos (list (abstracción-argumento e))])
           (if (abstracción? cuerpo)
               (itera (abstracción-cuerpo cuerpo)
                      (cons (abstracción-argumento cuerpo)
                            argumentos))
               (e-abstracción e (reverse argumentos) cuerpo)))]
        [(aplicación? e)
         (let itera ([operador (aplicación-operador e)]
                     [operandos (list (aplicación-operando e))]
                     [paréntesis (list (aplicación? (aplicación-operando e)))])
           (if (aplicación? operador)
```

```
                (let ([operando (aplicación-operando operador)])
                  (itera (aplicación-operador operador)
                         (cons operando operandos)
                         (cons (or (aplicación? operando)
                                   (abstracción? operando))
                               paréntesis)))
                (e-aplicación e (cons operador operandos)
                              (cons (or (aplicación? operador)
                                        (abstracción? operador))
                                    paréntesis))))]
          [(hueco? e)
           (e-hueco e)]
          [(metainstrucción? e)
           (e-metainstrucción e (metainstrucción-nombre e)
                              (metainstrucción-argumentos e))]))
```

Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, y variable 3b.

Escritor breve a texto plano La definición de un escritor breve a una cadena de caracteres en texto plano es relativamente fácil de implementar utilizando este algoritmo. Al igual que con `escribir-expresión-formal`, todos los escritores especializados regresan una cadena de caracteres.

29 ⟨*Escritores usuales* 24a⟩+≡ (22) ◃24a

```
(define escribir-expresión-breve
  (escritor-breve
   (lambda (e nombre) nombre)
   (lambda (e argumentos cuerpo)
     (format "~a~a.~a"
             (if hay-lambda? (integer->char 955) #\\)
             (apply string-append
                    (add-between (map escribir-expresión-breve argumentos)
                                 " "))
             (escribir-expresión-breve cuerpo)))
   (lambda (e operandos paréntesis)
     (apply string-append
            (add-between (map (lambda (o p)
                                (format "~a~a~a"
                                        (if p "(" "")
                                        (escribir-expresión-breve o)
                                        (if p ")" "")))
                              operandos paréntesis)
                         " ")))
   (lambda (e) "[ ]")
   (lambda (e nombre argumentos)
     (format "~a[~a]" nombre
             (apply string-append
                    (add-between (map escribir-expresión-breve argumentos)
                                 ", "))))))
```

Define:
  `escribir-expresión-breve`, utilizado en fragmentos 30a y 45.
Utiliza `escritor-breve` 27 y `hay-lambda?` 13b.

El caso de prueba "Escritura breve a texto plano" verifica que la implementación de este escritor sea correcta.

30a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂25b 32a▸
```
(test-case "Escritura breve a texto plano"
  (check-equal? (escribir-expresión-breve (parsear-cadena "foo"))
                "foo")
  (check-equal? (escribir-expresión-breve (parsear-cadena "\\foo.foo"))
                (format "~afoo.foo" (integer->char 955)))
  (check-equal? (escribir-expresión-breve
                 (parsear-cadena "foo bar (baz baz)"))
                "foo bar (baz baz)")
  (check-equal? (escribir-expresión-breve
                 (parsear-cadena "foo (bar baz) \\quux.quux quux"))
                (format "foo (bar baz) ~aquux.quux quux" (integer->char 955)))
  (check-equal? (escribir-expresión-breve (parsear-cadena " [    ]"))
                "[ ]")
  (check-equal? (escribir-expresión-breve
                 (parsear-cadena "foo[\\bar.baz,baz bar]"))
                (format "foo[~abar.baz, baz bar]" (integer->char 955))))
```
Utiliza escribir-expresión-breve 29, parsear 14a, y parsear-cadena 14b.

Escritor breve a LaTeX Para completar el ejemplo de escritura a LaTeX se presenta el escritor escribir-latex-expresión-breve utilizando el algoritmo que aplica abuso de notación.

30b ⟨*Escritores misceláneos* 25c⟩+≡ (22) ◂26b
```
(define escribir-latex-expresión-breve
  (escritor-breve
   (lambda (e nombre) nombre)
   (lambda (e argumentos cuerpo)
     (format "\\lambda ~a.~a"
             (apply string-append
                    (add-between (map escribir-latex-expresión-breve
                                      argumentos)
                                 "\\, "))
             (escribir-latex-expresión-breve cuerpo)))
   (lambda (e operandos paréntesis)
     (apply string-append
            (add-between (map (lambda (o p)
                                (format "~a~a~a"
                                        (if p "(" "")
                                        (escribir-latex-expresión-breve o)
                                        (if p ")" "")))
                              operandos paréntesis)
                         "\\, ")))
   (lambda (e) "[\\quad ]")
   (lambda (e nombre argumentos)
     (format "~a[~a]" nombre
             (apply string-append
                    (add-between (map escribir-latex-expresión-breve
                                      argumentos)
                                 ",\\, "))))))
```
Define:
escribir-latex-expresión-breve, nunca utilizado.

Utiliza escritor-breve 27.

2.3 Evaluación

Evaluación de expresiones Habiendo definido la sintaxis y las estructuras del lenguaje solo falta proveer un mecanismo para el análisis, manipulación y transformación de expresiones del lenguaje. Como se mencionó al implementar la estructura de las metainstrucciones, se registran procedimientos en una tabla hash llamada metainstrucciones los cuales esperan ser invocados con expresiones como argumentos.

La tarea de invocar metainstrucciones definidas la realiza un evaluador de expresiones, el algoritmo de evaluación consiste en recorrer las subexpresiones de una expresión en busca de invocaciones a metainstrucciones, al encontrar una metainstrucción se busca su nombre en el registro metainstrucciones, si su nombre es encontrado, se invoca su procedimiento asociado con las expresiones evaluadas de argumentos.

31a ⟨*Evaluación de expresiones* 31a⟩≡ (32b) 31b▸

```
(define/contract (evaluar-expresión e)
  (expresión? . -> . any)
  (cond [(abstracción? e)
         (abstracción (evaluar-expresión (abstracción-argumento e))
                      (evaluar-expresión (abstracción-cuerpo e)))]
        [(aplicación? e)
         (aplicación (evaluar-expresión (aplicación-operador e))
                     (evaluar-expresión (aplicación-operando e)))]
        [(metainstrucción? e)
         (cond [(hash-ref metainstrucciones (metainstrucción-nombre e) #f) =>
                (lambda (f)
                  (apply f (map evaluar-expresión (metainstrucción-argumentos e))))]
               [else
                (metainstrucción (metainstrucción-nombre e)
                                 (map evaluar-expresión
                                      (metainstrucción-argumentos e)))])]
        [else e]))
```

Define:
evaluar-expresión, utilizado en fragmentos 31b y 45.
Utiliza abstracción 4c, aplicación 5a, expresión? 3a, metainstrucciones 6c, y metainstrucción 6a.

De manera similar a la definición del procedimiento parsear-cadena se implementa evaluar-cadena para facilitar la invocación del evaluador.

31b ⟨*Evaluación de expresiones* 31a⟩+≡ (32b) ◂31a

```
(define/contract (evaluar-cadena s)
  (string? . -> . any)
  (evaluar-expresión (parsear-cadena s)))
```

Define:
evaluar-cadena, utilizado en fragmento 32a.
Utiliza evaluar-expresión 31a, parsear 14a, y parsear-cadena 14b.

Considerando metainstrucciones ficticias, el caso de prueba "Evaluación de expresiones" verifica que el evaluador funciona correctamente.

32a ⟨*Casos de prueba* 4a⟩+≡ (33) ◃30a

```
(define-metainstrucción (siempre-x . argumentos)
  (variable "x"))

(define-metainstrucción (encuentra-foo e)
  (define (busca-foo e)
    (cond [(or (variable? e) (hueco? e))
           #f]
          [(abstracción? e)
           (busca-foo (abstracción-cuerpo))]
          [(aplicación? e)
           (or (busca-foo (aplicación-operador e))
               (busca-foo (aplicación-operando e)))]
          [(metainstrucción? e)
           (if (string=? "foo" (metainstrucción-nombre e)) e #f)]))
  (let ([foo (busca-foo e)])
    (if foo foo e)))

(test-case "Evaluación de expresiones"
           (check-equal? (evaluar-cadena "y y siempre-x[y] y y")
                         (parsear-cadena "y y x y y"))
           (check-equal? (evaluar-cadena "\\x.encuentra-foo[x]")
                         (parsear-cadena "\\x.x"))
           (check-equal? (evaluar-cadena "\\x.encuentra-foo[foo[x]]")
                         (parsear-cadena "\\x.foo[x]"))
           (check-equal? (evaluar-cadena "siempre-x[encuentra-foo[foo[x]]]")
                         (parsear-cadena "x"))
           (check-equal? (evaluar-cadena "encuentra-foo[siempre-x[foo[x]]]")
                         (parsear-cadena "x")))
```

Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, evaluar-cadena 31b, hueco 5c, metainstrucción 6a, parsear 14a, parsear-cadena 14b, y variable 3b.

2.4 Código fuente

Los fragmentos de código presentados en este capítulo conforman un módulo de Racket llamado lambda-lib, este es dividido en dos partes: el archivo lambda-lib.rkt con la implementación del módulo y el archivo pruebas-lambda-lib.rkt el cuál contiene todos los casos de pruebas.

Archivo lambda-lib En este archivo se especifica el uso del lenguaje Racket y se proveen todas las definiciones presentes con (provide (all-defined-out)), el código principal de las tres secciones de este capítulo conforma el cuerpo del archivo, en orden.

32b ⟨*lambda-lib.rkt* 32b⟩≡

```
#lang racket
(provide (all-defined-out))
⟨Estructuras de las expresiones 11⟩
⟨Sintaxis del lenguaje 13a⟩
⟨Evaluación de expresiones 31a⟩
```

Archivo `pruebas-lambda-lib` Este archivo se conforma de todas las dependencias para los casos de prueba de este capítulo, una última dependencia es el archivo `lambda-lib.rkt` necesario para poder invocar los procedimientos definidos. El cuerpo de este archivo contiene todos los casos de prueba mencionados.

33 ⟨*pruebas-lambda-lib.rkt* 33⟩≡

```
#lang racket
⟨Dependencias de casos de prueba 4b⟩
(require "lambda-lib.rkt")
⟨Casos de prueba 4a⟩
```

Capítulo 3

Entorno interactivo

El sistema Lambda puede ser utilizado como biblioteca para otros programas que requieran trabajar con el cálculo λ, trabajando con Racket es suficiente importar el módulo `lambda-lib` o portando este módulo a otro lenguaje de programación y pasando las pruebas del archivo `pruebas-lambda-lib.rkt`.

En este capítulo se implementa un programa gráfico e interactivo para la exploración y estudio del cálculo λ puro. Primero se presentan dos mecanismos para la edición de expresiones, un editor de texto y un editor estructural. Podiendo editar expresiones se aborda un mecanismo para la evaluación de expresiones desde la interfaz gráfica. La segunda parte de la interfaz consiste en un historial de expresiones evaluadas, en donde se pueden observar las entradas y salidas. Para hacer más amena la experiencia de escribir expresiones se presenta un visualizador con tres maneras de visualizar la expresión siendo editada: en código de Racket, con escritura formal y con escritura breve. Otra componente de la interfaz es un registro de metainstrucciones utilizado como referencia en la escritura de expresiones. Finalmente, todos estos elementos son combinados en una interfaz gráfica única concretando la experiencia completa de explorar el cálculo λ.

Ya que la programación de interfaces gráficas no es el enfoque de este trabajo, se reduce al mínimo la explicación del funcionamiento de esta parte de Lambda.

3.1 Editores

Para construir expresiones en Lambda se pueden utilizar dos mecanismos: el editor de texto y el editor estructural.

3.1.1 Editor de texto

35 ⟨*Editor de texto* 35⟩≡ (56)

```
(define-lex-abbrevs
  [paréntesis (:or #\( #\) #\[ #\])]
  [delimitadores (:or #\, #\. #\\ #\u3BB #\;)])

(define lexer-lambda
  (lexer
   [(:+ (:~ delimitadores paréntesis whitespace))
    (values lexeme 'otro #f
            (position-offset start-pos)
            (position-offset end-pos))]
```

```
    [(:+ whitespace)
     (values lexeme 'white-space #f
             (position-offset start-pos)
             (position-offset end-pos))]
    [delimitadores
     (values lexeme 'delimitadores #f
             (position-offset start-pos)
             (position-offset end-pos))]
    [paréntesis
     (values lexeme 'delimitadores (string->symbol lexeme)
             (position-offset start-pos)
             (position-offset end-pos))]
    [(eof)
     (values lexeme 'eof #f #f #f)]))

(define editor-texto%
  (class color:text%
    (super-new)
    (define expresión null)
    (define/augment (after-insert start len)
      (if (= len 1)
          (let ([cadena (send this get-text start (+ start 1))])
            (unless (char-whitespace? (string-ref cadena 0))
              (parsear)))
          (parsear)))
    (define/augment (after-delete start len)
      (parsear))
    (define/public (parsear)
      (with-handlers ([exn:fail? (lambda (err) (set! expresión null))])
        (let ([e (parsear-cadena (send this get-text))])
          (if (eof-object? e)
              (set! expresión null)
              (set! expresión e))))
      (reportar-cambio-editor expresión))
    (define/public (obtener-expresión)
      expresión)
    (define/public (cambiar-expresión e)
      (send this erase)
      (send this insert (if (null? e) "" (escribir-expresión-formal e)))
      (parsear))
    (send this start-colorer
          (lambda (what)
            (case what
              [(delimitadores paréntesis) "Realzado"]
              [else "Sencillo"]))
          lexer-lambda
          (list (list '|(| '|)|)
                (list '|[| '|]|)))
    (send this set-tabs null 2 #f)
    ;; Estilos
    (define estilo-realzado (make-object style-delta% 'change-size 10))
    (send estilo-realzado set-weight-on 'bold)
    (send estilo-realzado set-family 'modern)
    (editor:set-standard-style-list-delta "Realzado" estilo-realzado)
```

```
(define estilo-sencillo (make-object style-delta% 'change-size 10))
(send estilo-sencillo set-family 'modern)
(editor:set-standard-style-list-delta "Sencillo" estilo-sencillo)
;; Atajos
(define atajos (new keymap%))
(define-syntax atajo
  (syntax-rules ()
    [(_ (nombre . argumentos) teclas cuerpo ...)
     (let ([nombre* nombre]
           [teclas* teclas])
       (send atajos add-function nombre*
             (lambda argumentos
               cuerpo ...))
       (send atajos map-function teclas* nombre*))]))
(for-each (lambda (normal griego)
            (atajo ((format "inserta-~a" griego) editor evento)
                   (format "m:~a" normal)
                   (send editor insert (format "~a" griego))))
          (string->list "abcdefghijklmnoprstuvwxyzCDFGJLPSUVW")
          '(#\u3b1 #\u3b2 #\u3c8 #\u3b4 #\u3b5 #\u3c6 #\u3b3 #\u3b7 #\u3b9 #\u3be
                   #\u3ba #\u3bb #\u3bc #\u3bd #\u3bf #\u3c0 #\u3c1 #\u3c3 #\u3c4
                   #\u3b8 #\u3c9 #\u3c2 #\u3c7 #\u3c5 #\u3b6 #\u3a8 #\u394 #\u3a6
                   #\u393 #\u39e #\u39b #\u3a0 #\u3a3 #\u398 #\u3a9 #\u3a3))
(atajo ("selecciona-todo" editor evento) "c:a"
       (send editor do-edit-operation 'select-all))
(atajo ("deshacer" editor evento) "c:z"
       (send editor do-edit-operation 'undo))
(atajo ("rehacer" editor evento) "c:y"
       (send editor do-edit-operation 'redo))
(atajo ("copiar" editor evento) "c:c"
       (send editor do-edit-operation 'copy))
(atajo ("pegar" editor evento) "c:v"
       (send editor do-edit-operation 'paste))
(atajo ("cortar" editor evento) "c:x"
       (send editor do-edit-operation 'cut))
(atajo ("borrar-línea" editor evento) "c:k"
       (send editor do-edit-operation 'kill))
(atajo ("enviar-expresión" editor evento) "c:enter"
       (enviar-expresión (send editor obtener-expresión)))
(send this set-keymap atajos)))
```

Define:
editor-texto%, utilizado en fragmento 51.
Utiliza escribir-expresión-formal 24a, parsear 14a, y parsear-cadena 14b.

3.1.2 Editor estructural

37 ⟨*Editor estructural* 37⟩≡ (56)

```
(define fuente-texto-figura
  (make-object font% 12 'modern 'normal 'bold))

(define (text* x)
  (text x fuente-texto-figura))
```

```
(define (expresión->dibujo expresión cursor índice)
  (define (p-fondo p color)
    (let ([ancho (pict-width p)]
          [alto  (pict-height p)])
      (cc-superimpose (color (filled-rectangle ancho alto)) p)))
  (define (p-nombre e nombre)
    (if (and (eq? e cursor) (or (variable? e) (metainstrucción? e)))
        (let ([n (string-length nombre)])
          (if (< índice n)
              (p-fondo (hbl-append (text* (substring nombre 0 índice))
                                   (frame (blue (text* (substring nombre índice
                                                                  (+ índice 1))))
                                          #:color "blue"
                                          #:line-width .5)
                                   (text* (substring nombre (+ índice 1) n)))
                       red)
              (p-fondo (hbl-append (text* nombre)
                                   (frame (blue (text* "·")) #:color "blue")) red)))
        (let ([p (text* nombre)])
          (p-fondo p (if (eq? e cursor) red white)))))
  (define (p-lambda e argumentos)
    (define en-cursor? #f)
    (define figuras (map (lambda (x)
                           (if (and (abstracción? cursor)
                                    (eq? x (abstracción-argumento cursor)))
                               (begin
                                 (set! en-cursor? #t)
                                 (frame (naive-layered (escribir-dibujo-breve x))
                                        #:color "red" #:line-width 2))
                               (naive-layered (escribir-dibujo-breve x))))
                         argumentos))
    (p-fondo (apply hbl-append (p-nombre (if en-cursor? cursor e) (string #\u3bb))
                    (text* "  ")
                    (add-between figuras (text* "  ")))
             white))
  (define (e-variable e nombre)
    (tree-layout #:pict (p-nombre e nombre)))
  (define (e-abstracción e argumentos cuerpo)
    (tree-layout #:pict (p-lambda e argumentos)
                 (tree-edge #:edge-width 2
                            (escribir-dibujo-breve cuerpo))))
  (define (e-aplicación e operandos paréntesis)
    (define en-cursor? #f)
    (define hijos (map (lambda (x)
                         (if (and (aplicación? cursor)
                                  (eq? x (aplicación-operando cursor)))
                             (begin
                               (set! en-cursor? #t)
                               (escribir-dibujo-breve x))
                             (escribir-dibujo-breve x)))
                       operandos))
    (apply tree-layout #:pict (disk 10
                                    #:color (if en-cursor? "red" "white")
                                    #:border-width 2)
```

```
            (tree-edge #:edge-width 2
                       #:edge-color (if en-cursor? "red" "gray")
                       (car hijos))
            (map (lambda (e x)
                   (tree-edge #:edge-width 2
                              #:edge-color
                              (if en-cursor?
                                  (if (eq? e (aplicación-operando cursor))
                                      (begin
                                        (set! en-cursor? #f)
                                        "red")
                                      "red")
                                  "gray")
                              x))
                 (cdr operandos)
                 (cdr hijos))))
  (define (e-hueco e)
    (tree-layout #:pict (p-nombre e "[ ]")))
  (define (e-metainstrucción e nombre argumentos)
    (define en-cursor? (eq? e cursor))
    (tree-layout
     #:pict (p-nombre e nombre)
     (tree-edge #:edge-width 2
                #:edge-color (if en-cursor? "red" "gray")
                (tree-layout
                 #:pict
                 (p-fondo
                  (apply ht-append
                         (append (list ((if en-cursor? red black) (text* "[ ")))
                                 (add-between
                                  (map (lambda (x)
                                         (naive-layered (escribir-dibujo-breve x)))
                                       argumentos)
                                  ((if en-cursor? red black) (text* " , ")))
                                 (list ((if en-cursor? red black) (text* " ]")))))
                  white)))))
  (define escribir-dibujo-breve
    (escritor-breve e-variable e-abstracción e-aplicación
                    e-hueco e-metainstrucción))
  (if (null? expresión)
      (blank 50)
      (naive-layered (escribir-dibujo-breve expresión))))

(define (escalar-dibujo dibujo ancho alto)
  (let* ([dibujo-ancho (pict-width dibujo)]
         [dibujo-alto (pict-height dibujo)])
    (scale dibujo
           (min (* (/ ancho dibujo-ancho) .9)
                (* (/ alto dibujo-alto) .9)))))

(define editor-estructural%
  (class canvas%
    (inherit refresh-now get-dc get-width get-height)
    (super-new)
```

```
(define expresión null)
(define cursor expresión)
(define índice 0)
(define atajos (new keymap%))
(define/override (on-char evento)
  (let ([código (send evento get-key-code)])
    (if (and (char? código)
             (or (char-alphabetic? código)
                 (char-numeric? código)
                 (member código (string->list "+-_~:<>=?¿!@#$%^&*") char=?))
             (not (send evento get-control-down))
             (not (send evento get-meta-down)))
        (send this inserta (send evento get-key-code))
        (send atajos handle-key-event this evento))))
(define/override (on-paint)
  (define dc (get-dc))
  (define editor-ancho (get-width))
  (define editor-alto (get-height))
  (define dibujo (escalar-dibujo (expresión->dibujo expresión cursor índice)
                                 editor-ancho editor-alto))
  (define dibujo-ancho (pict-width dibujo))
  (define dibujo-alto (pict-height dibujo))
  (draw-pict dibujo dc
             (- (/ editor-ancho 2) (/ dibujo-ancho 2))
             (- (/ editor-alto 2) (/ dibujo-alto 2))))
(define/public (obtener-expresión)
  expresión)
(define/public (inserta caracter)
  (when (or (variable? cursor) (metainstrucción? cursor))
    (let ([selector (if (variable? cursor)
                        variable-nombre
                        metainstrucción-nombre)]
          [mutador (if (variable? cursor)
                       set-variable-nombre!
                       set-metainstrucción-nombre!)])
      (if (= índice (string-length (selector cursor)))
          (mutador cursor
                   (string-append
                    (selector cursor)
                    (string caracter)))
          (mutador cursor
                   (string-append
                    (substring (selector cursor) 0 índice)
                    (string caracter)
                    (substring (selector cursor)
                               índice
                               (string-length (selector cursor)))))))
    (set! índice (+ índice 1))
    (send this refresh-now)))
(define/public (elimina)
  (when (and (or (variable? cursor) (metainstrucción? cursor))
             (> índice 0)
             (> (string-length ((if (variable? cursor)
                                    variable-nombre
```

```
                                    metainstrucción-nombre)
                                  cursor))
                1))
    (let* ([selector (if (variable? cursor)
                         variable-nombre
                         metainstrucción-nombre)]
           [mutador (if (variable? cursor)
                        set-variable-nombre!
                        set-metainstrucción-nombre!)]
           [nombre (selector cursor)]
           [n (string-length nombre)])
      (if (= índice n)
          (mutador cursor
                   (substring nombre 0 (- índice 1)))
          (mutador cursor
                   (string-append (substring nombre 0 (- índice 1))
                                  (substring nombre índice n)))))
    (set! índice (- índice 1))
    (send this refresh-now)))
(define/public (borrar)
  (set! expresión null)
  (set! cursor expresión)
  (set! índice 0)
  (send this refresh-now))
(define/public (cambiar-expresión e)
  (set! expresión e)
  (set! cursor expresión)
  (set! índice 0)
  (send this refresh-now))
;; Atajos
(define-syntax atajo
  (syntax-rules ()
    [(_ (nombre . argumentos) teclas cuerpo ...)
     (let ([nombre* nombre]
           [teclas* teclas])
       (send atajos add-function nombre*
             (lambda argumentos
               cuerpo ...))
       (send atajos map-function teclas* nombre*))]))
(atajo ("expresión-abajo" editor evento) "c:s"
       (cond [(abstracción? cursor)
              (set! cursor (abstracción-cuerpo cursor))
              (set! índice 0)
              (send editor refresh-now)]
             [(aplicación? cursor)
              (set! cursor (aplicación-operando cursor))
              (set! índice 0)
              (send editor refresh-now)]
             [(and (metainstrucción? cursor)
                   (not (null? (metainstrucción-argumentos cursor))))
              (set! cursor (car (metainstrucción-argumentos cursor)))
              (set! índice 0)
              (send editor refresh-now)]))
(atajo ("expresión-izquierda" editor evento) "c:a"
```

```
      (cond [(aplicación? cursor)
             (set! cursor (aplicación-operador cursor))
             (set! índice 0)
             (send editor refresh-now)]))
(atajo ("expresión-derecha" editor evento) "c:d"
      (cond [(abstracción? cursor)
             (set! cursor (abstracción-argumento cursor))
             (set! índice 0)
             (send editor refresh-now)]))
(letrec ([recorre
          (lambda (x)
            (cond [(variable? x) #f]
                  [(abstracción? x)
                   (if (or (eq? cursor (abstracción-cuerpo x))
                           (eq? cursor (abstracción-argumento x)))
                       x
                       (recorre (abstracción-cuerpo x)))]
                  [(aplicación? x)
                   (if (or (eq? cursor (aplicación-operador x))
                           (eq? cursor (aplicación-operando x)))
                       x
                       (cond [(recorre (aplicación-operador x))
                              => identity]
                             [else (recorre (aplicación-operando x))]))]
                  [(hueco? x) #f]
                  [(metainstrucción? x)
                   (if (memq cursor (metainstrucción-argumentos x))
                       x
                       (let itera ([argumentos (metainstrucción-argumentos x)])
                         (cond [(null? argumentos) #f]
                               [(recorre (car argumentos))
                                => identity]
                               [else (itera (cdr argumentos))])))]
                  [else #f]))])
  (atajo ("expresión-arriba" editor evento) "c:w"
         (cond [(recorre expresión)
                => (lambda (x)
                     (set! cursor x)
                     (set! índice 0)
                     (send editor refresh-now))]))
  (atajo ("expresión-siguiente" editor evento) "c:e"
         (cond [(recorre expresión)
                => (lambda (p)
                     (when (metainstrucción? p)
                       (let itera ([argumentos (metainstrucción-argumentos p)])
                         (cond [(null? argumentos) (void)]
                               [(null? (cdr argumentos)) (void)]
                               [(eq? (car argumentos) cursor)
                                (set! cursor (cadr argumentos))
                                (set! índice 0)
                                (send editor refresh-now)]
                               [else (itera (cdr argumentos))]))))]))
  (atajo ("expresión-anterior" editor evento) "c:q"
         (cond [(recorre expresión)
```

```
              => (lambda (p)
                   (when (metainstrucción? p)
                     (let itera ([argumentos (metainstrucción-argumentos p)])
                       (cond [(null? argumentos) (void)]
                             [(null? (cdr argumentos)) (void)]
                             [(eq? (cadr argumentos) cursor)
                              (set! cursor (car argumentos))
                              (set! índice 0)
                              (send editor refresh-now)]
                             [else (itera (cdr argumentos))])))))]))
(atajo ("variable-aquí" editor evento) "c:1"
       (let ([x (variable "x")])
         (cond [(recorre expresión)
                => (lambda (p)
                     (cond [(abstracción? p)
                            (if (eq? cursor (abstracción-argumento p))
                                (set-abstracción-argumento! p x)
                                (set-abstracción-cuerpo! p x))]
                           [(aplicación? p)
                            (if (eq? cursor (aplicación-operador p))
                                (set-aplicación-operador! p x)
                                (set-aplicación-operando! p x))]
                           [(metainstrucción? p)
                            (set-metainstrucción-argumentos! p
                              (map (lambda (e)
                                     (if (eq? e cursor) x e))
                                   (metainstrucción-argumentos p)))]))]
               [else
                (set! expresión x)])
         (set! cursor x)
         (set! índice 0)
         (send editor refresh-now)))
(atajo ("abstracción-aquí" editor evento) "c:2"
       (let ([x (abstracción (variable "x") (variable "x"))])
         (cond [(recorre expresión)
                => (lambda (p)
                     (cond [(abstracción? p)
                            (set-abstracción-cuerpo! p x)]
                           [(aplicación? p)
                            (if (eq? cursor (aplicación-operador p))
                                (set-aplicación-operador! p x)
                                (set-aplicación-operando! p x))]
                           [(metainstrucción? p)
                            (set-metainstrucción-argumentos! p
                              (map (lambda (e)
                                     (if (eq? e cursor) x e))
                                   (metainstrucción-argumentos p)))]))]
               [else
                (set! expresión x)])
         (set! cursor x)
         (set! índice 0)
         (send editor refresh-now)))
(atajo ("aplicación-aquí" editor evento) "c:3"
       (let ([x (aplicación (variable "x") (variable "x"))])
```

```
          (cond [(recorre expresión)
                 => (lambda (p)
                      (cond [(abstracción? p)
                             (set-abstracción-cuerpo! p x)]
                            [(aplicación? p)
                             (if (eq? cursor (aplicación-operador p))
                                 (set-aplicación-operador! p x)
                                 (set-aplicación-operando! p x))]
                            [(metainstrucción? p)
                             (set-metainstrucción-argumentos! p
                              (map (lambda (e)
                                     (if (eq? e cursor) x e))
                                   (metainstrucción-argumentos p)))]))]
                [else
                 (set! expresión x)])
          (set! cursor x)
          (set! índice 0)
          (send editor refresh-now)))
  (atajo ("hueco-aquí" editor evento) "c:4"
         (let ([x (hueco)])
          (cond [(recorre expresión)
                 => (lambda (p)
                      (cond [(abstracción? p)
                             (set-abstracción-cuerpo! p x)]
                            [(aplicación? p)
                             (if (eq? cursor (aplicación-operador p))
                                 (set-aplicación-operador! p x)
                                 (set-aplicación-operando! p x))]
                            [(metainstrucción? p)
                             (set-metainstrucción-argumentos! p
                              (map (lambda (e)
                                     (if (eq? e cursor) x e))
                                   (metainstrucción-argumentos p)))]))]
                [else
                 (set! expresión x)])
          (set! cursor x)
          (set! índice 0)
          (send editor refresh-now)))
  (let ([metainstrucción-aquí-n
         (lambda (n)
          (lambda (editor evento)
            (let ([x (metainstrucción "f" (build-list n (lambda (i)
                                                          (variable "x"))))])
              (cond [(recorre expresión)
                     => (lambda (p)
                          (cond [(abstracción? p)
                                 (set-abstracción-cuerpo! p x)]
                                [(aplicación? p)
                                 (if (eq? cursor (aplicación-operador p))
                                     (set-aplicación-operador! p x)
                                     (set-aplicación-operando! p x))]
                                [(metainstrucción? p) p
                                 (set-metainstrucción-argumentos! p
                                  (map (lambda (e)
```

```
                                                (if (eq? e cursor) x e))
                                              (metainstrucción-argumentos p)))]))]
                                [else
                                 (set! expresión x)])
                          (set! cursor x)
                          (set! índice 0)
                          (send editor refresh-now))))])
              (for-each (lambda (i)
                          (send atajos add-function (format "metainstrucción-aquí-~a" i)
                                (metainstrucción-aquí-n i))
                          (send atajos map-function
                                (format "c:5;~a" i)
                                (format "metainstrucción-aquí-~a" i)))
                        (range 0 10))))
      (atajo ("índice-derecha" editor evento) "right"
             (cond [(variable? cursor)
                    (when (< índice (string-length (variable-nombre cursor)))
                      (set! índice (+ índice 1))
                      (send editor refresh-now))]
                   [(metainstrucción? cursor)
                    (when (< índice (string-length (metainstrucción-nombre cursor)))
                      (set! índice (+ índice 1))
                      (send editor refresh-now))]))
      (atajo ("índice-izquierda" editor evento) "left"
             (cond [(or (variable? cursor) (metainstrucción? cursor))
                    (when (> índice 0)
                      (set! índice (- índice 1))
                      (send editor refresh-now))]))
      (for-each (lambda (normal griego)
                  (atajo ((format "inserta-~a" griego) editor evento)
                         (format "m:~a" normal)
                         (send editor inserta griego)))
                (string->list "abcdefghijkmnoprstuvwxyzCDFGJLPSUVW")
                '(#\u03b1 #\u03b2 #\u03c8 #\u03b4 #\u03b5 #\u03c6 #\u03b3 #\u03b7 #\u03b9 #\u03be
                          #\u03ba  #\u03bc #\u03bd #\u03bf #\u03c0 #\u03c1 #\u03c3 #\u03c4
                          #\u03b8 #\u03c9 #\u03c2 #\u03c7 #\u03c5 #\u03b6 #\u03a8 #\u394 #\u03a6
                          #\u393 #\u39e #\u39b #\u03a0 #\u03a3 #\u398 #\u03a9 #\u03a3))
      (atajo ("eliminar-texto" editor evento) "backspace"
             (send editor elimina))
      (atajo ("enviar-expresión" editor evento) "c:enter"
             (enviar-expresión (send editor obtener-expresión)))))
```

Define:
  editor-estructural%, utilizado en fragmento 51.
  escalar-dibujo, utilizado en fragmento 47.
Utiliza abstracción 4c, aplicación 5a, escritor-breve 27, hueco 5c, metainstrucción 6a, y variable 3b.

3.2 Historial

45 ⟨*Historial* 45⟩≡ (56)

```
(define expresión-snip%
  (class editor-snip%
    (init-field formal
                breve)
```

```
      (super-new)
      (define actual breve)
      (define editor (new text%))
      (send this set-editor editor)
      (refrescar)
      (define/override (on-event dc x y ex ey evento)
        (case (send evento get-event-type)
          [(left-down)
           (enviar-texto actual)
           (enfocar-editor)]
          [(right-down)
           (if (eq? actual breve)
               (set! actual formal)
               (set! actual breve))
           (refrescar)
           (enfocar-editor)]
          [else
           (enfocar-editor)]))
      (define/public (refrescar)
        (send editor erase)
        (send editor insert actual))))

  (define evaluación%
    (class object%
      (init-field entrada)
      (super-new)
      (define entrada-formal (escribir-expresión-formal entrada))
      (define entrada-breve (escribir-expresión-breve entrada))
      (define salida
        (with-handlers ([exn:fail? (lambda (err)
                                     (cons 'error (exn-message err)))])
          (evaluar-expresión entrada)))
      (define salida-formal
        (with-handlers ([exn:fail? (lambda (err)
                                     (format "~v" salida))])
          (if (and (pair? salida) (eq? 'error (car salida)))
              (cdr salida)
              (escribir-expresión-formal salida))))
      (define salida-breve
        (with-handlers ([exn:fail? (lambda (err)
                                     (format "~v" salida))])
          (if (and (pair? salida) (eq? 'error (car salida)))
              (cdr salida)
              (escribir-expresión-breve salida))))
      (define/public (obtener-entrada)
        entrada)
      (define/public (obtener-entrada-formal)
        entrada-formal)
      (define/public (obtener-entrada-breve)
        entrada-breve)
      (define/public (obtener-entrada-snip)
        (new expresión-snip% [formal entrada-formal] [breve entrada-breve]))
      (define/public (obtener-salida)
        salida)
```

```
    (define/public (obtener-salida-formal)
      salida-formal)
    (define/public (obtener-salida-breve)
      salida-breve)
    (define/public (obtener-salida-snip)
      (new expresión-snip% [formal salida-formal] [breve salida-breve]))))


(define historial%
  (class text%
    (super-new)
    (define evaluaciones null)
    (define bloqueado? #t)
    (define/augment (can-insert? s l) (not bloqueado?))
    (define/augment (can-delete? s l) #f)
    (define/public (insertar evaluación)
      (define (al-final x)
        (send this insert x (send this last-position)))
      (set! evaluaciones (cons evaluación evaluaciones))
      (set! bloqueado? #f)
      (al-final #\newline)
      (al-final (send evaluación obtener-entrada-snip))
      (al-final (new pict-snip% [pict (arrow 10 0)]))
      (al-final (send evaluación obtener-salida-snip))
      (al-final #\newline)
      (set! bloqueado? #t))
    (send this hide-caret #t)))
```

Define:
evaluación%, utilizado en fragmento 51.
expresión-snip%, nunca utilizado.
historial%, utilizado en fragmento 51.
Utiliza escribir-expresión-breve 29, escribir-expresión-formal 24a, y evaluar-expresión 31a.

3.3 Visualizador

47 ⟨*Visualizador* 47⟩≡ (56)

```
(define (expresión->código expresión [columnas 50])
  (codeblock-pict
   #:keep-lang-line? #f
   (string-append "#lang racket\n"
                  (pretty-format expresión columnas))))

(define (expresión->figura-escrita-formal expresión)
  (define (e-variable e nombre)
    (colorize (text nombre) (dark "green")))
  (define (e-abstracción e argumento cuerpo)
    (hbl-append (colorize (text (string-append "(" (string #\u3BB))) (dark "brown"))
                (escribir-dibujo-formal argumento)
                (colorize (text ".") (dark "brown"))
                (escribir-dibujo-formal cuerpo)
                (colorize (text ")") (dark "brown"))))
  (define (e-aplicación e operador operando)
    (hbl-append (colorize (text "(") (dark "brown"))
```

```
                (escribir-dibujo-formal operador)
                (text " ")
                (escribir-dibujo-formal operando)
                (colorize (text ")") (dark "brown"))))
  (define (e-hueco e)
    (colorize (text "[ ]") (dark "brown")))
  (define (e-metainstrucción e nombre argumentos)
    (apply hbl-append
           (colorize (text nombre) (dark "green"))
           (colorize (text "[") (dark "brown"))
           (append (add-between (map escribir-dibujo-formal argumentos)
                                (colorize (text ",") (dark "brown")))
                   (list (colorize (text "]") (dark "brown"))))))
  (define escribir-dibujo-formal
    (escritor-formal e-variable e-abstracción e-aplicación
                     e-hueco e-metainstrucción))
  (if (null? expresión)
      (blank 50)
      (escribir-dibujo-formal expresión)))

(define (expresión->figura-escrita-breve expresión)
  (define (e-variable e nombre)
    (colorize (text nombre) (dark "green")))
  (define (e-abstracción e argumentos cuerpo)
    (apply hbl-append
           (colorize (text (string #\u3BB)) (dark "brown"))
           (append (add-between (map escribir-dibujo-breve argumentos)
                                (text " "))
                   (list (colorize (text ".") (dark "brown"))
                         (escribir-dibujo-breve cuerpo)))))
  (define (e-aplicación e operandos paréntesis)
    (apply hbl-append
           (add-between (map (lambda (x p?)
                               (if p?
                                   (hbl-append (colorize (text "(") (dark "brown"))
                                               (escribir-dibujo-breve x)
                                               (colorize (text ")") (dark "brown")))
                                   (escribir-dibujo-breve x)))
                             operandos
                             paréntesis)
                        (text " "))))
  (define (e-hueco e)
    (colorize (text "[ ]") (dark "brown")))
  (define (e-metainstrucción e nombre argumentos)
    (apply hbl-append
           (colorize (text nombre) (dark "green"))
           (colorize (text "[") (dark "brown"))
           (append (add-between (map escribir-dibujo-breve argumentos)
                                (colorize (text ",") (dark "brown")))
                   (list (colorize (text "]") (dark "brown"))))))
  (define escribir-dibujo-breve
    (escritor-breve e-variable e-abstracción e-aplicación
                    e-hueco e-metainstrucción))
  (if (null? expresión)
```

```
      (blank 50)
      (escribir-dibujo-breve expresión)))

(define código-lienzo%
  (class canvas%
    (inherit refresh refresh-now get-dc get-width get-height)
    (super-new)
    (send this init-auto-scrollbars 1000 1000 0.0 0.0)
    (define expresión null)
    (define expresión-pict (blank 50))
    (define/override (on-paint)
      (define dc (get-dc))
      (draw-pict expresión-pict dc 0 0))
    (define/public (actualizar-expresión e)
      (set! expresión e)
      (set! expresión-pict
            (scale (expresión->código e (inexact->exact
                                         (floor (/ (get-width) 15))))
                   2))
      (send this refresh-now))))

(define figura-escrita-formal-lienzo%
  (class canvas%
    (inherit refresh refresh-now get-dc get-width get-height)
    (super-new)
    (define expresión null)
    (define expresión-pict (expresión->figura-escrita-formal expresión))
    (define/override (on-paint)
      (define dc (get-dc))
      (define lienzo-ancho (get-width))
      (define lienzo-alto (get-height))
      (define dibujo (escalar-dibujo expresión-pict
                                     lienzo-ancho lienzo-alto))
      (define dibujo-ancho (pict-width dibujo))
      (define dibujo-alto (pict-height dibujo))
      (draw-pict dibujo dc
                 (- (/ lienzo-ancho 2) (/ dibujo-ancho 2))
                 (- (/ lienzo-alto 2) (/ dibujo-alto 2))))
    (define/public (actualizar-expresión e)
      (set! expresión e)
      (set! expresión-pict
            (expresión->figura-escrita-formal e))
      (send this refresh-now))))

(define figura-escrita-breve-lienzo%
  (class canvas%
    (inherit refresh refresh-now get-dc get-width get-height)
    (super-new)
    (define expresión null)
    (define expresión-pict (expresión->figura-escrita-breve expresión))
    (define/override (on-paint)
      (define dc (get-dc))
      (define lienzo-ancho (get-width))
      (define lienzo-alto (get-height))
```

```
        (define dibujo (escalar-dibujo expresión-pict
                                       lienzo-ancho lienzo-alto))
        (define dibujo-ancho (pict-width dibujo))
        (define dibujo-alto (pict-height dibujo))
        (draw-pict dibujo dc
                   (- (/ lienzo-ancho 2) (/ dibujo-ancho 2))
                   (- (/ lienzo-alto 2) (/ dibujo-alto 2))))
      (define/public (actualizar-expresión e)
        (set! expresión e)
        (set! expresión-pict (expresión->figura-escrita-breve e))
        (send this refresh-now))))
```

Define:
código-lienzo%, utilizado en fragmento 51.
figura-escrita-breve-lienzo%, utilizado en fragmento 51.
figura-escrita-formal-lienzo%, utilizado en fragmento 51.
Utiliza abstracción 4c, aplicación 5a, escalar-dibujo 37, escritor-breve 27, escritor-formal 23a, hueco 5c, metainstrucción 6a, y variable 3b.

3.4 Registro de metainstrucciones

50 ⟨*Registro de metainstrucciones* 50⟩≡ (56)

```
(pretty-print-current-style-table
 (pretty-print-extend-style-table (pretty-print-current-style-table)
                                  '(define-metainstrucción match)
                                  '(define case)))

(pretty-print-columns 80)

(define metainstrucciones-código (make-hash))

(define-syntax define-metainstrucción
  (syntax-rules ()
    [(define-metainstrucción (id . args) cuerpo ...)
     (begin (define (id . args) cuerpo ...)
            (hash-set! metainstrucciones
                       (symbol->string 'id)
                       id)
            (hash-set! metainstrucciones-código
                       (symbol->string 'id)
                       '(define-metainstrucción (id . args) cuerpo ...)))]))

(define (cargar-archivo-metainstrucciones archivo)
  (let ([file (open-input-file archivo)])
    (let loop ([e (read file)])
      (unless (eof-object? e)
        (eval e)
        (loop (read file)))))
  (reportar-cambio-registro))

(define código-racket%
  (class racket:text%
    (super-new)
    (define bloqueado? #t)
```

```
    (define/augment (can-insert? s l) (not bloqueado?))
    (define/augment (can-delete? s l) #t)
    (define/public (insertar sexp)
      (set! bloqueado? #f)
      (send this erase)
      (let ([p (open-output-string)])
        (pretty-print sexp p 1)
        (let ([cadena (get-output-string p)])
          (for-each (lambda (x)
                      (send this insert x)
                      (send this insert-return))
                    (map string-trim (string-split cadena "\n"))))
      (set! bloqueado? #t)))))
```

Utiliza define-metainstrucción 6c, metainstrucciones 6c, y metainstrucción 6a.

3.5 Interfaz gráfica

51 ⟨*Uniendo las componentes de la interfaz gráfica* 51⟩≡ (56)

```
;; Ventana registro

(define ventana-registro
  (new frame%
       [label "Registro Metainstrucciones"]
       [width 800]
       [height 600]))

(define registro-panel
  (new panel:horizontal-dragable%
       [parent ventana-registro]
       [border 0]
       [spacing 20]))

(define listado-panel
  (new vertical-panel%
       [parent registro-panel]
       [spacing 20]))

(define listado-metainstrucciones
  (new list-box%
       [parent listado-panel]
       [label #f]
       [choices (list "")]
       [style (list 'single
                    'column-headers)]
       [columns (list "Nombre")]
       [callback
        (lambda (listado evento)
          (when (eq? 'list-box (send evento get-event-type))
            (let [(seleccionados (send listado get-selections))]
              (unless (null? seleccionados)
                (send código-texto insertar
                      (send listado get-data (car seleccionados)))))))]))
```

```
  (define botón-cargar
    (new button%
         [label "Cargar archivo"]
         [parent listado-panel]
         [stretchable-width #t]
         [callback
          (lambda (botón evento)
            (when (eq? 'button (send evento get-event-type))
              (let ([p (get-file "Cargar archivo de metainstrucciones")])
                (when p
                  (cargar-archivo-metainstrucciones (path->string p))))))]))

  (define código-texto (new código-racket%))

  (define lienzo-implementación (new canvas:color%
                                     [parent registro-panel]
                                     [editor código-texto]))

  (define ventana-visualizador
    (new frame%
         [label "Visualizador"]
         [width 800]  [min-width 200]
         [height 600] [min-height 200]))

  (define visualizador-panel
    (new vertical-panel%
         [parent ventana-visualizador]
         [border 0]
         [spacing 20]))

  (define visualizador-código
    (new código-lienzo%
         [parent visualizador-panel]
         [style (list 'vscroll 'hscroll)]))

  (define visualizador-formal
    (new figura-escrita-formal-lienzo%
         [parent visualizador-panel]
         [stretchable-height #f]
         [min-height 50]))

  (define visualizador-breve
    (new figura-escrita-breve-lienzo%
         [parent visualizador-panel]
         [stretchable-height #f]
         [min-height 50]))

  (define ventana-principal
    (new frame%
         [label "Lambda"]
         [width 800]  [min-width 200]
         [height 600] [min-height 200]))

  (define panel-principal
```

```
  (new vertical-panel%
       [parent ventana-principal]))

(define panel-herramientas
  (new horizontal-panel%
       [parent panel-principal]
       [border 0]
       [stretchable-height #f]))

(define botón-visualizador
  (new button%
       [parent panel-herramientas]
       [label "visualizador"]
       [callback
        (lambda (botón evento)
          (send ventana-visualizador show #t))]))

(define botón-metainstrucciones
  (new button%
       [parent panel-herramientas]
       [label "metainstrucciones"]
       [callback
        (lambda (botón evento)
          (send ventana-registro show #t))]))

(define herramientas-espacio
  (new panel%
       [parent panel-herramientas]))

(define botón-ayuda
  (new button%
       [parent panel-herramientas]
       [label "ayuda"]))

(define panel-historial/editor
  (new panel:vertical-dragable%
       [parent panel-principal]))

(define historial-panel
  (new panel%
       [parent panel-historial/editor]
       [border 0]))

(define historial (new historial%))

(define historial-lienzo
  (new editor-canvas%
       [parent historial-panel]
       [editor historial]))

(define panel-tabs
  (new tab-panel%
       [parent panel-historial/editor]
       [choices (list "Texto" "Estructura")]
```

```
        [callback
         (lambda (panel evento)
           (case (send panel get-selection)
             [(0)
              (send panel change-children
                    (lambda (x)
                      (list editor-texto-panel)))
              (reportar-otro-editor 'Texto)]
             [(1)
              (send panel change-children
                    (lambda (x)
                      (list editor-estructural-panel)))
              (reportar-otro-editor 'Estructura)]))]))

(define editor-texto-panel
  (new panel%
       [parent panel-tabs]
       [border 0]))

(define editor-texto (new editor-texto%))

(define editor-texto-lienzo
  (new editor-canvas%
       [parent editor-texto-panel]
       [editor editor-texto]))

(define editor-estructural-panel
  (new panel%
       [parent panel-tabs]
       [border 0]))

(define editor-estructural
  (new editor-estructural%
       [parent editor-estructural-panel]))

(send panel-tabs change-children
      (lambda (x)
        (list editor-texto-panel)))

;;
;; Procedimientos conectivos
;;

(define (enviar-expresión expresión)
  (send historial insertar
        (new evaluación% [entrada expresión]))
  (case editor-actual
    [(Texto)
     (send editor-texto erase)]
    [(Estructura)
     (send editor-estructural borrar)]))

(define (reportar-cambio-editor expresión)
  (send visualizador-código actualizar-expresión expresión)
```

```
    (send visualizador-formal actualizar-expresión expresión)
    (send visualizador-breve actualizar-expresión expresión))

  (define (enfocar-editor)
    (case editor-actual
      [(Texto)
       (send editor-texto-lienzo focus)]
      [(Estructura)
       (send editor-estructural focus)]))

  (define (enviar-texto cadena)
    (case editor-actual
      [(Texto)
       (send editor-texto erase)
       (send editor-texto insert cadena)]
      [(Estructura)
       (send editor-estructural borrar)
       (send editor-estructural cambiar-expresión
             (parsear-cadena cadena))]))

  (define editor-actual 'Texto)

  (define (reportar-otro-editor objetivo)
    (unless (eq? editor-actual objetivo)
      (case editor-actual
        [(Texto)
         (send editor-estructural cambiar-expresión
               (send editor-texto obtener-expresión))]
        [(Estructura)
         (send editor-texto cambiar-expresión
               (send editor-estructural obtener-expresión))]))
    (set! editor-actual objetivo)
    (enfocar-editor))

  (define (reportar-cambio-registro)
    (hash-for-each metainstrucciones
                   (lambda (nombre proc)
                     (send listado-metainstrucciones append
                           nombre (hash-ref metainstrucciones-código nombre)))
                   #t))

  ;;
  ;; Mostrar la ventana principal
  ;;

  (send ventana-principal show #t)
```

Utiliza código-lienzo% 47, editor-estructural% 37, editor-texto% 35, evaluación% 45, figura-escrita-breve-lienzo% 47, figura-escrita-formal-lienzo% 47, historial% 45, metainstrucciones 6c, parsear 14a, y parsear-cadena 14b.

3.6 Código fuente

56 ⟨*lambda-gui.rkt* 56⟩≡

```
#lang racket/gui
(require "lambda-lib.rkt")
(require framework)
(require (except-in parser-tools/lex
                    blank))
(require (prefix-in : parser-tools/lex-sre))
(require pict)
(require pict/snip)
(require pict/color)
(require pict/code)
(require pict/tree-layout)

⟨Editor de texto 35⟩
⟨Editor estructural 37⟩
⟨Historial 45⟩
⟨Visualizador 47⟩
⟨Registro de metainstrucciones 50⟩
⟨Uniendo las componentes de la interfaz gráfica 51⟩
```

Capítulo 4

Extensiones

En este capítulo se muestran ejemplos de metainstrucciones, estos procedimientos podrán ser incluídos por programas que usen `lambda-lib` como por ejemplo `lambda-gui`. Las metainstrucciones presentes se dividen en tres categorías: análisis de estructura de términos, transformación de términos y manejo de codificaciones.

4.1 Análisis de términos

57a ⟨*Metainstrucciones* 57a⟩≡ (60b) 57b▸

```
(define-metainstrucción (longitud M)
  (if (término? M)
      (let recurre ([M M])
        (match M
          [(variable x)
           1]
          [(abstracción x M)
           (+ 1 (recurre M))]
          [(aplicación M N)
           (+ (recurre M) (recurre N))]))
      -1))
```

Define:
`longitud`, utilizado en fragmento 20b.
Utiliza `abstracción` 4c, `aplicación` 5a, `define-metainstrucción` 6c, `metainstrucción` 6a, `término?` 7c, y `variable` 3b.

57b ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂57a 58a▸

```
(define-metainstrucción (variables-libres M)
  (match M
    [(variable x)
     (list (variable x))]
    [(abstracción x M)
     (remove x (variables-libres M))]
    [(aplicación M N)
     (remove-duplicates
      (append (variables-libres M)
              (variables-libres N)))]
    [_ null]))
```

Define:
`variables-libres`, utilizado en fragmento 58.

Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, y variable 3b.

4.2 Transformación de términos

58a ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂57b 58b▸

```
(define-metainstrucción (sustituye M x N)
  (match M
    [(== x)
     N]
    [(variable y)
     M]
    [(aplicación P Q)
     (aplicación (sustituye P x N) (sustituye Q x N))]
    [(abstracción (== x) P)
     M]
    [(abstracción y P)
     (cond [(not (member x (variables-libres P)))
            (abstracción y P)]
           [(not (member y (variables-libres N)))
            (abstracción y (sustituye P x N))]
           [else
            (let ([z (variable-ajena (aplicación N P))])
              (abstracción z (sustituye (sustituye P y z) x N)))])]
    [_ M]))
```

Define:
sustituye, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, variable 3b, variable-ajena 58b, y variables-libres 57b.

58b ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂58a 59a▸

```
(define-metainstrucción (variable-ajena M)
  (let ([variables (variables-libres M)])
    (let itera ([letra #\a])
      (if (char>? letra #\z)
          (let itera [(letra #\a) (número 2)]
            (if (char>? letra #\z)
                (itera #\a (+ número 1))
                (let ([v (variable (string-append (string letra)
                                                  (number->string número)))])
                  (if (member v variables)
                      (itera (integer->char (+ (char->integer letra) 1))
                             número)
                      v))))
          (let ([v (variable (string letra))])
            (if (member v variables)
                (itera (integer->char (+ (char->integer letra) 1)))
                v))))))
```

Define:
variable-ajena, utilizado en fragmento 58a.
Utiliza define-metainstrucción 6c, metainstrucción 6a, variable 3b, y variables-libres 57b.

59a ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂58b 59b▸
```
(define-metainstrucción (llenar-huecos M N)
  (match M
    [(variable x)
     M]
    [(abstracción x P)
     (abstracción x (llenar-huecos P N))]
    [(aplicación P Q)
     (aplicación (llenar-huecos P N)
                 (llenar-huecos Q N))]
    [(hueco)
     N]
    [(metainstrucción x Ps)
     (metainstrucción x (map (lambda (P)
                               (llenar-huecos P N)) Ps))]
    [_ M]))
```
Define:
llenar-huecos, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, hueco 5c, metainstrucción 6a, y variable 3b.

4.3 Codificaciones

59b ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂59a 59c▸
```
(define-metainstrucción (T)
  (parsear-cadena "\\x y.x"))

(define-metainstrucción (F)
  (parsear-cadena "\\x y.y"))
```
Define:
F, nunca utilizado.
T, nunca utilizado.
Utiliza define-metainstrucción 6c, metainstrucción 6a, parsear 14a, y parsear-cadena 14b.

59c ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂59b 60a▸
```
(define-metainstrucción (decodifica-numeral M)
  (match M
    [(abstracción (variable f)
                  (abstracción (variable x) M))
     (let itera ([M M] [n 0])
       (match M
         [(== (variable x)) n]
         [(aplicación (== (variable f)) M)
          (itera M (+ n 1))]
         [_ -1]))]
    [_ -1]))
```
Define:
decodifica-numeral, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, y variable 3b.

60a ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◃59c

```
(define-metainstrucción (codifica-número M)
  (match M
    [(variable x)
     (let ([n (string->number x)])
       (if (and (number? n) (integer? n) (>= n 0))
           (abstracción (variable "f")
                        (abstracción (variable "x")
                                     (let itera ([n n])
                                       (if (zero? n)
                                           (variable "x")
                                           (aplicación (variable "f")
                                                       (itera (- n 1)))))))
           M))]
    [_ M]))
```

Define:
codifica-número, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, y variable 3b.

60b ⟨*lambda-init.rkt* 60b⟩≡
⟨*Metainstrucciones* 57a⟩

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. The Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, 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. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 2 (Apr. 1936), 345–363.

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

[11] 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/2268573Electronic Edition.

[12] Crossley, J. N. *Reminiscences of logicians*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975, pp. 1–62.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] Knuth, D. E. Mathematics and computer science: Coping with finiteness. *Science 194*, 4271 (1976), 1235–1242.

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

[20] Mendelson, E. *Introduction to Mathematical Logic*, 5 ed. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2010.

[21] 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.

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

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

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

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

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

[27] Scott, D. S. Lambda calculus then and now. In *ACM Turing Centenary Celebration* (2012), ACM-TURING '12, ACM.

[28] 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.

[29] Steele, Guy Lewis, J. Lambda: The ultimate declarative. Tech. Rep. AIM-379, Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, 1976.

[30] 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.

[31] Steele, Guy Lewis, J., and Sussman, G. J. Lambda: The ultimate imperative. Tech. Rep. AIM-353, Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, 1976.

[32] Steele, Guy Lewis, J., and Sussman, G. J. Design of lisp-based processors, or scheme: A dielectric lisp, or finite memories considered harmful, or lambda: The ultimate opcode. Tech. Rep. AIM-514, Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, 1979.

[33] Stoy, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[34] Sussman, G. J., and Steele Jr, G. L. SCHEME : an interpreter for extended lambda calculus. Tech. Rep. AI 349, Massachussetts Institute of Technology (MIT). Cambridge (MA US), 1975.

[35] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 2*, 42 (1936), 230–265.

[36] Turner, D. A. A new implementation technique for applicative languages. *Software-Practice and Experience 9* (1979), 31–49.